Configuration of Network Parameters in Operating Systems using Machine Learning

Parameter Optimization through Genetic Algorithm

Bartosz Gembala

Thesis submitted for the degree of
Master in Network and system administration
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Configuration of Network Parameters in Operating Systems using Machine Learning

Parameter Optimization through Genetic Algorithm

Bartosz Gembala

Configuration of Network Parameters in Operating Systems using Machine Learning

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

By default, the Linux network stack is not configured for high-speed large file transfer. The reason behind this is to merely save memory resources. We may tune the Linux network stack by increasing network buffers size for high-speed networks that connect server systems in order to handle more network packets. There are also several other TCP/IP parameters that can be tuned in an OS. In this thesis, the goal is to make a system which learns from the history of the network traffic and leverages this knowledge to optimize the current performance by adjusting the parameters. This can be done for a standard Linux kernel using sysctl or /proc. For a VM, virtually any type of OS can be installed and an image can swiftly be compiled and deployed. By being a sandboxed environment, risky configurations can be performed without the danger of harming the system.

Different scenarios for network parameter configurations are thoroughly tested, and an increase of at least 65% throughput speed is achieved compared to the default configuration.

Acknowledgement

I would like to thank my supervisors Hårek Haugerud and Anis Yazidi for their support. They always allowed my work to be my own, but guided me in the right direction whenever I needed some help.

Also a big thank you to my family and friends for inspiration and their continuous support.

Contents

List of Figures

List of Tables

Listings

Preface

Abbreviations

GA	Genetic Algorithms
HAProxy	High Availability Proxy
KVM	Kernel-based Virtual Machine
ML	Machine Learning
MPM	Multi-Processing Modules
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIC	Network Interface Controller
RL	Reinforcement Learning
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1

Introduction

1.1 Motivation

By default, the Linux network stack is not configured for high-speed large file transfer, and this is in order save memory resources. But in a system where we do not mind sacrificing memory resources for some extra network speed, we want to maximize this gain to the largest possible extent.

The file sizes are getting bigger by the day, whereas there used to exist floppy disks that used to have two to three megabytes of space and it was still possible to store a game on it. Nowadays, we have games with tens or hundreds of gigabytes, same goes for the different video formats, now as 4K and 8K are getting more common, there is an ever-growing need of large file transfer in order to be able to stream those videos without waiting for the rendering. By just looking at YouTube, as of February 2017, there are more than 400 hours of content uploaded to YouTube each minute and one billion hours of content are watched on YouTube every day[16] [17]. This requires immense data throughput, which is what needs to increase in order to continue the evolution of streaming services.

During the early 2000s is when users had access to increased network bandwidth. It is those technological improvements that facilitated streaming of audio and video content to computer users in their homes and workplaces. That is when the use of protocols like TCP/IP, HTTP, HTML and the Internet became standardized, and this sector became more prioritized leading to more funds being used on this[18].

This has brought forth attention on how to improve the throughput. There is exponential number of parameter combinations that can be changed for the network, OS and Apache configurations. Genetic Algorithms are most often used to solve search and optimization problems and will here be used to discover the most efficient set of parameter combination for the given problem. The Genetic Algorithm will be working in such a way that the better configuration sets will have a higher chance of survival and through multiple rounds (generations) of selection, crossover

and mutation, the sets will improve, giving us a combination of parameters that contribute to reaching the goal. Genetic Algorithms are being used for the sake of efficiently searching through the immense number of available combinations, which else wise brute force testing of all different combinations is unfeasible.

1.2 Problem Statement

How to achieve a bigger network throughput for different payloads by using genetic algorithms, where network configurations change dynamically

A major part in choosing the right network configurations will be based on finding the relevant network parameters. Do we want to look mainly at the throughput for fast transfer of data, or do we want to give latency some level of importance as well. The default settings do not solve these problems in an optimal way, they are a good hybrid solution, but they are static with no regards to the traffic going through.

This project will be solving following problems:

- How to use genetic algorithm for optimizing the different network configuration parameters
- How to identify relevant network parameters
- How to make an adapted approach which can change these parameters dynamically
- How to make these network parameters change depending on the payload which is received
- In what extent are the reconfigured network parameters better than the default ones

It is no easy task to find the right configuration parameters, and among them, it is even harder to find the right combination, machine learning does help with that. A correct configuration setting can fully utilize the system resources and hence lead the system to the best quality of services (QoS) such as short request response time and high throughput.

1.3 Report structure

- Introduction - Describes the problem and challenges
- Background - The theory and technicality is explained
- Approach - How the problem statement is answered and what approaches are taken
- Result - Shows what was accomplished

- Discussion - Talk around what was proved / disproved
- Conclusion - Talk about how the result compare to what was already defined, what went right and what went wrong
- Further work - Discusses improvements that could have been made, as well as ideas for new features

Chapter 2

Background

2.1 Virtual Machines

Virtual machine (VM) is an emulation of a computer system. There are apps which can create virtualized environments called for virtual machines, that behave like a computer system without the hardware in it. It runs as a process on your current operating system or even in the cloud. Although VMs are great overall, they do add some overhead, demanding apps or games which require serious CPU power and graphics will not perform very well.

The reasons why one would create a virtual machine is because it offers a serious number of uses. For instance, it allows you to experiment with different types of OS, you can test out and see how it feels for you and delete it after use. Another reason is VM being sandboxed, software inside a VM will not get out to the rest of your system, hence making it a safe place to visit websites, test risky configurations etc. A VM is created by installing a hypervisor on a physical computer. The hypervisor does provide a layer of software abstraction, a virtual layer that helps to separate the underlying hardware and software above it. There are two types of hypervisors.

1. A Type 1 hypervisor (Bare Metal hypervisor) - runs on top of physical hardware.
2. A Type 2 hypervisor (Hosted supervisor) - runs inside an operating system which runs on physical hardware.

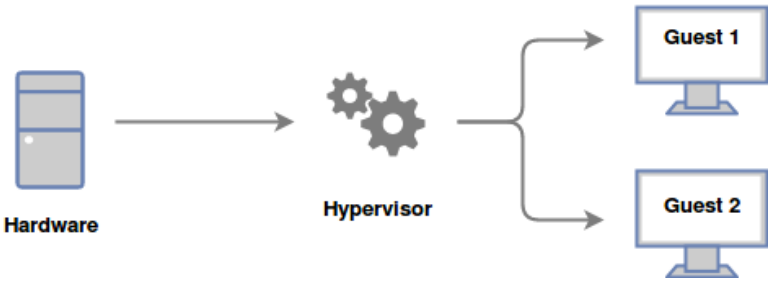

Figure 2.1: Type 1 hypervisor

Type 1 hypervisor is a lot more common since there is direct access to the underlying hardware which results in less overhead and has a bigger max capacity for the amount of VMs which can reside on a physical system.

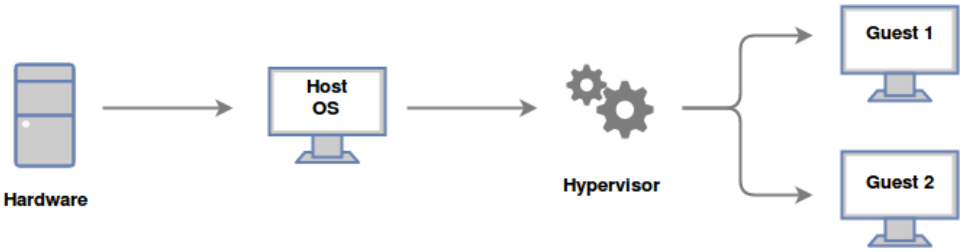

Figure 2.2: Type 2 hypervisor

Type 2 hypervisors run on an OS. A guest OS runs as a process on the host, and the overhead reduces available computing power and hence the max amount of VM which can reside inside a physical system is less than a type 1 hypervisor.

Advantages

- Multiple OS environments can coexist on the same machine, independent from each other
- Easy to duplicate, remove, availability and recovery
- Increases utilization of the available physical resources which leads to lower power consumption and less cooling demand.

Disadvantages

- Unstable performance if several virtual machines are running on a host computer, all depending on the size of the workload
- Not as efficient as a real machine
- If a virtualized server fails, then all VMs running on that server will become inaccessible

2.2 Protocols

2.2.1 TCP

A Transmission Control Protocol (TCP) connection is established through a three-way handshake 2.3, it is a process of where the user asks for permission to connect to host / server. When such connection is established, the data transfer may begin, and after the connection is closed, this link is terminated.

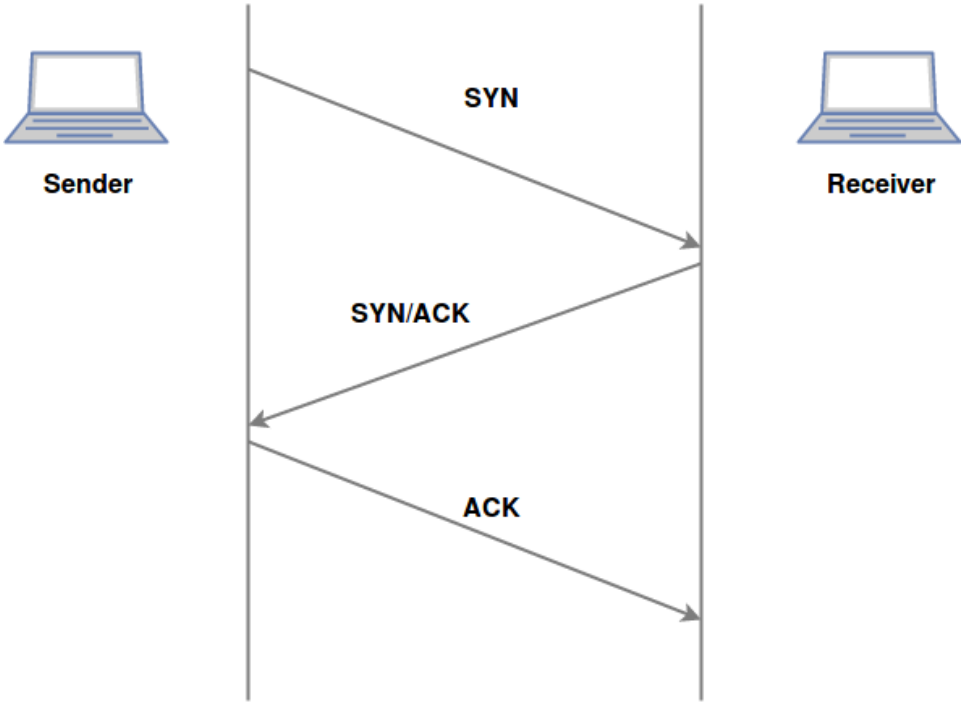

Figure 2.3: Three-way handshake

A three-way handshake is used in a TCP / IP network to establish a connection between host / client and server. As the name initiates, it is a three-step method where client and server exchange SYN and ACK packets.

1. Client sends a TCP SYN packet to the server
2. Server receives clients SYN
3. Server sends SYN-ACK
4. Client receives SYN-ACK
5. Client sends ACK
6. Server receives ACK

TCP socket connection is established

2.2.2 UDP

User Datagram Protocol (UDP) is a simpler connectionless protocol based on messages. There is no end-to-end connection, the way this is done to send information in one direction and hoping most of the packets will go through. It has a lot lower bandwidth overhead and latency.

2.2.3 Differences

TCP	UDP
Reliable: In a TCP connection a file or message will be sent unless the connection breaks. If the connection is lost, the server will request for the lost part, there is also no corruption under such data transfer	Unreliable: UDP is connectionless, so no handshake is required. When a file or message is sent one does not know if it will reach or it will be lost during the way, there is also chance for corruption.
Ordered: The messages sent over a connection will come in the same order you send them (1,2) -> (1,2)	Not ordered: Here the messages are received at any given order, depends on which message comes first e.g (1,2) -> (2,1)
Used: Apache TCP port 80, OpenSSH, FTP, SMTP - sending mail	Used: TFTP and online MMO games, Voice over IP, Tunneling / VPN
Stream: The data is read as a stream, can send multiple packets at once	Datagrams: The data is sent in packets one by one, packets can be lost or come unordered, but the content in every each one of the packets is whole when it arrives
Heavy: If some parts of the TCP stream comes in wrong order, resend requests are sent and all the parts are reordered to be in the same order as originally, that takes a bit toll on the system	Light: There is no tracking between connections, an no order of messages. Which means one does not have to rearrange anything and hence it is quicker and lighter on the system

2.3 Load balancer

There are various ways to set up a load balancer[24], and there are different algorithms which provide different benefits [26], the choice of load balancing methods depends on your needs:

- **Round Robin** - Requests are distributed across the group of servers in a sequence

- **Least Connections** - A new request is sent to the server with the fewest current connections to the client.
- **IP Hash** - IP address of a client us used to determine which server receives a request

Let's first look at how a setup with no load balancer works.

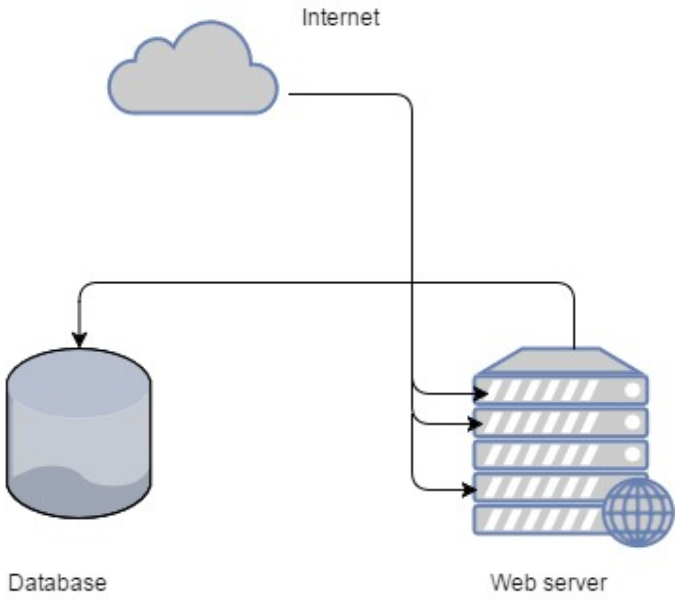

Figure 2.4: No load balancer

The problem with no load balancer is that when the web server gets too heavy load it can crash and shut down, hereby the user will be denied access to the database. This can happen if a server originally supports hundreds or thousand of requests, but suddenly this number is quadrupled or even in the range of a million, and if the server can not handle the load, it will shut down. This is where we are in need of a loadbalancer which can distribute the load evenly across different web-servers.

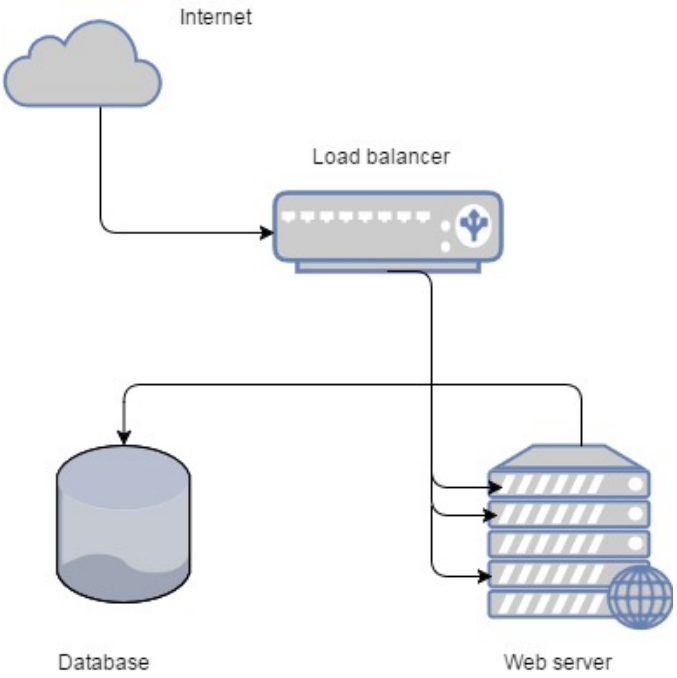

Figure 2.5: Load balancer

We can avoid this problem by implementing a load balancer, then we can distribute the load across different servers and therefore handle a bigger load. This way if a web server crashes, the load balancer will redirect us to another working web server, and we can still access our database. We will be using the round robin algorithm so the order will be chosen sequentially, nothing too fancy behind this. All the traffic will now have to go through this one load balancer, which can cause problem if there is a lot of traffic this can get pretty slow, even worse if the load balancer crashes, then we will not have access to anything. Which leads us to another solution, now with mutliple load balancers.

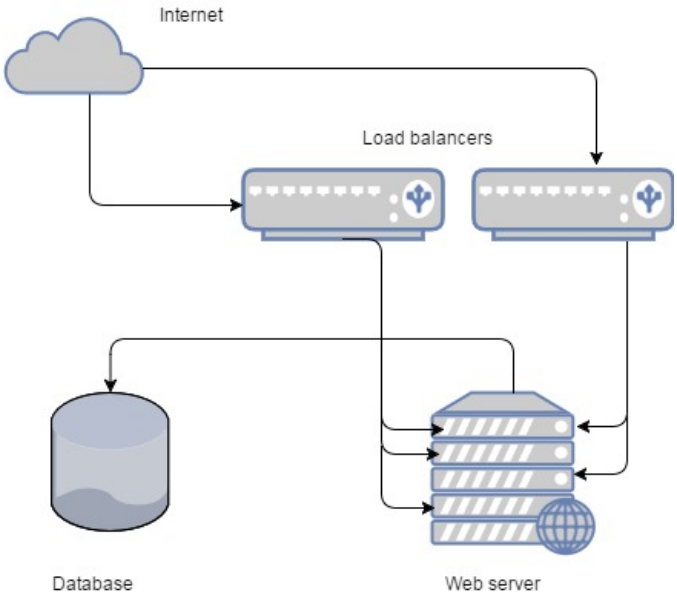

Figure 2.6: Multiple load balancers

As seen above, now we will avoid the previously mentioned problem, we will get rid of the bottleneck and can distribute the traffic twice as fast. If a load balancer dies, the other will still be up and running, this way the user will still have access to the database, although the traffic flow will get slower.

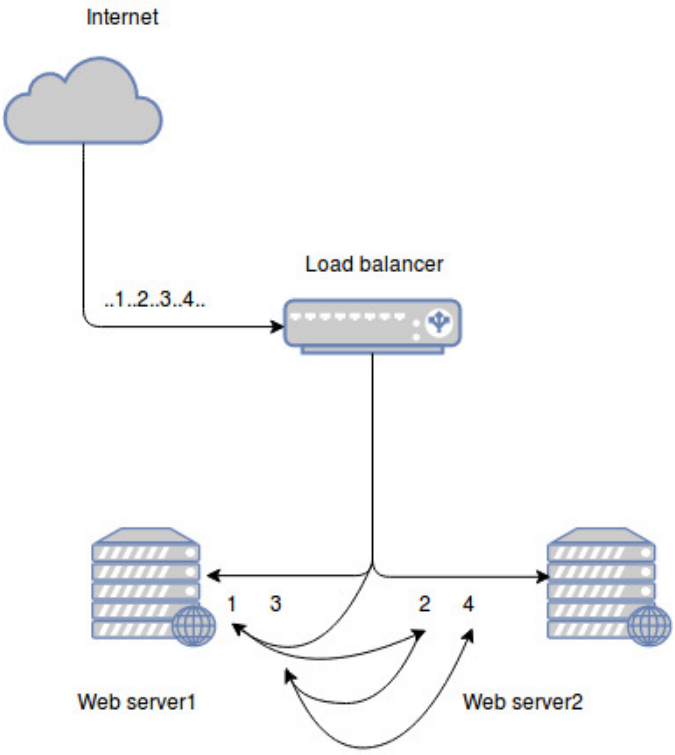

Figure 2.7: Round robin

Round robin is the simplest and most straightforward one, easy to implement and not too difficult to maintain, producing more than satisfactory results. As mentioned by H. Bryhni et al [25], round robin is one of the most efficient algorithms. They way it works is shown in fig 2.7, let's say we got four incoming requests, load balancer will distribute it sequentially, it will first redirect the request to the first webserver, then to the second and over again, that way every other request will be sent somewhere else and the load will be distributed pretty evenly.

2.3.1 HAProxy

HAProxy [23] stands for High Availability Proxy and is a popular open source software TCP / HTTP Load Balancer and proxying solution that can be run on Linux, FreeBSD and Solaris. The main purpose of this is to improve the reliability, performance and stability of a server by distributing the given workload around on the different servers linked to it. Basically by spreading requests across multiple servers. It is very well suited for web sites that have a very high factor of traffic, and is used among quite a few world's most visited sites, like GitHub, Stack Overflow, Reddit, Twitter etc.
.

Health check
HAProxy uses health checks to decide if a backend server is available to process requests. Normally this is done through TCP, establishing connection to the server and checks if the backend server is listening on the configured IP address on port. If this check fails on a server, that server will automatically be disabled in the backend, until it becomes healthy again.

2.4 A/B testing

A / B testing also known as split testing or bucket testing is a method to compare two variants, A and B against each other to determine which one of these has better performance. A / B testing is an experiment where different versions are shown to users at random and by using statistical analysis it is determined which version performs better.

As the name implies, two versions are compared to each other, which are identical except for very small variations that could affect a users behaviour. In an A / B test, you take a web page or an app screen and create another version of it. This change can be something very small as a picture or a title, but also a complete change of the design. Half of the traffic coming through will be redirected to the original version, while the other half will be sent to the new version.

While the users engage with those different versions, their statistics is gathered and analyzed through a statistical analysis. It can then be determined in what grade the changes affect the users, either negative or positive. By testing different versions, marketers can learn which variation attracts customers best, and the user experience can be optimized.

The A / B testing process can be divided into multiple stages:

- **Collect data:** Analytics will often provide insight in whereas the problem areas are. Looking at pages with low conversion rates or high bounce rates.
- **Observe user behavior:** Tools like heat maps or visitor recordings could be utilized in order to see what is stopping the users from converting.
- **Hypothesis:** From the observation of user behavior a hypothesis can now be deducted aiming to increase the amount of users.
- **Test:** The hypothesis needs to be tested along with the original page, an A / B test is performed.
- **Analyze test data:** A / B test results data is run through and analyzed to see which of the variations performed the best. If there was a clear winner among the variation, this variation can now be implemented. If the test results are unclear, a new hypothesis needs to be reworked.

This can potentially be used with parameter combinations on different host-machines, testing the best found "frozen" combination at a given time to see if the throughput traffic can be optimized. By frozen being the last best found configuration, even some generations old. Two host machines could be testing different configurations while receiving the same payload with the help of a load balancer. The one having the best speed, could tell the server and a further optimization of that parameter combination could be performed. Once the bandwidth speed falls under a certain threshold, another parameter combination should be found and tested, such a fall in throughput speed could be caused by change in type of traffic coming through.

2.5 Python

Python[27] is a high level programming language for general purpose programming. It supports object-oriented-, imperative-, functional- and procedural-programming also has a very big library. Python might not be as efficient as lower level languages like java or C, but in return offers a great variety of possibilities for programmers.

According to a research [28] done by data scientist Jean-Francois Puge, where data was gathered through indeed.com, which looks for trend searches of selected terms in job offers, comparing different popular machine languages such as python, R, C, javascript etc. There is an observation of who the clear leader is, python, it is one of the reasons for using python as the main programming language in this thesis. The philosophy of python is simple, as described in the document Zen of Python[29], including following lines:

- Beautiful is better than ugly
- Simple is better than complex
- Complex is better than complicated
- Readability counts
- Errors should never pass silently

2.6 Apache

Apache is a free and open-source cross-platform web server software. It is one of the most widely used web server software currently available. A web server is like your host in a restaurant. It greets you, takes your order and shows you your table. Just like the host, a web server checks for the web page you have requested and shows it to you in order for you to view the page. Webserver is both a host and a server, once you have found what you want it will serve you the page. All the communication is being taken

care of by through web server.

Apache provides a diversity of MultiProcessing Modules (MPMs), that allows Apache to be run in hybrid, event-hybrid or process-based mode, that provides an improved way of matching the demands for various infrastructures. Hence it means that it is important to choose good configurations and MPM.

2.7 Genetic algorithms

Genetic algorithms (GA) are inspired by the process of natural selection and are a version of evolutionary algorithms (EA) [9]. Genetic algorithms are most often used to solve search and optimization problems by generating high quality solutions using biologically inspired operators like selection, crossover and mutation. The common underlying idea behind this is: given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (survival of the fittest). This results in a rise of fitness of the population [7]. Then based upon fitness of the candidates the better candidates are chosen to seed the next generation. After x amount of generations we will come to a point where we are not getting considerable fitter offspring, that is when we stop.

2.7.1 Genetic operators

Genetic operator is an operator used in genetic algorithms that leads the algorithm toward a solution. Those three type of operators are selection, crossover and mutation operators, they are used together with each other and contribute to the algorithm doing its purpose. Those operators contribute to crossing existing solutions into new solutions (crossover), diversifying the population (mutation) and selecting the best solutions for every generation (selection).

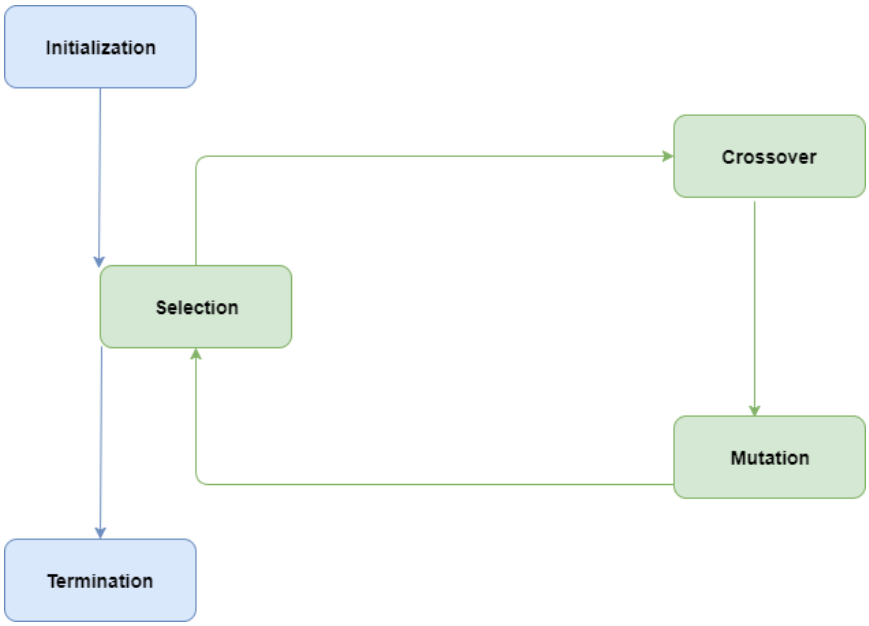

Figure 2.8: Algorithm lifecycle

And as shown in figure 2.8, a population is being "sent" for evaluation, the fittest parents are chosen and a crossover operation is being performed on parents depending on the algorithm used for crossover. Another algorithm for mutation is also present, most often there is a slight chance for a mutation for the offspring, thus from time to time an offspring with a genetic variation will be born in order to ensure diversity which can contribute to a fitter population in the long run. At the end of the cycle new parents will be chosen depending on their fitness (and possibly their age), or some other criteria, thus we get some new parents from the offspring and keep the old parents as well.

2.7.2 Representation

In genetic algorithms a representation (fig 2.9) is a way to present individuals, it can show their behaviour, different qualities, appearance etc. The array data type is called for a chromosome, each chromosome consists of multiple genes, while the possible values for each gene are called alleles. A programmer can represent all individuals in a population in many different ways, some of them being:

- Binary
- Integer
- Real-valued or floating-point
- Permutation
- Tree

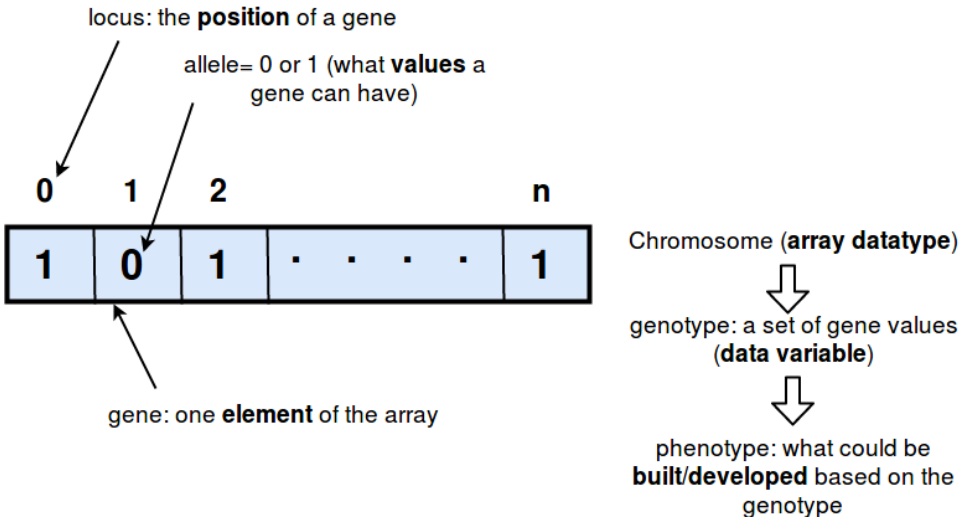

Figure 2.9: Genotype Representation

Binary

The simplest way of representing chromosomes is binary representation. A chromosome will then be represented by genes with the allele value of 0 and 1. The genotype will be consisting a string of binary numbers. There are various problems that use binary representations, some problems consist of Boolean decision variables like yes or no, while others that deal with numbers could represent numbers with their binary representation.

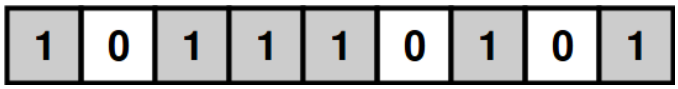

Figure 2.10: Binary representation

Integer

But binary representations is not always the most suitable representation if in our problem different genes can have different values from within a certain set of values. One such example would be to find the optimal integer values out from a set of integers, whereas those values could by any integer value possible, or even restricted, such like 0,1,2,3,4 representing colors or such.

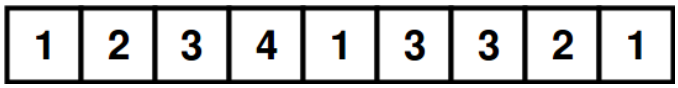

Figure 2.11: Integer representation

Real-Valued or Floating-Point

For problems where we want to use continuous instead of discrete values, a real-valued representation is in that case most natural. Values could represent physical quantities such as length or weight within values that are less than integer values. The limitation of these real-valued or floating-point numbers are limited to the computer.

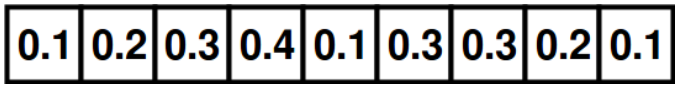

Figure 2.12: Float / real-value representation

Permutation

Another way to represent problems are with permutation, where the order of elements matter. A very known example of such a problem is the travelling salesman problem (TSP). The goal for the salesman is to visit each city, but simultaneously use the shortest route possible. The solution of the problem is presented by an ordering of all the cities and therefore a permutation representation is the wisest choice.

Figure 2.13: Permutation representation

Tree

Trees are one of the most commons structures to represent objects in computing. They represent expressions in a given formal syntax depending on the problem at hand, for example an arithmetic formula:

$2 * p + ((x + 5) - \frac{y}{3+2})$

Figure 2.14 is a representation of the formula above, this example illustrates how a parse tree can be used to present the equation.

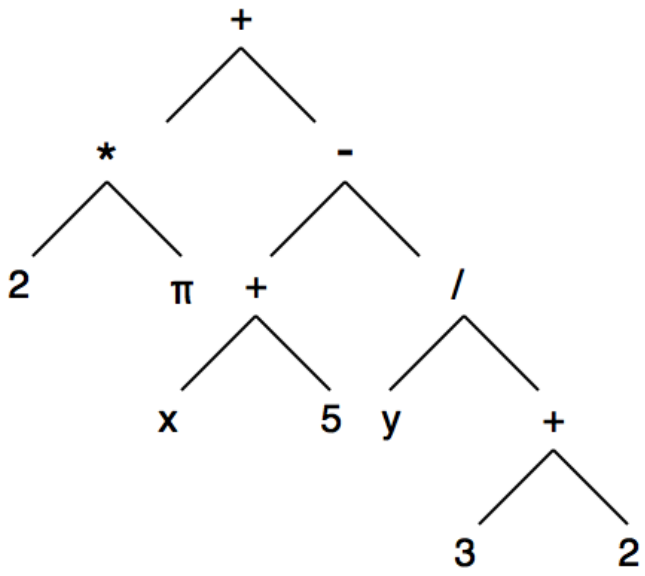

Figure 2.14: Tree representation

2.7.3 Crossover

Crossover or recombination is an operation that merges information from two parent genotypes into one or two offspring genotypes. Crossover is a stochastic operator where what part of each parent will be combined are decided by the algorithm. The main idea behind this is to partner up two different individuals with desirable features to create an offspring which combines those features. This has been done over millennial to plants or livestock breeders to produce species that give higher yield or have other desirable features [7].

One Point Crossover

A random crossover point is selected and the tails of those two parents are swapped and new offspring is thereby created. Another way to do this is by a two point crossover, which is similar to 2.15, but instead of one crossing point, there are two, and the parents are split into three.

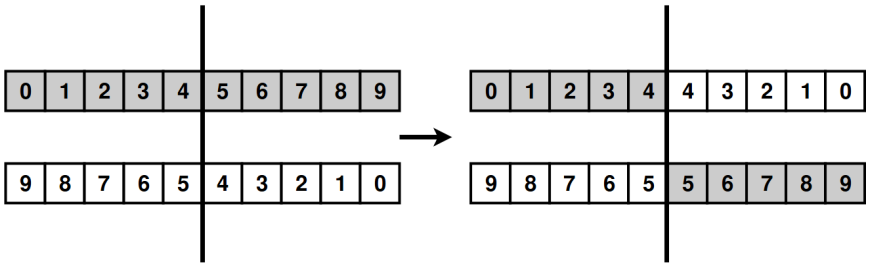

Figure 2.15: One Point Crossover

Uniform crossover

To difference to the previous method, the uniform crossover rather lets the parents contribute with gene level instead of segment level. Mixing ratio is chosen, and if we choose it to be 0.5, the offspring will have roughly half of the genes from the first parent and the second parent. The uniform crossover will evaluate each bit in the parent string for a swap with the given probability.

Ordered crossover

With ordered crossover it is very important that we do not remove or add any genes. An example of where we can use ordered crossover is for the travelling salesman problem, the locations in the route should only be shuffled and not removed, otherwise we could risk in making an invalid solution. In ordered crossover a subset from the first parent will be selected, this subset will be add to the offspring. Any missing values will be added to the offspring from the second parent in the same order which they are found, but no location should appear more than once.

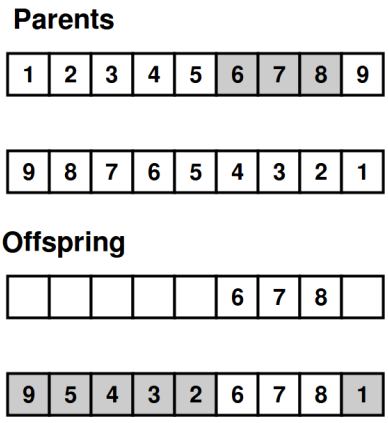

Figure 2.16: Ordered crossover

A subset of the genotype is selected, here being (6,7,8) and added to the offspring. The empty gene spots are added in order of the second parent. For any already existing location in the offspring, one jumps to the next one and selects it instead, inserts if it does not already exist in the offspring or else repeat.

2.7.4 Mutation

Mutation is a genetic operator which contributes to diversity from one generation to another in genetic algorithms. The goal is to change one or multiple gene values to something different, the mutation probability is defined in the algorithm.

Bitwise Mutation

The mutation of bits happen through bit flips at chosen locations, the selected bits get inverted as shown in fig 2.17. Each gene is considered separately with a small probability p_m.

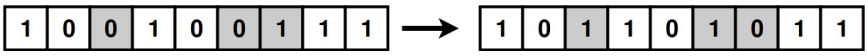

Figure 2.17: Bitwise Mutation

Swap Mutation

In swap mutation two genes in a chromosome are selected at random and their values swapped. As it can be seen in fig 2.18 where one and nine has been swapped.

Figure 2.18: Swap Mutation

Insert Mutation

Two genes are selected at random and the last one is moved right beside the first one, pushing and shuffling all the others along to make space (fig 2.19).

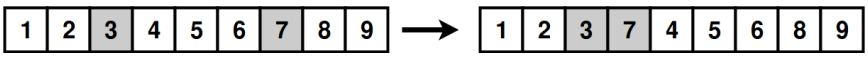

Figure 2.19: Insert Mutation

Scramble Mutation

A randomly chosen subset of values from the chromosome gets chosen and all the values within get scrambled. The positions for the genes in the subset gets randomly swapped with each other (fig 2.20).

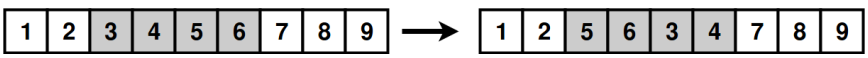

Figure 2.20: Scramble Mutation

2.7.5 Selection

The selection operator gives better individuals stronger preference and allowing them to pass their genes on to the next generation. How good an individual is depends on their fitness, and there are different methods to choose from like fitness proportional selection, ranking selection, tournament selection etc., for every method there is a different criteria of what being fit is.

Parent selection

Fitness Proportional Selection is when the probability that an individual is selected depends on its absolute fitness value compared to other individuals in the population. Every individual has a chance to be selected, but more fitter candidates are more likely to be selected. An individuals survival probability is a function of its fitness score.

Roulette wheel

One implementation of the fitness proportional selection is roulette wheel selection 2.21. A fixed point is placed while the wheel is rotated, and the region which the fixed point (black arrowhead) points at, will be the parent, giving the individual with highest fitness the biggest probability of being chosen. And for the second parent we repeat the process and spin the wheel again.

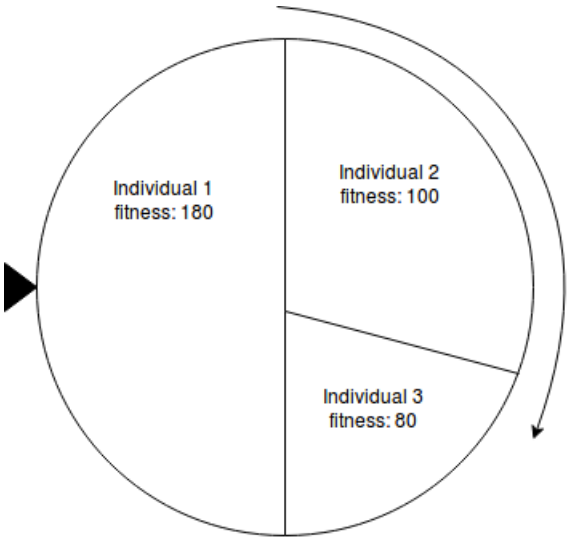

Figure 2.21: Roulette wheel

```
BEGIN
 /* Given the cumulative probability distribution a */
 /* and assuming we wish to select / members of the mating pool */
 set current member = 1;
 WHILE ( current member   / ) DO
    Pick a random value r uniformly from [0, 1];
```

```
    set i = 1;
    WHILE ( a_i < r ) DO
     set i = i + 1;
    OD
    set mating pool[current member] = parents[i];
    set current member = current member + 1;
 OD
END
```

Listing 2.1: Pseudocode for roulette wheel algorithm

Another implementation of the fitness proportional selection is stochastic universal sampling [8] which works in a very similar way, except instead of just one fixed point, there is two, and both parents are chosen at once, this setup encourages the highly fit individuals to be chosen at least once.

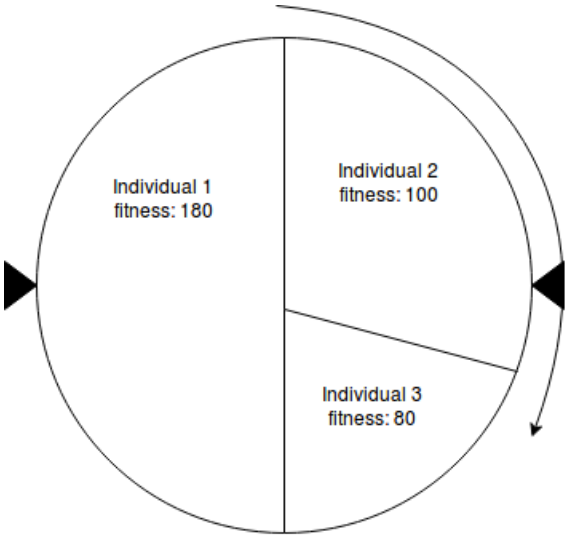

Figure 2.22: Stochastic universal sampling

Tournament selection

Tournament selection is a method for selecting individuals from a population where several "tournaments" are being held between random individuals from the population. The winner of each tournament, individual with best fitness will be selected. The only things that is being compared is rank of any two individuals and therefore this method is fast and simple to implement. Tournament selection works on the rule to select l members of a pool of m individuals.

```
BEGIN
 /* Assume we wish to select l members of a pool of m individuals
    */
 set current member = 1;
 WHILE ( current member   l ) DO
    Pick k individuals randomly, with or without replacement;
```

```
6       Compare these k individuals and select the best of them;
7       Denote this individual as i;
8       set mating pool[current member] = i;
9       set current member = current member + 1;
10   OD
11  END
```

Listing 2.2: Pseudocode for tournament selection algorithm

Survivor selection

There are various number of strategies which help choosing which m of the m parents + l offspring should go forward to the next generation.

- **Replace worst** is where the worst m members of the population are replaced. This can contribute to fast improvements in population, but also to too quick convergence since the focus is being laid on the fittest members.
- **Elitism** mostly used to prevent losing the fittest member of the population. The fittest member is "marked" and always kept in the population.
- **Round-robin tournament** works by holding pairwise tournament competitions in round-robin format, where each individual is evaluated against q others randomly chosen from the merged parent and offspring populations. For each comparison, a "win" is assigned if the individual is better than its opponent. And after finishing all tournaments, the m individuals with the greatest number of wins are selected [7].
- (m + l) **selection**: Set of offspring and parents are merged and ranked based on fitness, survivors among parents and offspring.
- (m, l) **selection**: Children are created from the population off parents, just offspring survive while parents are discarded.

2.8 Network performance

For network performance measurement there are different tools available, some of the most popular ones are Iperf and netperf. It refers to the measurement of service quality of a network, and there are many ways to measure it as every network is different in nature and design. A few of the most important measures are as following:

- **Throughput** is the number of how many units of information a system can handle at a given amount of time.
- **Latency** tells us how fast the response time, it is the delay between sender and receiver, how fast the signal travels from one end to another.

- **Bandwidth** determines that maximum possible throughput that is achievable, can not send more data than the specified bandwith size.
- **Jitter** variation in packet delays, an undesired deviation.
- **Error rate** number of corrupted bit errors measured from the total bits which were sent.
- **Quality of Service** QoS measure accounts for importance of a specific metric to one type of application.

2.8.1 Tools

iPerf

iPerf is a tool to measure the maximum achievable bandwidth on a network. There are various parameters which can be customized, buffers and protocols such like TCP and UDP. It measures the bandwidth, loss, transfer size and other parameters. iPerf has client and client functionality, and can make data streams in order to measure the throughput, in one or two directions.[19].

netperf

Netperf is also a software application that measures network bandwidth testing between two hosts. It has a variety of tests to measure unidirectional data transfer and request / response performance

libvirt

Libvirt is an open-source API, daemon and management tool and serves as a toolkit to manage virtualization platforms. Supports hypervisors such as KVM, QEMU,VMWare etc.

Virsh[30] - A very popular command line interface for virtual machine managing is virsh. It can be used to create, pause and shutdown domains as well as list the existing domains. Virsh operations are often dependant on an already running libvirtd service.

2.9 Related work

Genetic Algorithms in Search, Optimization & Machine Learning

A book [31] presenting the simple procedures and applications of genetic algorithms. The author of this book have tried to bring together the computer techniques, mathematical tools, and research results that might enable to use GA to problems in a given field. There are chapters dedicated to search and optimization problems and also those dealing with machine learning. Some other worthwhile mentions which give a good introduction to GA [32], [33].

Introduction to Evolutionary Computing

Offers a thorough introduction to evolutionary computing (EC), descriptions of popular evolutionary algorithm variants, discussion of methodological issues and particular evolutionary computing techniques. Also discusses the future of evolutionary robotics in the future. Focuses on using instead of just studying EC, and wishes for us to apply it to a given problem [7].

Genetic Algorithm For Tightening Security

A thesis with similar problems to ours when it comes to having an x amount of parameters, merging them into different sets and finding the most fit solution for the given problem. The objective in [1], is to improve the security solutions by applying the genetic algorithm to the configuration parameters of Apache2.0.

Tuning 10Gb network cards on Linux

While the growth of Ethernet has improves and surpassed the growth of microprocessor performance in mainstream servers and computers, the focus of those machines are places in computing rather than input and output. Each generation of network cads has different features, and if not fully configured the network performance might lag behind. But for linux, where the operating system runs on a various type of machines, the default configurations are not tunes to for 10Gbit/s network cards, or 1Gbit/s in our case. This paper [10] describes the basic settings that can be changed in a linux environment in order to maximize the throughput speed.

Optimizing TCP Receive Performance

The performance of receive side TCP processing has usually been overruled by "per-byte" operations, like check-summing and copying. But as the architecture in modern processors have changed, "per-packet" operations are becoming the main source of overhead [34]. Two optimization techniques are represented to improve the receive side TCP performance. A similar benchmark for testing the TCP streaming receive throughput to netperf is used. Results of another study [35] shows that TCP is not the source of overhead often observed in packet processing, and it could support a lot higher speeds if correctly implemented

A Reinforcement Learning Approach to Online Web Systems Auto-configuration

Presents learning through reinforcement learning (RL) to improve different configuration parameters. It uses RL approach for autonomic configuration and reconfiguration of multi-tier web systems. It is able to adapt performance parameter settings not only to the change of workload, but also to the

change of virtual machine configurations. The approach is evaluated using TPC-W benchmark on a three-tier web-site hosted on a Xen-based virtual machine environment [15]. Experiment results demonstrate that the approach can auto-configure the web system dynamically in response to the change in both workload and VM resource. Mostly focusing on Apache parameters, but the auto-configuration part is relevant to what we are trying to achieve.

Chapter 3

Approach

This chapter explains the approach, walk-through and how the different problem statements were solved.

- How to use genetic algorithm for optimizing the different network configuration parameters
- How to identify relevant network parameters
- How to make an adapted approach which can change these parameters dynamically
- How to make these network parameters change depending on the payload which is received
- In what extent are the reconfigured network parameters better than the default ones

3.1 Planning the project

3.1.1 Configuration parameter selection

Will be using Netperf for measuring the speed. Netperf is a software application that can be used to measure different aspects of network performance. It supports Unix domain sockets, but mainly focuses on bulk data transfer request / response using TCP, UDP and Berkeley Sockets interface [20]. It provides numerous predefined tests.

And just like in a cluster-based web service performance, the performance improvement can not easily be achieved by tuning individual components [13], there is no single universal configuration that is good for all workloads. Therefore we are using a genetic algorithm to find the optimal configuration setup for a given payload which is going through the server.

Found very many relevant parameters in [11], [10], some of them being:

- **Jumbo Frames -** Jumbo frames are Ethernet frames with over 1500 bytes per payload, which is the standard limit [21]. Per definition Jumbo Frames can carry up to 9000 bytes. But while working with Gigabit Ethernet switches the network interface cards can support frames bigger than the default, hence why it could be increased. Frame size is directly bound to the interface Maximum Transfer Unit (MTU).

- **Multi streams -** Throughput which goes throughout the network depends on the type of traffic that is flowing in the wire. The number of streams is important, also known as a socket, opened during a time slice. It involves creating and opening multiple sockets and parallelizing the data transfer among those sockets[11]. Netperf tool has a mode where one can use multi streams with 8 active streams.

- **Transmission queue size -** It is a buffer which holds packets that are scheduled to be sent to the card. If one wants to avoid the packet descriptors being lost the size of the buffer should be tuned. The default size of 1000 packets can be too little.

- **TX Checksum -** Is a checksum offload option that asks the card to compute the segment checksum before it is sent. When this is enabled, the kernel will fill a random value in the checksum field for the TCP header and depend on the network adapter to fill it correctly.

- **TCP Segmentation Offload -** Is an element which is used to reuce CPU over while running TCP/IP. TCP Segmentation Offload (TSO) is a trait to break down large groups of data sent through network into smaller segments that pass through different network elements between source and the destination. The network interface controller segments the data and later adds the TCP/IP and data link layers to each segment.

- **Large Receive Offload -** LRO combines multiple Ethernet frames into one receiver and contributing to decreased CPU utilization when system is receiving numerous small packets. It is a technique which increases inbound throughput of high-bandwidth network connections. There are two types of LRO, one that is usually turned on by default and other which is specific for a given device driver. The last one yields much better results as it accumulates the frames into the card.

- **Generic Segmentation Offload (GSO) -** It has been observed that a lot of savings in TSO come from traversing the network stack once rather than multiple times[22]. GSO is like TSO, only effective if the MTU is around the default value of 1500.

- **TCP Window Scaling -** Is an option that can increase the size of the receive window allowed in TCP beyond its default value of 65535 bytes. The throughput is limited by two windows, the receive and

congestion window. Receive window tries not to go past the limit of the receiver to process data, while congestion window tries not to breach the limit of the network (congestion control).

- **TCP Timestamp -** can provide a more clear round trip time measurement, but it also adds an overhead to the throughput and CPU usage. This option should be disabled if one wants to increase the speed.
- **Memory -** *net.ipv4.tcp_mem = 287121 382828 57424* defines how the TCP stack should behave considering memory usage. First value telling kernel the low threshold, seconds defines the starting point at which kernel should start pressuring memory usage down and final sets the max amount of memory pages.
- **Read and Write Memory -** there are two parameters that control the write and read memory, *net.ipv4.tcp_rmem* and *net.ipv4.tcp_wmem*. Read memory takes care of the size of receive buffer used by TCP sockets, while write memory adjusts the amount of memory reserved for send buffers. Each having three values *net.ipv4.tcp_rmem = 4096 87380 16777216*, first value tells kernel the minimum receive buffer for a TCP connection and is allocated to a TCP socket, second is the default receive buffer and the third is the maximum receive buffer. While for the *net.ipv4.tcp_wmem* it is about how much TCP sendbuffer memory space each TCP socket has that can be used.

3.1.2 Topology

The setup in order to perform experiments is shown in 3.1, which consists of two physical servers lent from the Oslo-Met university with Ubuntu 16.04 installed on them. Servers are connected through a switch to the Oslo-Met internet, and through another switch through their 1 gigabit Ethernet ports to make it a bit more realistic by adding a hop in-between.

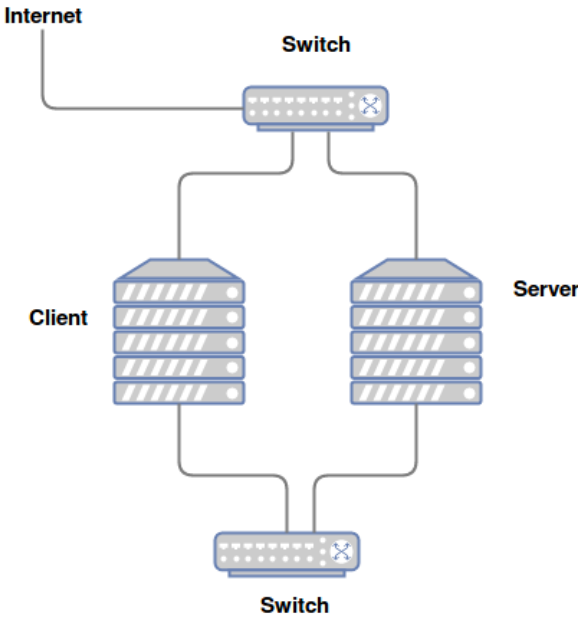

Figure 3.1: Topology of the setup

The specifications for each of those machines are as following:

OS	Ubuntu 16.04 xenial
CPU	2x Intel Xeon CPU E5530 @ 2.394GHz
NICs	Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet x4 Intel Corporation 82576 Gigabit Network Connection x4
RAM	290MiB / 24098MiB
Disk space	151 GiB

Table 3.1: Machine specifications

3.1.3 Initial tests

Cleaning script

In order for everything to run as smoothly as possible, a cleaning script was run. Even on freshly made instances there are always some services running in the background that could influence the throughput speed in some way, hence we want to kill all known services that are known to cause even minor disturbance. This way we will be in the clear that they do not contribute to anomalies in the results.

```
1  #! /bin/bash
2
3  services="iscsid cron atd rsyslog acpid libvirt bin libvirt
   guests apparmor ebtables friendly recovery resolvconf ntp atop
```

```
    pmlogger pmie pmcd pmproxy open iscsi openipmi lxcfs bind9
   accounts daemon  metricbeat redis server collectl"

   for service in $services; do
   service $service stop
   done

   apt purge snapd ubuntu core launcher squashfs tools
   systemctl mask accounts daemon.service
   apt remove policykit 1 y   purge

```

Listing 3.1: Cleaning script - stopServices.sh

Default parameter values

List of the different parameters and their default values that were pre-defined on the machines:

Type	Parameters	Default
TCP - net.ipv4	tcp_mem tcp_rmem tcp_wmem tcp_moderate_rcvbuf tcp_no_metrics_save tcp_timestamps tcp_window_scaling tcp_sack	287121 382828 57424 4096 87380 629145 4096 16384 419430 1 0 1 1 1
net.core	wmem_max rmem_max rmem_default wmem_default netdev_max_backlog	212992 212992 212992 212992 1000
UDP - net.ipv4	udp_mem udp_rmem_min udp_wmem_max	574242 765657 1148484 4096 4096
MTU	mtu	1500

Table 3.2: Default parameter values on host and server machine

Performed an iperf test from client to server, in order to check the throughput speed before changing any values:

`Throughput:`

```
iperf -c 10.0.0.1
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
```

```
[  3] local 10.0.0.2 port 55880 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.10 GBytes   941 Mbits/sec
```

```
Latency:

--- 10.0.0.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 18997ms
rtt min/avg/max/mdev = 0.175/0.217/0.257/0.022 ms
```

As seen above the throughput speed for the default values is already almost at its limit, 941 Mb/s out of 1 G is almost as good as it can be taken in account there will always be some overhead making it impossible to reach full speed.

New parameter values

Type	Parameters	New
TCP - net.ipv4	tcp_mem	16777216 16777216 16777216
	tcp_rmem	4096 87380 16777216
	tcp_wmem	4096 65536 16777216
	tcp_moderate_rcvbuf	1
	tcp_no_metrics_save	1
	tcp_timestamps	0
	tcp_window_scaling	1
	tcp_sack	0
net.core	wmem_max	16777216
	rmem_max	16777216
	rmem_default	412992
	wmem_default	412992
	netdev_max_backlog	2500
UDP - net.ipv4	udp_mem	574242 765657 1148484
	udp_rmem_min	4096
	udp_wmem_max	4096
MTU	mtu	1500

Table 3.3: New parameter values

Another iperf test was performed in order to check the throughput speed after changing the values:

```
Throughput:
iperf -c 10.0.0.1
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
```

```
------------------------------------------------------------
[  3] local 10.0.0.2 port 56046 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.10 GBytes   949 Mbits/sec

Latency:

--- 10.0.0.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 18999ms
rtt min/avg/max/mdev = 0.186/0.206/0.268/0.023 ms
```

A slight increase of 8 Mbits/sec in bandwidth speed has been noticed, as suspected not a big change since the starting point is already near top speed.

```
MTU=2700 on client and server
iperf -c 10.0.0.1
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 55878 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.13 GBytes   971 Mbits/sec
```

During a test to check how important MTU was when increased on both client and server, the speed was increased to very likely its maximum. Unfortunately there will not be opportunities to change settings like these on the client machine, hence all future tests will be done by just changing parameters on the server machine.

3.1.4 The algorithm

The genetic algorithm implementation used is a combination of the three operators; selection, crossover and mutation. In this case each chromosome is a representation of a parameter combination, whereas a gene is a single parameter among many. There are 14 different parameters available in the list where the genetic algorithm will be extracting them from. These values were found from various articles and online posts such as [10], [11]. For every run of the algorithm each parameter from the list can be extracted, although it is the algorithm that decides if all the parameters will be used or just a selected few based on the fitness.

```
1 sysctl  w net.ipv4.tcp_mem=;'287121 382828 574242';'16777216
    16777216 16777216'
2 sysctl  w net.ipv4.tcp_rmem=;'4096 87380 6291456';'8192 873800
    16777216'
```

```
sysctl  w net.ipv4.tcp_wmem=;'4096 16384 4194304';'8192 873800
    16777216'
sysctl  w net.ipv4.tcp_moderate_rcvbuf=;0;1
sysctl  w net.ipv4.tcp_no_metrics_save=;0;1
sysctl  w net.ipv4.tcp_timestamps=;0;1
sysctl  w net.ipv4.tcp_window_scaling=;0;1
sysctl  w net.ipv4.tcp_sack=;0;1
sysctl  w net.core.wmem_max=;212992;16777216
sysctl  w net.core.rmem_max=;212992;16777216
sysctl  w net.core.rmem_default=;212992;412992
sysctl  w net.core.wmem_default=;212992;412992
sysctl  w net.core.netdev_max_backlog=;1000;5000
ifconfig eno2 mtu ;1500;2700
```

Listing 3.2: Parameter list

For the sake of simpleness, there are minimum and maximum values listed for every parameter, and the genetic algorithm will choose a random value in between the minimum and maximum, hence if the population is either big or small there will still be diversity. The minimum value is the value/s placed in the first ampersands after the semicolon, ex. '287121 382828 574242', while the maximum is right after a following semicolon: '16777216 16777216 16777216'. The way this is solved in python is:

```
    if len(parmin) == 3:
            for i in range(len(parmin)):
                nyrange = random.randint(int(parmin[i]), int(
    parmaks[i]))
                nyverdi += "%s " %str(nyrange)
            nyverdi += "'"

    else:
            mini = int(para[p][1])
            maks = int(para[p][2])
            nyverdi = str(random.randint(mini,maks))
```

Listing 3.3: Parameter value selection

If there are multivalued parameters such as TCP read or write memory, which consist of 3 values, a function-call on random is performed in python that will choose a value between the first index of minimum and first index of maximum value, 287121-16777216, then the second and the third. But if there is a single valued parameter like netdev_max_backlog, it makes it even easier and the line is just split on semicolons, 1000-5000.

Now as there are different parameters and parameter-values to choose from, the initialization of parents can take place. A variable called ant decides how many parameters out of the total should be given to each parameter-combination, parent. Each parameter is a string that is a Linux-specific call, the parameter values are split from the string and converted to integers in order to perform a function call in python on those numbers.

The first chromosomes are initiated randomly by using random.sample, it

picks x unique random elements, a sample, from a sequence of parameters. Henceforth those random elements are saved to a parent stored in a given population.

```
def gen_paracombo(ant, combo, pop):
    perm = [] #permutation

    for i in range(0,pop):
        randperm = random.sample(range(0,(len(para))), ant)
        perm.append(randperm)

    return perm
```

Listing 3.4: Initialization of new parents

When all the parents are finished initializing and placed in the population pool, the next phase begins, and that is to measure the fitness for every member of the population. A call on the function speedtest as shown below, results first in applying the old parameters to the server, this way we can reset the previous configurations. Next, every parameter in each parent by order is applied to the server while a test with netperf is performed. The netperf test is set for one second as it would take immensely long time to use the default time of ten seconds for every parent. The throughput speed is extracted from the output and saved to a list. Now every parent has a throughput speed assigned to itself, which is their fitness value as well.

```
def speedtest(params, cnt):
    paramset = []
    combspeed = []

    process = subprocess.Popen(['sh','old.sh'], stdout=subprocess.
    PIPE, stderr=subprocess.PIPE)
    process.wait()

    open('paratest.sh', 'w').close()
    with open('paratest.sh', 'a') as file:
        for p in params:
            file.write('%s\n' % (p))

    process2 = subprocess.Popen(['sh','paratest.sh'], stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
    process2.wait() # Wait for process to complete.

    for p in params:
        print (p)
        paramset.append(p)

    parcomb.append(paramset)

    # ssh barcli@10.0.0.2  p 2222  i ~/.ssh/master "netperf  H
    10.0.0.1  l 1" | tail  1 | awk '{print $NF}'
    thro = subprocess.Popen(['sh','throughput.sh'], stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
```

```
    thro.wait()

    for line in thro.stdout.readlines():
        throughput = line

    throughput = throughput.decode('UTF 8')

    throughspeed.append(float(throughput.strip("\n")))
```

Listing 3.5: Fitness test for every member of the population

And as shown in figure 2.8, next phase of the genetic life cycle is selection. The user can select the percentage of population that should be disposed of for every generation by setting the prob value to f.ex. 0.1 which will delete 10 percent of the worst fit individuals in every generation. Following the selection, comes crossover.

```
def crossover(ant, bestfit, best, kortest, crossprob, mutprob):
    parent1 = bestfit[  1][:]
    parent2 = bestfit[  2][:]

    size = min(len(parent1), len(parent2))
        for i in range(size):
            if random.uniform(0, 1) < crossprob:
                parent1[i], parent2[i] = parent2[i], parent1[i]
```

Listing 3.6: Crossover

The type of crossover used is uniform crossover with a mixing ratio of 0,5, for starters, adjustable when needed. Which means that the offspring will get around 50 percent of the genetic material from parent 1 and parent 2. In uniform crossover the parent is contributing on the gene level instead of the segment level.

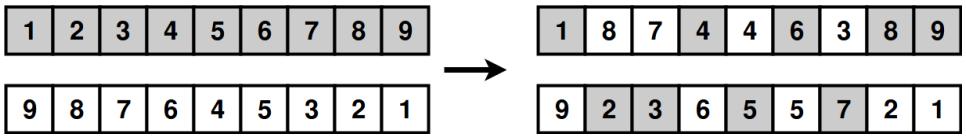

Figure 3.2: Uniform crossover

As it can be seen there is not always possible to cross the exact amount of genes, that is because there is an x% probability for every single gene to be swapped, in the long run it will add up, but not for every single chromosome.

After the crossover has been performed it is necessary to run a mutation operation in order to prevail genetic diversity and avoid being stuck in a local maximum. A local maximum is the largest value of a function within a given range, hence if it so happens that all the parents are gathered around

the local maximum the goal is to try to reach a breakthrough, and it is only possible through a mutation. Else wise the top solution will just be the best possible solution locally, but not globally. The global maximum is what one strives to find, which is the overall best solution and the highest hill top in 3.3.

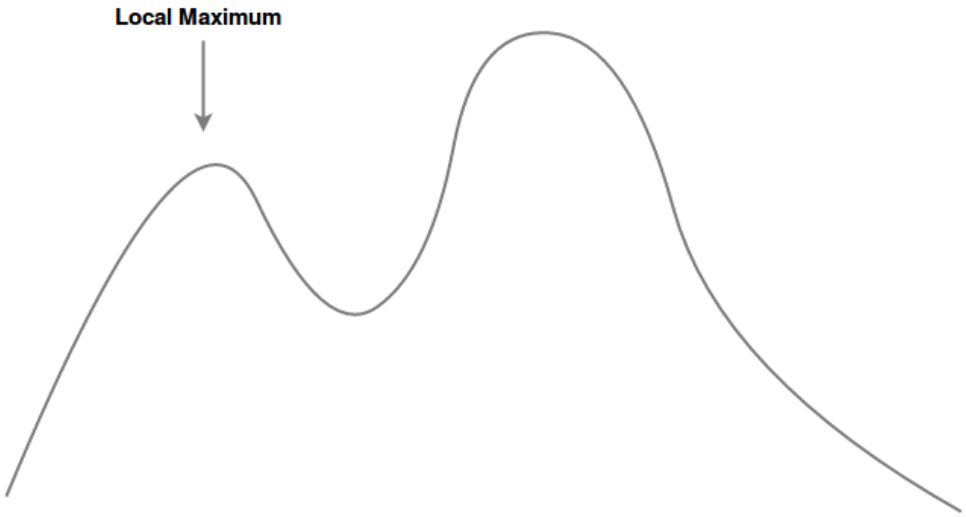

Figure 3.3: Local maximum

The mutation probability is set to be that of variable mutprob, the probability for genetic algorithms should be low because one wants to focus on exploitation instead of exploration. In genetic algorithms, mutation operator focuses on exploration while crossover is used to lead a population to convergence towards a local maximum, which also can be the global maximum, this is called exploitation. High mutation rate does increase the probability for searching different areas, but it also prevents population to converge to an optimum solution. While on the other side a too small mutation rate will result in premature convergence.

```
    p = randint(0, ant  1)

    if (randint(0,100) < mutprob):
        print ("Mutation on parent1")
        parent1[p] = mutationparam(ant)
    if (randint(0,100) < mutprob):
        print ("Mutation on parent2")
        parent2[p] = mutationparam(ant)
```

Listing 3.7: Mutation

3.2 Approach 1

Figure 3.4 shows a simplified flowchart for the entire algorithm. Starts with initialization of parents, following a fitness evaluation for every individual

(throughput testing) in the whole population. Those fitness values along with the according individuals are passed along to go through the selection, crossover and mutation cycle, whereas the new-found individuals are evaluated before being added to the population. This cycle repeats several times, in a real life system this would go on forever, but in our algorithm we terminate it after x-amount of iterations in order to get the results.

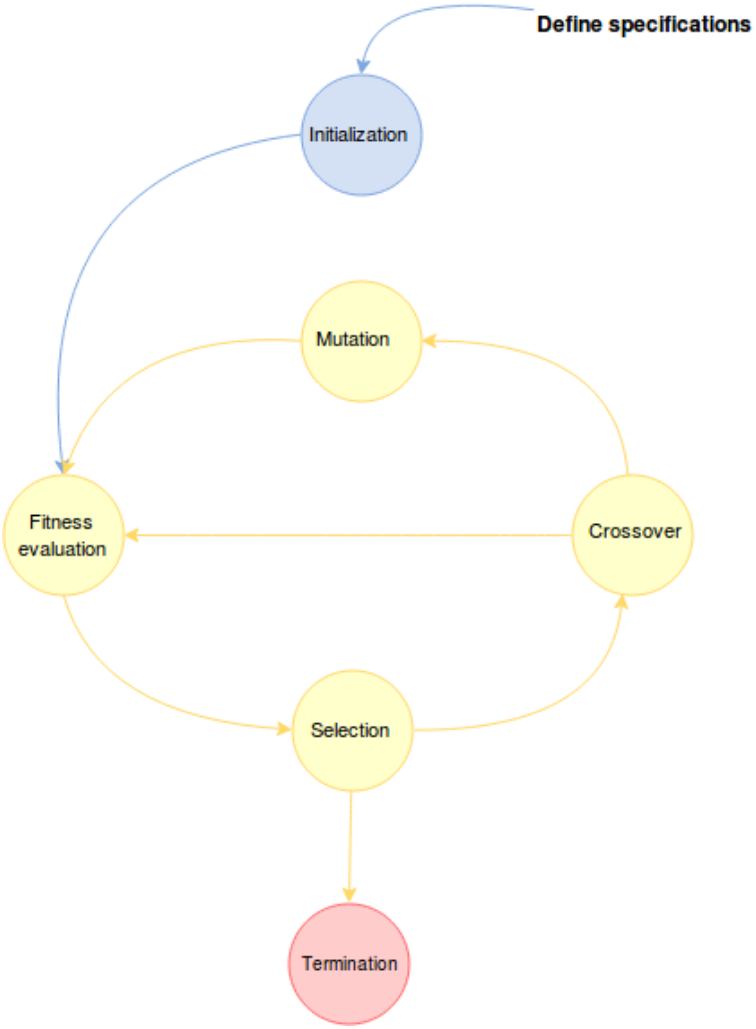

Figure 3.4: Flowchart for the algorithm

First test run of the algorithm was performed on a server with no traffic, allowing the throughput speed to reach very high values, almost attaining the maximum speed possible. It was run with a population of 50 individuals for 20 generations. This can be seen in figure 4.1, showing the average throughput speed for every population in each generation, and figure 4.2 that represents the best individuals also in every generation. Values for the best individuals are "frozen", which means that for every time a current best solution is found, it is saved as the best individual until an individual

with better fitness is discovered. The results are shown in section 4.1, registering very little to no improvement. Following tests will be performed on server with traffic.

3.3 Approach 2

A little change in the topology 3.5 was required, as opposed to topology 3.1, two additional ethernet cables were connected to the switch and client in order for every Network Interface Controller (NIC) to be able to perform its own payload generation. Along with a few more new-found parameters, to increase the range for parameter-combinations.

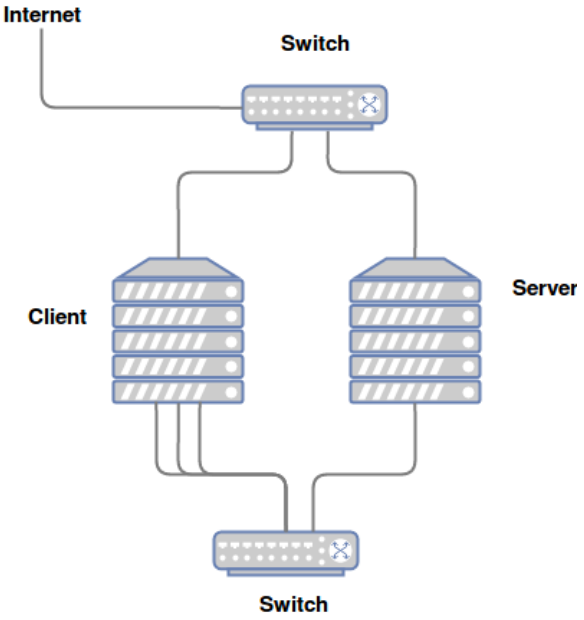

Figure 3.5: New topology

The numerous new parameters 3.3 which were introduced in this second approach.

Type	Parameters	Min	Max
TCP - net.ipv4	tcp_mem	287121 382828 574242	16777216 16777216 16777216
	tcp_rmem	4096 87380 6291456	8192 873800 16777216
	tcp_wmem	4096 16384 4194304	8192 873800 16777216
	tcp_moderate_rcvbuf	0	1
	tcp_no_metrics_save	0	1
	tcp_timestamps	0	1
	tcp_window_scaling	0	1
	tcp_sack	0	1
	tcp_tw_reuse	0	1
	tcp_keepalive_probes	5	9
	tcp_keepalive_intvl	15	75
	tcp_fin_timeout	15	60
net.core	wmem_max	212992	16777216
	rmem_max	212992	16777216
	rmem_default	212992	412992
	wmem_default	212992	412992
	netdev_max_backlog	1000	5000
	bpf_jit_enable	0	2
	dev_weight	64	600
	rps_sock_flow_entries	0	32768
	optmem_max	20480	25165824
	somaxconn	128	4096
	busy_read	0	50
	busy_poll	0	100
	tstamp_allow_data	0	1
MTU	mtu	1500	2700
Txqueuelen	txqueuelen	1000	5000

Table 3.4: Minimum and maximum values for all parameters

As the traffic was introduced the speed dropped significantly, giving the algorithm more room for improving the throughput speed.

```
iperf -c 10.0.0.1 -B 10.0.0.2 -t 20000

iperf -c 10.0.0.1 -B 10.0.0.3 -t 20000

iperf -c 10.0.0.1 -B 10.0.0.4 -t 20000
```

Listing 3.8: Traffic generated with 3 NICs

Calls used in order to generate the traffic, each going on for 20000 seconds, having enough time for the algorithm to run once. While this was run in the background, the tests could be performed 4.2.

3.4 Approach 3

The question now was, do we use the right factors (mutation-, selection-, crossover- probabilities), or could and should they be changed? Following this problem we performed tests while changing the various probabilities to see if we can get any improvement. But before that, the duration was also changed, up until now all the parameter combinations were tested throughout just one second. It was mostly for the sake of time consumption, running the algorithm once takes around two hours with the population of 80 and 40 generations. Running it ten times to get better results takes a lot longer, around a day, so by increasing the duration of how long netperf should test every individual, it increases tenfold. A test was performed, comparing the 1 second test to 10 second with the exact same factors.

Following the duration-experiment, we ran single tests, increasing the mutation and selection probabilities, and decreasing the crossover probability to see what differences we would get by the variation of one single factor at a time.

3.5 Approach 4

In order to make a more standardized test, we now wanted to test the algorithm and its behaviour on VMs. For that to work the topology was changed, libvirt and KVM were installed on both client and server in order to create virtual instances on those machines. The reason for why we chose to run the VMs from the physical machines and not from a cloud based environment is because we want to test the performance both while we change the parameters on the VMs as well as the host machine, individual tests. It is a way to establish if changes on just the VMs actually have as much effect as the host machine they are operating on, and also if just changing parameters on the host machine will affect the performance on the VMs.

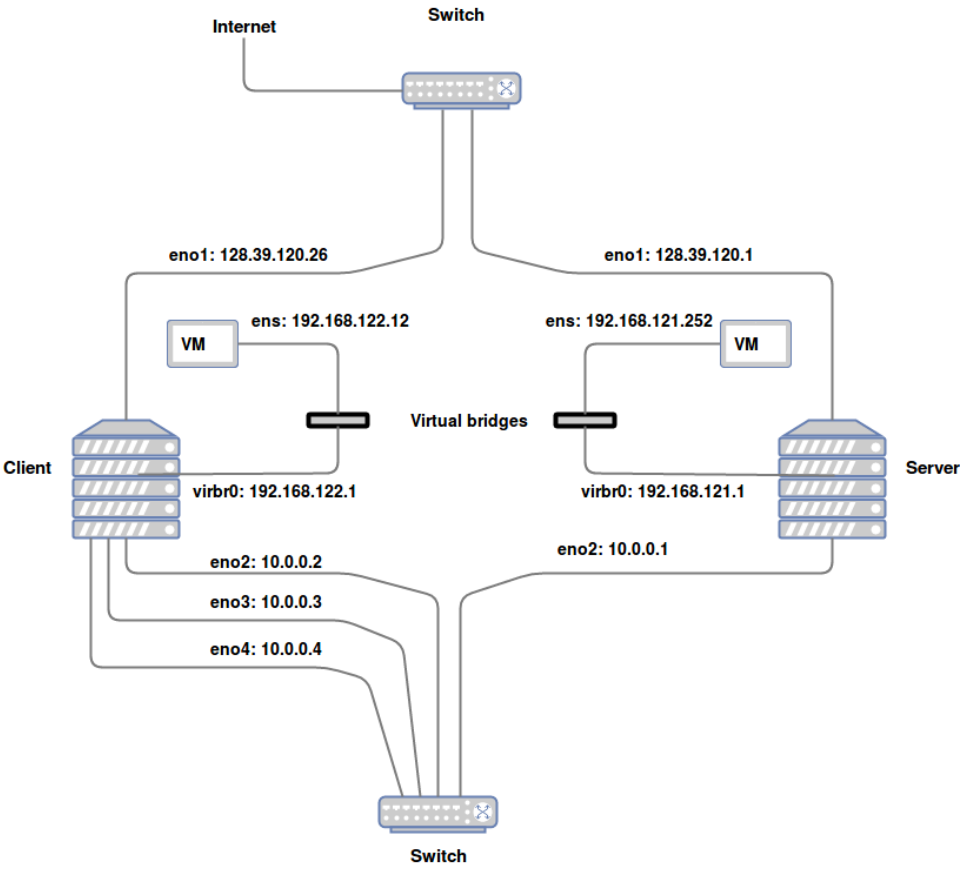

Figure 3.6: Topology with VMs

Upon the setup of libvirt, virtual bridges were added on the host machines for a means to reach the VMs. Virbr0 interface is used for NAT (network address translation) and is used by the VMs to connect to the outside network.

Performed various results in 4.4, with different scenarios, such as:

- Configurations changed directly inside the VM
- Configurations changed on the host machine (server)

This was to assure which of these scenarios was most reliable and if more experiments were to be performed, where would be the best machine, virtual or physical, to change the different configurations.

Chapter 4

Results

The following chapter will be focusing on giving an overall idea of the tests performed with the thought of a clear and simple image of all the experiments performed. From the starting phase and until we achieve the results we are aiming for, or at least close in on them.

4.1 Tests with no traffic on the server

First set of tests (Approach1 3.2) performed on the server with no traffic, meant to give an overview of the actual bandwidth and how much throughput we can achieve.

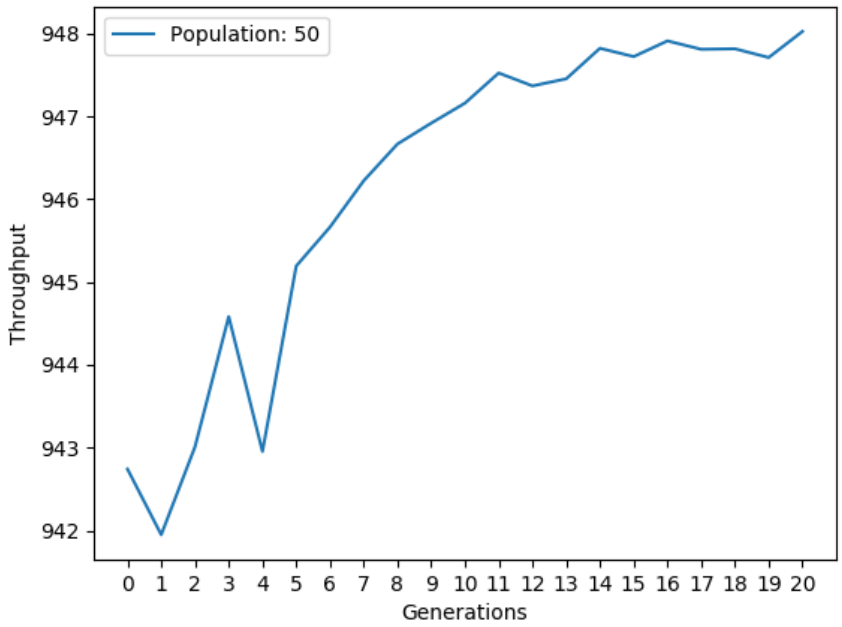

Figure 4.1: The average throughput speed for every generation

It can be observed that the graph is pretty bulky and uneven, jumping

up and down, but the more iterations, the more the throughput is improving. It starts at around 943, but at around 16th generation the throughput speed is flatting out at 948 throughput, not reaching significantly higher speed. The population pool and generations are a bit small, but that was one of the first tests performed with a still not improved algorithm.

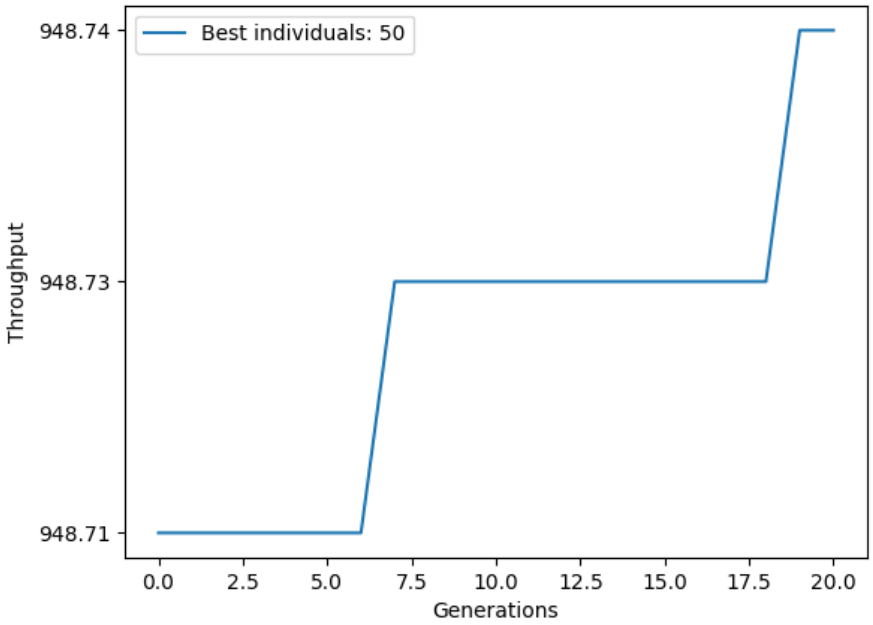

Figure 4.2: The best individual found for every generation

As for the graph listing the best individuals, there is not really much improvement, going from 948.71 to 948.74 throughout 20 generations. Again it is the high initial speed that is to blame, not much room for improvement.

Another thing to consider is that the measurements were taken for the whole population pool, even after removing 10 percent of the worst population and doing crossover. Then adding the new-found offspring to the population which could have even worse fitness than any other individuals, even those previously removed. This is something that was noticed after the experiments were performed, and thus the new-found offsprings contribute to uneven variation in the graph of averages. For the next set of experiments this mistake was corrected, and the graph shows a more correct average for every generation.

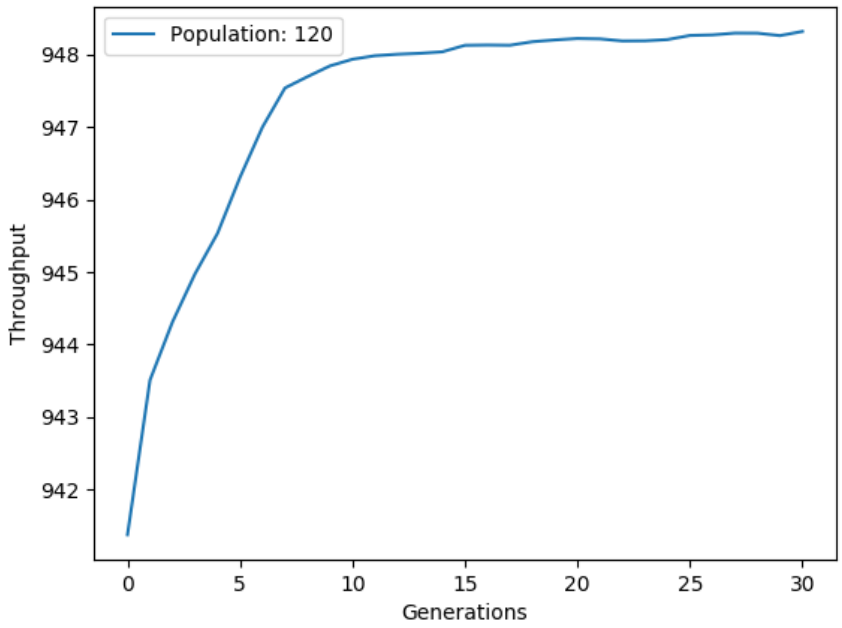

Figure 4.3: Average: 2nd run

A new test with corrected code and bigger population pool along with more generations. The graph is a lot more smoother now with no up and downs, it can be seen that the average fitness of the population is just going upwards.

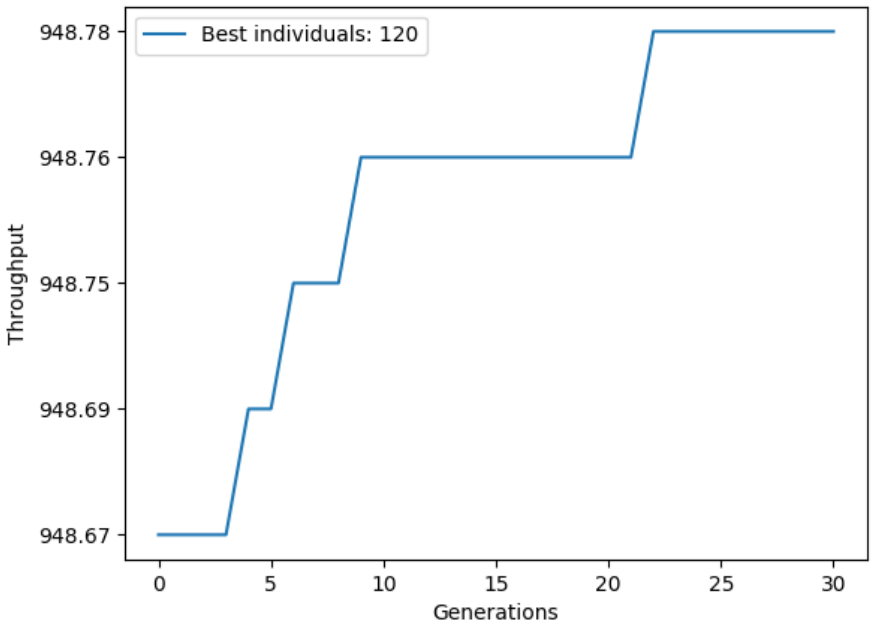

Figure 4.4: Best individuals: 2nd run

The results for best individuals are still not improving greatly, even with a greater population pool.

4.2 Tests with traffic

Second round of tests (Approach2 3.3) performed on the server with introduced traffic, gives us an idea of how the algorithm would behave in a more real life scenario.

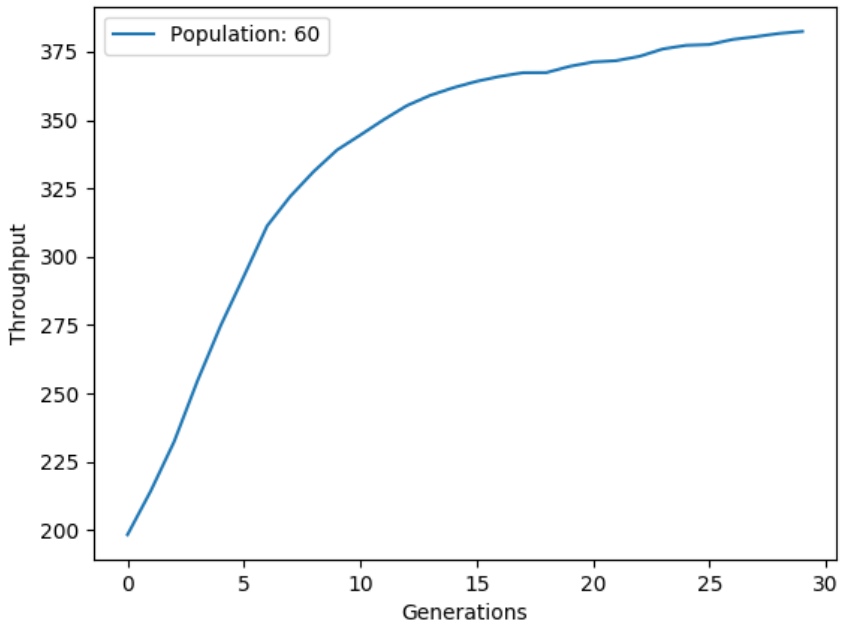

Figure 4.5: Average - Traffic on the server

Now as there is traffic going through the server, we can see that the initial starting point is not near the maximal throughput speed, hence there is a lot greater room for improvement. And as the graph presents, the average throughput speed improves almost twofold. The type of traffic which was generated was done through the client using the iperf tool. The client host has four NICs where each of the iperf tests were bound to one of three NICs (one is being used for connection to the internet) sending data through the switch to the server. There is no loadbalancer as of this moment that could handle the load-distribution, which is planned in the future when VMs get introduced to the topology. The data is iperf generated and is prone to varying speed at different times, even seconds apart, since all the NICs are fighting for the right to send.

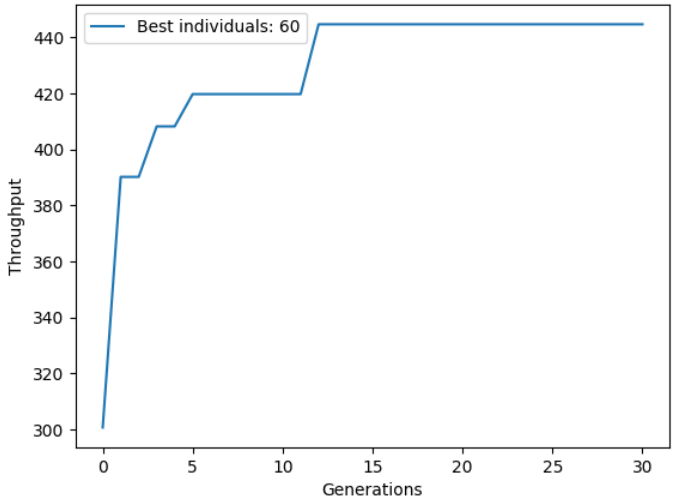

Figure 4.6: Best individuals - Traffic on the server

Even the results for best individual for every generation is giving results with pretty good improvement, not like before by just a few mbit/s. Cause of this is of course a low starting point, but factors that could contribute to this might also be iperf generated traffic. Something which is still unknown, the results may vary or could be very different if it was real life traffic going through.

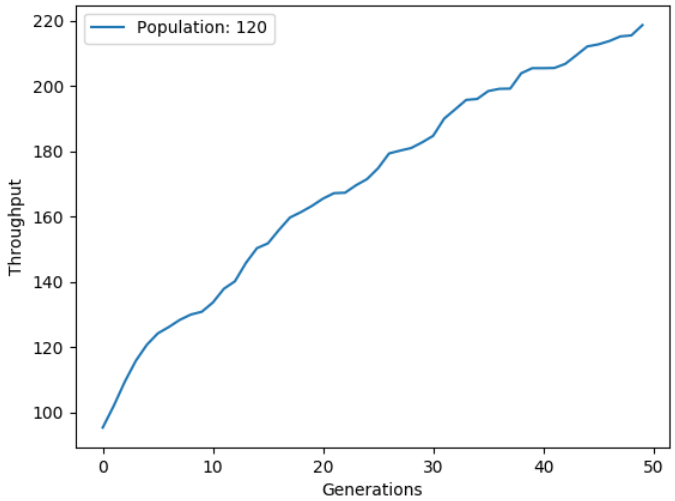

Figure 4.7: Average - Increased population pool and more generations

The algorithm was run with a bigger population pool and more gener-

ations, again it can be seen that even though the iperf tests to generate payload were run the same way as earlier, the speed dropped considerably, the speed relies on how the iperf traffic fights its way to the source competing with all the other data. Different starting point of iperf payload generation on different NICs can give different result, but although the traffic is different from another session, it is still pretty consistent throughout the entire already ongoing session.

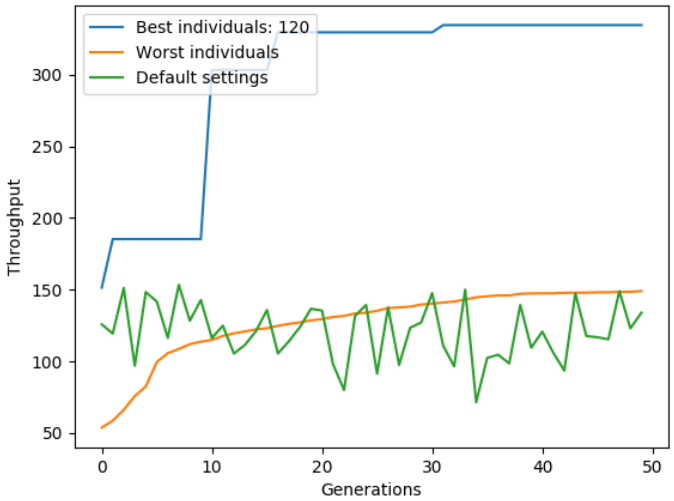

Figure 4.8: Best, worst and default individuals

And since there was no distinction or correlation to how well the new parameter settings actually performed relative to the default settings. This part was added to the algorithm, now showing the best and worst individuals for every generation, in addition testing the default parameter settings for each iteration as well. It is now possible to tell how well the genetic algorithm chooses the new parameters by comparing them to the default ones. Can be observed that the default ones are staying around 100 and 150, while even the first generation of new parameters is pretty much at 150 throughput speed. Although it is still one test, there should be multiple runs of the algorithm to sum the average of the results over multiple runs in order to have a better idea of the total outcome. At least just by looking at the default settings graph, it is very stuttering, if the algorithm was performed ten times the line would give us a better understanding of the overall throughput performance.

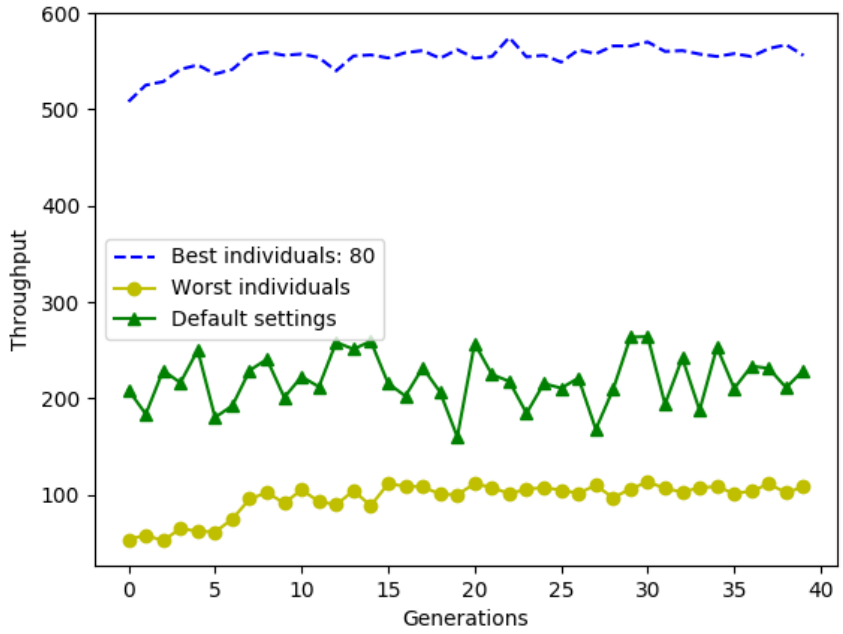

Figure 4.9: Best, worst and default individuals: 10 runs and every parameter combination retested

For all the previous tests the values were "frozen", which means that whenever a throughput speed that was higher than any previously found, it was set as the top speed, although the traffic varies and we can not be sure that this set of parameters will perform as well for other types of payload. This time every set of parameters is retested through every generation, resulting in more accurate data and fluctuations in speed. Before, the speed in the graphs never went down for the top or average individuals, now it jumps both ways, and the results are more realistic.

From here on forward we decided upon using the same variables for the different factors, as used in fig 4.9, calling this for a standard preset. This way it will be easier to compare the results to each other.

- Parameter number: 27
- Generations: 40
- Population size: 80
- Selection probability: 10%
- Crossover probability: 10%
- Mutation probability: 16%

4.3 Parameter and duration variation

Tests in this section, as explained in (Approach3 3.4), compare the different duration intervals each parameter configuration is tested with, as well as the operator probabilities to find out what gives us the best results.

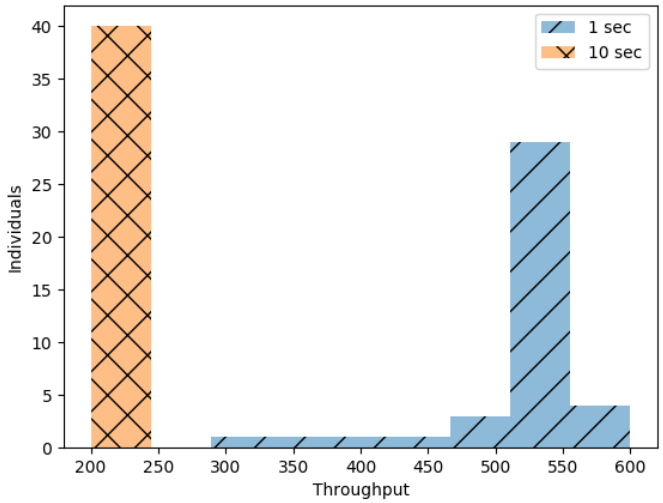

Figure 4.10: Comparison between one and ten second run

When comparing these two experiments it can be seen that the 10 second results are a lot more stable, ranging from 200-250 while 1 second results are placed between 300 and 600. The reason for this is random spikes in throughput speed at random times, if it is tested over a longer period it does get pretty even, but for a short measurement it will not. Hence the ten second test is a lot more realistic, and is something we will be using forward on for the sake of certainty.

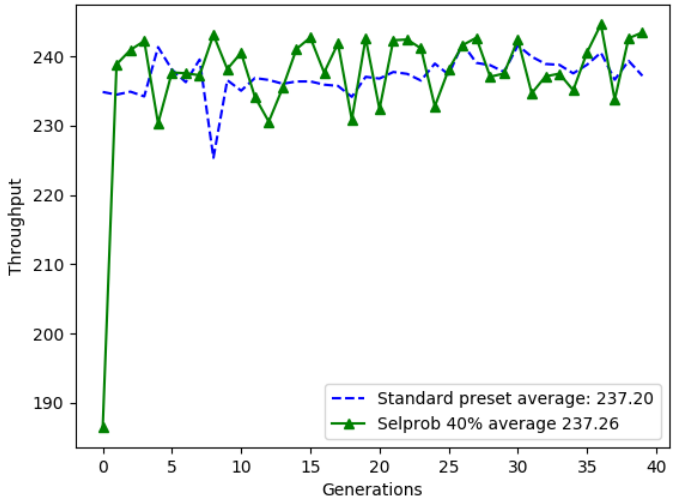

Figure 4.11: Comparison between standard preset and heightened selection probability

A comparison between the top two yielding results is also shown, the throughput speed is very close, although the experiment with increased selection probability takes a lot longer time, hence the standard preset is still favored over the other. It also seems to be more stable with less extreme hops.

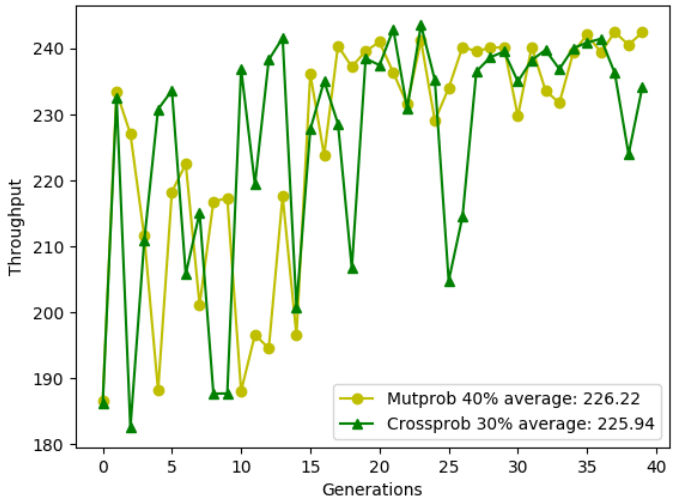

Figure 4.12: Comparison between heightened mutation probability and lowered crossover probability

The increased mutation and decreased crossover probability are a lot more uncertain. Although they are reaching higher speed on many hops,

the variation is still too much, leading to both lower averages and bigger hops. This concludes in us continuing to use the standard preset as the increase in the different factors did not yield better results, and when it did by a slight increase in throughput speed, the time performed was a lot longer. The standard preset was also the most stable set of the different factors.

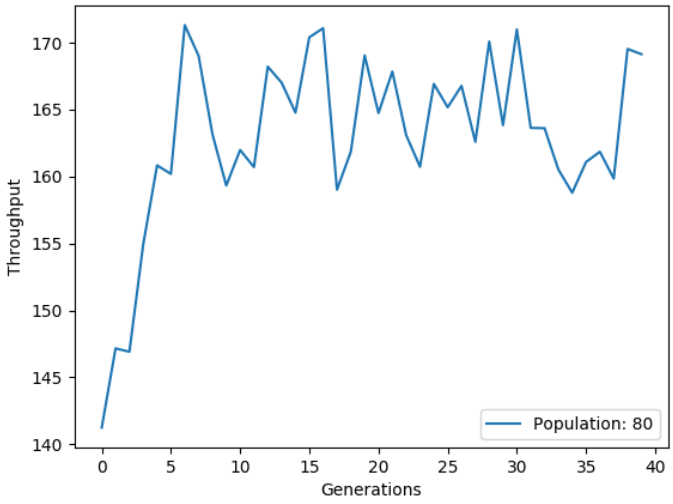

Figure 4.13: Average: Ten second tests with standard preset

Now as we have concluded that the 10 second duration run on each parameter configuration is the most stable, along with the standard preset for a run, we decided to see how well it performed during a test. In fig 4.13 it can be see that the average does increase from the first generation, but just after few runs it does not go any higher and jumps up and down. The improvement in speed are not as great as first anticipated, but this is closer of how it will be in a real life scenario, the shorter one second tests had too much uncertainty.

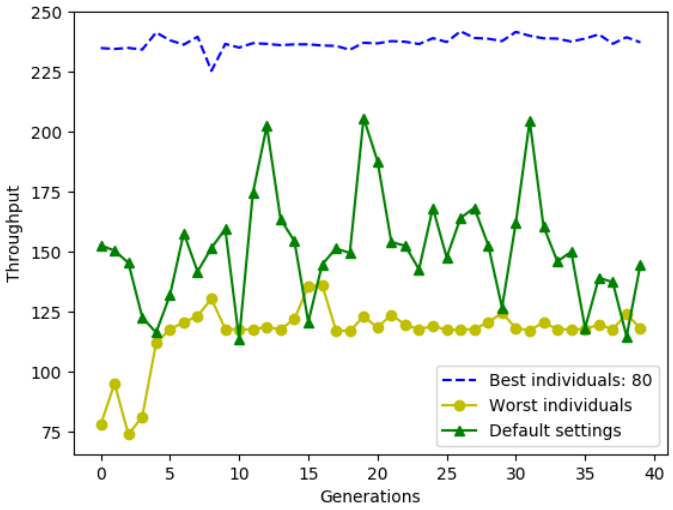

Figure 4.14: Best, worst and default individuals: Ten second tests with standard preset

As for the top individuals in fig 4.14, it is more stable and higher than the default settings, by a whole 65% in average, but the speed does not increase throughout the generations, jumps slightly up and down. This means that a good parameter combination is found pretty fast, and when it is found, there is not much improvement that can be achieved with the amount of parameters we are working with and the type of payload we are generating.

We wanted to see if there were any drawbacks with higher throughput by testing the latency through hping3 on a server with traffic, with default and fast configurations 30:

```
sudo hping3 -c 10000 -i u10000 10.0.0.1 -p 8000
    Fast parameter combination
        round-trip min/avg/max = 1.2/6.8/1005.8 ms
    Default
        round-trip min/avg/max = 1.3/7.0/1006.1 ms
```

There is not much difference between the latency, meaning that the new set of parameters do not affect the latency in a bad way.

4.4 Tests on VMs

Performed different tests on VMs with a new topology (Approach4 3.5), one in where the parameters were directly changed inside the VM residing within the server machine, another one where parameters were changed on the host machine of the VM. Virtual machines do not have physical access to the network card or hardware, making it questionable if changing parameters inside the VM instance actually gives any difference.

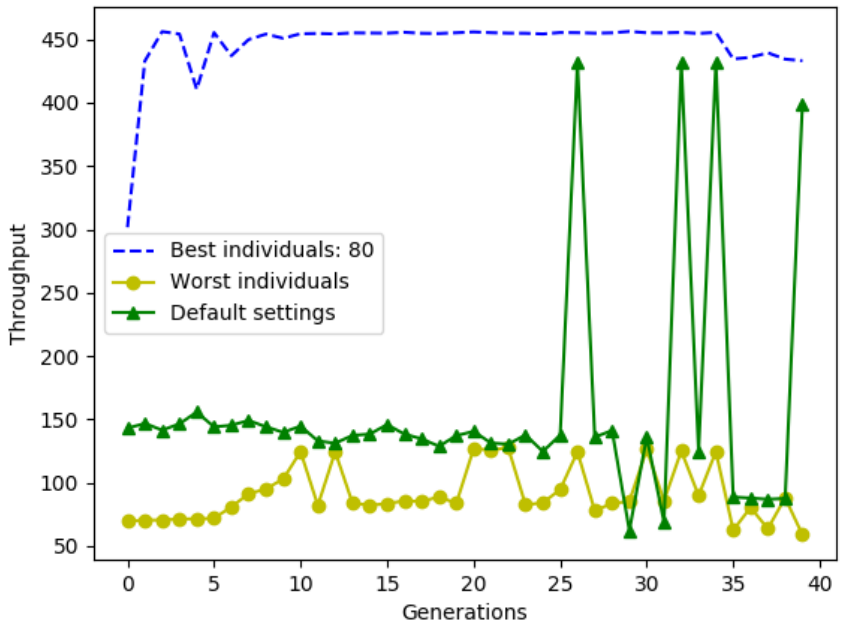

Figure 4.15: Best, worst and default individuals: Test performed on a VM

According to fig 4.15, the throughput speed still increases on a VM, even if it has restricted access to the resources. Although the results seem suspiciously high, whereas a VM should be slower and not as efficient as a physical machine, at least the one it is being run on, the results may show otherwise. This could be due to the fact that VMs are just an emulation of a computer system, and the results they are spewing out might not be as real, but simulated. Another factor could be that the load on the server at the given time was not as high as usually, resulting in higher speed. The most important results while running this kind of tests is how the speed is relative to the default speed, we are looking for the improvement not the actual speed which can vary.

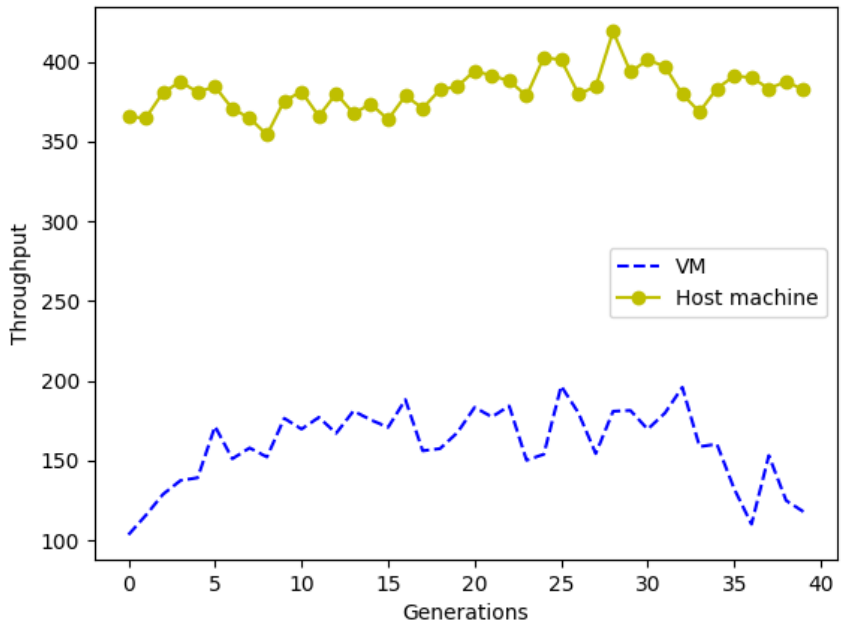

Figure 4.16: Average: Comparing the throughput based on where the different parameter configurations applied, on the VM and host machine

Fig 4.16 tells us that the average population is less fit when the configurations are just changed inside the VM, opposed to the one in the host machine. This is due to the fact that a VM has no hardware and just uses the resources of the host machine.

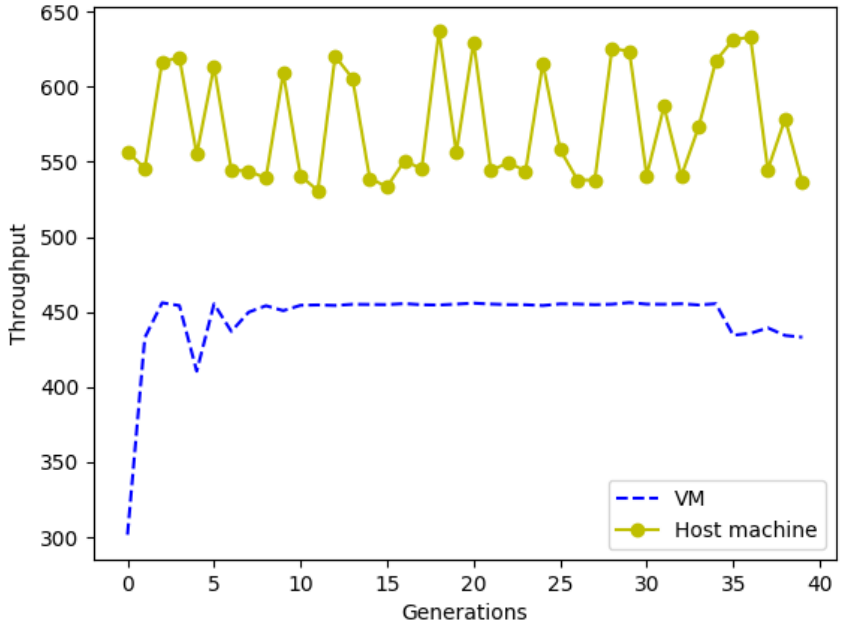

Figure 4.17: Top individuals: Comparing the throughput based on where the different parameter configurations applied, on the VM and host machine

Higher fitness also applies in fig 4.17 for the top individuals when configurations are applied on the host machine instead of the VM, yielding higher speed than that of the default parameter settings being 352 in average. While the throughput speed for default settings on the VM is 158. So even though the population in fig 4.16 does not get much better, the top individuals are still a lot faster, on both the experiments. This means that even though the VM might not have direct access to the resources on the host machine it is being run on, it can still manage to improve the throughput by changing relevant parameters inside of itself.

Chapter 5

Discussion

This chapter discusses the different implementation steps, challenges and the different error sources that could have affected the experiments.

The demand for high-speed large file transfer is increasing, but the Linux network stack is not configured for this. This problem can be improved by finding the correct parameter combinations for the received payload at a given time.

There is an infinite number of parameter combinations and some of the parameters can lead to loss in throughput speed if incorrectly adjusted. A given set of parameters needs to be correctly configured in order to yield better results than those that are already predefined in a system (the default settings).

The goal for this thesis was to write a genetic algorithm that would solve the problem statements listed below, running in the background as it changes the parameters on a system dynamically.

- How to use genetic algorithm for optimizing the different network configuration parameters
- How to identify relevant network parameters
- How to make an adapted approach which can change these parameters dynamically
- How to make these network parameters change depending on the payload which is received
- In what extent are the reconfigured network parameters better than the default ones

There were several problem statements, but the main problem was to find the correct set of parameters which would contribute to an optimized high-speed file transfer depending on the payload that is received. With so many available parameter configurations, one had to ensure the correct configurations were chosen. Finding them by hand would have been impossible, but with the help of genetic algorithms, which specialize in search and optimization problems, the task become achievable.

The idea with the genetic algorithm was for it to find the correct configuration parameters independently for what type of traffic was going through. If the throughput speed suddenly dropped by a significantly amount, the genetic algorithm would begin a search for other combinations and parameter values than those currently used. Fitness is being determined by the throughput speed, hence whenever the type of traffic changes and a set of parameters loses speed, it also loses fitness. Our algorithm solves that, discards the worst fit combinations and searches for something that works better.

Different machines will have different parameters, and it is the goal for the GA to find relevant network parameters specifically for the machine it is being run on. These parameter configurations change dynamically upon the received payload by the GA, leaving all the unnecessary configuration handling to itself.

5.1 Project evaluation

In this thesis we want to solve the problem of too slow large file transfer, by using the GA. Not optimally configured network configurations cause the speed to not be fully optimized, whereas in theory the throughput can be bigger than the default speed. The GA is used to run a number of iterations with the chromosomes, which are a representation of a parameter combination. Those chromosomes reside within an environment that has limited resources, and the competition for the fittest individual begins. This results in a rise of fitness of the whole population, finding fitter solutions to carry on to the next generations. Those most fit solutions will become the set of parameters currently applied on our machine.

Finding out if the configurations are not optimal is no easy task, as there are a very large number of available options. Our GA manages to optimize the configurations by picking the most fit individuals in each generation. The algorithm is supposed to be running non-stop, hence there will not be a predefined amount of generations it should stop after, just in the result section as we need the data for analysis. After running it for a few generations we can see improvement in performance.

Thanks to the crossover and mutation functions we are able to achieve fitter individuals while also not reaching a premature convergence due to the mutation probability that helps us keep the diversity in the population. Although for the part of finding top individuals, the improvements are not very big from first to x-other generation.

By increasing the values in different parameters we are enabling the increase of various factors, such as kernel threshold, memory usage limit, max amount of memory pages, amount of memory reserved for buffers etc. which results in a bigger throughput. But by increasing the throughput speed, we could be sacrificing some other parts of the service quality of network, something that we have been unable to detect yet.

Through various adjustments in the different tests, we were able to find a good set of factors that yields advantageous results. By testing different duration for testing it was found that longer duration throughput tests are more reliable unlike shorter intervals. Throughput speed is not something that can be measured over just one second, even though it made the iterations in our algorithm quicker. The increase in test duration made one run of our algorithm take over 10 hours, due to the high amount of parameters that all had to be tested over a 10 second interval each. This forced us to decrease the number of iterations for each test as there simply was not enough time.

An observation in results fig 4.14, shows that the throughput speed is indeed higher than those of the default settings, it might not increase twofold, but it did increase by around 65% for that specific experiment compared to the default settings. There were second thoughts on if the service quality of network did decrease. By running ping and hping3 (5) tests from the client to the server with traffic, we were able to deduct that the latency was not lost, same goes for the packets, all packets sent were received. This tells us that at least for the type of load that was generated from client to server in our experiments, there are no drawbacks that could be detected.

The achieved results on physical machines were satisfactory, but as different machines have different specifications a more standardized test was performed. For a VM, it allows us to experiment with different types of OS, test out how they feel, it is a sandbox where nothing gets out to the system enabling us to test risky configurations without harming the system. They can also be duplicated, if users on a system would be using VMs to connect to a server instead of physical machine, it could be a lot easier for the system administrator to adjust the network configurations on the server in order to satisfy most amount of users.

We conducted a series of experiments on the VMs, comparing the data gathered on tests where we changed the configurations directly inside the VM, and another one where we changed the configurations inside the host machine, testing the speed from client VM to server VM. The results did show that by changing configurations on the host machine instead of inside the VM directly the throughput was affected more heavily. Most probably, the reason behind this is the fact that the host machine is lending its resources to the VM, so the traffic has to go through the host machine before it reaches the VM. Another test which could have improved the results even further could be by changing the configurations both at the VM and host machine at once.

5.2 Error sources

5.2.1 Inconsistent traffic generation

First of all, the payload received for each session is different, hence making it harder to compare the results to each other, as the traffic that is going to the server is not the same for every experiment. How much data each NIC is managing to push through varies, the NICs are fighting for the bandwidth. A packet streaming method such as the one used in iPerf consists of sequences of packets separated by a steady interval. Those methods form the regular router performance tests such as those suggested in RFC 2889[36] and RFC 2544[37]. There are no easy ways to simulate network traffic, and there are various papers dealing with this topic, each suggesting different tools such as, [38] [39].

Chapter 6

Further work

In this Chapter, we give suggestions to our current work and list new features that can be implemented.

6.1 Improvements

This thesis could be improved further by testing out more variations of factor values, like crossover and mutation probabilities. Adding more parameters to the configuration set, and in addition the GA could support automatic parameter extraction. At the existing moment they are extracted from a list which we wrote, but every machine has different parameters. This could be done by browsing through all available parameters on the systems, putting them on a list and testing different values for each and everyone of them.

Another thing that could have been improved is by going up a layer, changing the configurations on different layers. If Apache web-server was used, which was the initial plan to f.ex. host videos, by changing the default Apache configurations on top of the configurations on the machine, the overall throughput could have been improved even further.

6.2 New features

6.2.1 Ability to adjust latency and throughput ratio

As a future work, it is possible provide the user (server-admin) with a choice on how much he wants to prioritize when it comes to the throughput over latency, all depending on his focus. For watching big video files the overall throughput is to be preferred. But if watching a football match live, the latency could be prioritized, as it is undesired for the quality of experience to notice such a delay in the case of a football match by hearing the neighbors celebrating a goal before the current user notices it. As well as places where you need fast response time, like in certain video games such as gun games, where good latency will give you an edge over the others.

6.2.2 A/B testing

An A/B test could have been performed in order to find an algorithm with best possible factors for selection, crossover and mutation. Having every instance try different probabilities, split the users and test different versions on them to see which one is preferable. One instance could have higher throughput, another have higher latency and a third one something in between.

6.2.3 Load-balancer

It would be efficient to divide the traffic between different machines so they all could contribute in the search for an optimal solution. With the introduction of a load-balancer between multiple physical machines or VMs the algorithm could be running at multiple instances at once. And if one instance finds a more fit parameter combination than what already was running, this combination could be the new top pick used.

Chapter 7

Conclusion

The main objective of this thesis was to improve the throughput speed by finding the correct set of configurations depending on the traffic, using a genetic algorithm. After various experiments we are able to conclude that the throughput speed did improve and is faster than that of the default settings. In fact, we were able to achieve a throughput speed increase by at least 65%. The fitness of the whole population raises from one generation to another, but the performances of the top individuals are pretty stationary. After discovering the top individual in the first generation, the speed does not increase significantly through more generations.

Not fully optimized throughput speed on a machine can be caused by many factors, the different relevant parameter values could be too low, not allowing for maximum send/receive capability. A wrong chain of configurations could also be the problem, something that is impossible or very hard to detect manually due the immense number of available combinations. Hence a genetic algorithm is being used to search for the best solution. Each set of parameters is represented as a chromosome, those chromosomes go through multiple crossover, mutation and selection processes and the most fit configuration is selected in every generation.

Going through those various operations means that our GA can find an optimized version of configurations exclusively for the type of traffic that is going through the server at a given time. An adapted approach to change network parameters dynamically has been developed, which can result in a significant increase of speed compared to the default settings. The latency is also not lost and stays equivalent to that of the default.

A series of experiments was performed on the VMs, resulting in a discovery in accordance with previous assumptions. Changing the configurations directly on the VM opposed to changing it on the host machine seems to be less effective. In both cases we are able to achieve faster throughput relative to the default, but changes on the host machine resulted in higher overall speed.

Bibliography

[1] Adam Buji. 'Genetic Algorithm For Tightening Security'. MA thesis. UIO, 2017 (accessed January 19, 2017). URL: https://www.duo.uio.no/handle/10852/58270.

[2] Eyal Zohar and Yuval Cassuto. 'Automatic and Dynamic Configuration of Data Compression for Web Servers'. In: *28th Large Installation System Administration Conference, LISA '14, Seattle, WA, USA, November 9-14, 2014.* 2014, pp. 97–108. URL: https://www.usenix.org/conference/lisa14/conference-program/presentation/zohar.

[3] Michael Gopshtein and Dror G. Feitelson. 'Trading off quality for throughput using content adaptation in web servers'. In: *SYSTOR*. 2011.

[4] Y. Diao et al. 'Managing Web Server Performance with AutoTune Agents'. In: *IBM Syst. J.* 42.1 (Jan. 2003), pp. 136–149. ISSN: 0018-8670. DOI: 10.1147/SJ.2003.5386833. URL: http://dx.doi.org/10.1147/SJ.2003.5386833.

[5] Haifeng Chen et al. 'Boosting the Performance of Computing Systems Through Adaptive Configuration Tuning'. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 1045–1049. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529511. URL: http://doi.acm.org/10.1145/1529282.1529511.

[6] Yuqing Zhu et al. 'BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning'. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: ACM, 2017, pp. 338–350. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128605. URL: http://doi.acm.org/10.1145/3127479.3128605.

[7] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015. ISBN: 3662448734, 9783662448731.

[8] J.E. Baker. *Reducing bias and inefficiency in the selection algorithm, In Grefenstette [198], pages 14–21.*

[9] Melanie Mitchell. *An Introduction to Genetic Algorithms. The MIT Press, 1996.*

[10] Breno Henrique Leitao. *Tuning 10Gb network cards on Linux*. 18 pp. URL: http://landley.net/kdocs/ols/2009/ols2009-pages-169-184.pdf.

[11] Konstantin Ivanov. *Linux TCP Tuning*. URL: http://www.linux-admins.net/2010/09/linux-tcp-tuning.html.

[12] A. P. Foong et al. 'TCP performance re-visited'. In: *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.* Mar. 2003, pp. 70–79. DOI: 10.1109/ISPASS.2003.1190234.

[13] I. H. Chung and J. K. Hollingsworth. 'Automated cluster-based Web service performance tuning'. In: *Proceedings. 13th IEEE International Symposium on High performance Distributed Computing, 2004.* June 2004, pp. 36–44. DOI: 10.1109/HPDC.2004.1323484.

[14] Wei Zheng, Ricardo Bianchini and Thu D. Nguyen. 'Automatic Configuration of Internet Services'. In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 219–229. ISSN: 0163-5980. DOI: 10.1145/1272998.1273020. URL: http://doi.acm.org/10.1145/1272998.1273020.

[15] X. Bu, J. Rao and C. Z. Xu. 'A Reinforcement Learning Approach to Online Web Systems Auto-configuration'. In: *2009 29th IEEE International Conference on Distributed Computing Systems*. June 2009, pp. 2–11. DOI: 10.1109/ICDCS.2009.76.

[16] YouTube. *YouTube press*. URL: https://www.youtube.com/yt/about/press/.

[17] Statistic Brain. *YouTube Company Statistics*. URL: https://www.statisticbrain.com/youtube-statistics/.

[18] Bill Reinstein. *Webcasts Mature as Marketing Tool (25 June 2001)*.

[19] Jon Dugan et al. *iPerf*. URL: https://iperf.fr/.

[20] HP Information Networks Division et al. *netperf*. URL: https://linux.die.net/man/1/netperf.

[21] 'Ethernet Jumbo Frames'. In: (Ethernet Alliance. 2009-11-12, accessed April 4, 2018). URL: http://ethernetalliance.org/library/whitepaper/ethernet-jumbo-frames/.

[22] Herbert Xu. *Generic Segmentation Offload*. URL: https://wiki.linuxfoundation.org/networking/gso.

[23] HAProxy. *HAProxy*. URL: http://www.haproxy.org/.

[24] V. Cardellini, M. Colajanni and P. S. Yu. 'Dynamic load balancing on Web-server systems'. In: *IEEE Internet Computing* 3.3 (May 1999), pp. 28–39. ISSN: 1089-7801. DOI: 10.1109/4236.769420.

[25] H. Bryhni, E. Klovning and O. Kure. 'A comparison of load balancing techniques for scalable Web servers'. In: *IEEE Network* 14.4 (July 2000), pp. 58–64. ISSN: 0890-8044. DOI: 10.1109/65.855480.

[26] Sandeep Sharma, Sarabjit Singh and Meenakshi Sharma. 'Performance Analysis of Load Balancing Algorithms'. In: (). URL: https://pdfs.semanticscholar.org/ec06/a6d2cba3a1aa77084b34f13e424725880294.pdf.

[27] *Python*. URL: https://www.python.org/.

[28] Jean Francois Puget. *The Most Popular Language For Machine Learning*. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en.

[29] Tim Peters. *TPEP20 – The Zen of Python[2004]*. URL: https://www.python.org/dev/peps/pep-0020/.

[30] Sean Dague and Daniel Stekloff. *virsh(1) - Linux man page*. URL: https://linux.die.net/man/1/virsh.

[31] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[32] David E. Goldberg. 'Zen and the Art of Genetic Algorithms'. In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 80–85. ISBN: 1-55860-006-3. URL: http://dl.acm.org/citation.cfm?id=93126.93153.

[33] D. Adler. 'Genetic algorithms and simulated annealing: a marriage proposal'. In: *IEEE International Conference on Neural Networks*. 1993, 1104–1109 vol.2. DOI: 10.1109/ICNN.1993.298712.

[34] Aravind Menon and Willy Zwaenepoel. 'Optimizing TCP Receive Performance'. In: *USENIX 2008 Annual Technical Conference*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 85–98. URL: http://dl.acm.org/citation.cfm?id=1404014.1404021.

[35] D. D. Clark et al. 'An analysis of TCP processing overhead'. In: *IEEE Communications Magazine* 27.6 (June 1989), pp. 23–29. ISSN: 0163-6804. DOI: 10.1109/35.29545.

[36] *RFC288*. URL: https://tools.ietf.org/html/rfc2889.

[37] S. Bradner and J. McQuaid. *RFC2544*. URL: https://www.ietf.org/rfc/rfc2544.txt.

[38] Paul Barford and Mark Crovella. 'Generating Representative Web Workloads for Network and Server Performance Evaluation'. In: *SIGMETRICS Perform. Eval. Rev.* 26.1 (June 1998), pp. 151–160. ISSN: 0163-5999. DOI: 10.1145/277858.277897. URL: http://doi.acm.org/10.1145/277858.277897.

[39] Joel Sommers and Paul Barford. 'Self-configuring Network Traffic Generation'. In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. IMC '04. Taormina, Sicily, Italy: ACM, 2004, pp. 68–81. ISBN: 1-58113-821-0. DOI: 10.1145/1028788.1028798. URL: http://doi.acm.org/10.1145/1028788.1028798.

Appendices

Appendix A

Firewall settings

SERVER

```
1
2     barserv@bar server:~$ sudo iptables  t nat  S
3  P PREROUTING ACCEPT
4  P INPUT ACCEPT
5  P OUTPUT ACCEPT
6  P POSTROUTING ACCEPT
7  A PREROUTING  i eno2  p tcp  m tcp   dport 0  j DNAT   to
     destination 192.168.121.252
8  A PREROUTING  i eno2  p tcp  m tcp   dport 90  j DNAT   to
     destination 192.168.121.252
9  A PREROUTING  i eno2  p tcp  m tcp   dport 5001  j DNAT   to
     destination 192.168.121.252
10 A POSTROUTING  o eno2  j MASQUERADE
11 A POSTROUTING  s 192.168.121.0/24  d 224.0.0.0/24  j RETURN
12 A POSTROUTING  s 192.168.121.0/24  d 255.255.255.255/32  j RETURN
13 A POSTROUTING  s 192.168.121.0/24 !  d 192.168.121.0/24  p tcp  j
     MASQUERADE  to ports 1024 65535
14 A POSTROUTING  s 192.168.121.0/24 !  d 192.168.121.0/24  p udp  j
     MASQUERADE  to ports 1024 65535
15 A POSTROUTING  s 192.168.121.0/24 !  d 192.168.121.0/24  j
     MASQUERADE
16
17    barserv@bar server:~$ sudo iptables  S
18 P INPUT ACCEPT
19 P FORWARD ACCEPT
20 P OUTPUT ACCEPT
21 A INPUT  i virbr0  p udp  m udp   dport 53  j ACCEPT
22 A INPUT  i virbr0  p tcp  m tcp   dport 53  j ACCEPT
23 A INPUT  i virbr0  p udp  m udp   dport 67  j ACCEPT
24 A INPUT  i virbr0  p tcp  m tcp   dport 67  j ACCEPT
25 A FORWARD  d 192.168.121.252/32  p tcp  m tcp   dport 0  j ACCEPT
26 A FORWARD  d 192.168.121.252/32  p tcp  m tcp   dport 5001  j
     ACCEPT
27 A FORWARD  d 192.168.121.252/32  i eno2  p icmp  j ACCEPT
28 A FORWARD  d 192.168.121.252/32  p icmp  j ACCEPT
29 A FORWARD  d 192.168.121.252/32  p tcp  m tcp   dport 90  j
     ACCEPT
30 A FORWARD  d 192.168.121.0/24  o virbr0  m conntrack   ctstate
     RELATED,ESTABLISHED  j ACCEPT
31 A FORWARD  s 192.168.121.0/24  i virbr0  j ACCEPT
32 A FORWARD  i virbr0  o virbr0  j ACCEPT
```

```
33 A FORWARD  o virbr0  j REJECT   reject with icmp port unreachable
34 A FORWARD  i virbr0  j REJECT   reject with icmp port unreachable
35 A OUTPUT  o virbr0  p udp  m udp   dport 68  j ACCEPT
```

Listing A.1: Server firewall settings

CLIENT

```
1    sudo iptables  t nat  S
2 P PREROUTING ACCEPT
3 P INPUT ACCEPT
4 P OUTPUT ACCEPT
5 P POSTROUTING ACCEPT
6 A PREROUTING  i eno2  p icmp  j DNAT   to destination
     192.168.122.12
7 A POSTROUTING  o eno2  j MASQUERADE
8 A POSTROUTING  s 192.168.122.0/24  d 224.0.0.0/24  j RETURN
9 A POSTROUTING  s 192.168.122.0/24  d 255.255.255.255/32  j RETURN
10 A POSTROUTING  s 192.168.122.0/24 !  d 192.168.122.0/24  p tcp  j
     MASQUERADE   to ports 1024  65535
11 A POSTROUTING  s 192.168.122.0/24 !  d 192.168.122.0/24  p udp  j
     MASQUERADE   to ports 1024  65535
12 A POSTROUTING  s 192.168.122.0/24 !  d 192.168.122.0/24  j
    MASQUERADE
13
14    barcli@bartek client:~$ sudo iptables  S
15 P INPUT ACCEPT
16 P FORWARD ACCEPT
17 P OUTPUT ACCEPT
18 A INPUT  i virbr0  p udp  m udp   dport 53  j ACCEPT
19 A INPUT  i virbr0  p tcp  m tcp   dport 53  j ACCEPT
20 A INPUT  i virbr0  p udp  m udp   dport 67  j ACCEPT
21 A INPUT  i virbr0  p tcp  m tcp   dport 67  j ACCEPT
22 A FORWARD  p icmp  j ACCEPT
23 A FORWARD  d 192.168.122.0/24  o virbr0  m conntrack   ctstate
     RELATED,ESTABLISHED  j ACCEPT
24 A FORWARD  s 192.168.122.0/24  i virbr0  j ACCEPT
25 A FORWARD  i virbr0  o virbr0  j ACCEPT
26 A FORWARD  o virbr0  j REJECT   reject with icmp port unreachable
27 A FORWARD  i virbr0  j REJECT   reject with icmp port unreachable
28 A OUTPUT  o virbr0  p udp  m udp   dport 68  j ACCEPT
```

Listing A.2: Client firewall settings

Appendix B

Best parameters

On Server

```
Fastest:
 sudo sysctl  w net.ipv4.tcp_keepalive_probes=7
sudo sysctl  w net.core.busy_poll=5
sudo sysctl  w net.ipv4.tcp_timestamps=0
sudo sysctl  w net.ipv4.tcp_tw_reuse=1
sudo sysctl  w net.core.optmem_max=10012802
sudo ifconfig eno2 mtu 2384
sudo sysctl  w net.ipv4.tcp_keepalive_intvl=51
sudo ifconfig eno2 mtu 1896
sudo sysctl  w net.core.flow_limit_table_len=8192
sudo sysctl  w net.core.somaxconn=3221
sudo sysctl  w net.core.rps_sock_flow_entries=19001
sudo sysctl  w net.ipv4.tcp_moderate_rcvbuf=1
sudo ifconfig eno2 txqueuelen 1241
sudo ifconfig eno2 txqueuelen 2273
sudo sysctl  w net.core.busy_read=20
sudo sysctl  w net.ipv4.tcp_timestamps=1
sudo sysctl  w net.ipv4.tcp_sack=1
sudo sysctl  w net.core.rmem_max=9173925
sudo sysctl  w net.ipv4.tcp_rmem='7038 250716 10226292 '
sudo sysctl  w net.ipv4.tcp_sack=1
sudo sysctl  w net.core.rmem_default=386582
sudo sysctl  w net.core.rmem_max=3181362
sudo sysctl  w net.ipv4.tcp_fin_timeout=35
sudo sysctl  w net.ipv4.tcp_no_metrics_save=1
sudo sysctl  w net.ipv4.tcp_no_metrics_save=1
sudo sysctl  w net.core.wmem_max=16579436
sudo sysctl  w net.core.rps_sock_flow_entries=12
 = 319.15
```

Listing B.1: Fast parameter combination on server

```
Slowest:
  sudo sysctl  w net.ipv4.tcp_keepalive_intvl=47
sudo sysctl  w net.ipv4.tcp_wmem='5298 125259 12163956 '
sudo sysctl  w net.ipv4.tcp_moderate_rcvbuf=0
sudo sysctl  w net.core.rmem_max=2606112
sudo sysctl  w net.ipv4.tcp_timestamps=1
sudo sysctl  w net.core.bpf_jit_enable=2
sudo sysctl  w net.ipv4.tcp_rmem='4881 367716 11882146 '
sudo sysctl  w net.ipv4.tcp_sack=1
```

```
sudo sysctl  w net.core.optmem_max=18534087
sudo sysctl  w net.core.netdev_max_backlog=3987
sudo sysctl  w net.core.busy_poll=72
sudo sysctl  w net.core.rmem_default=335829
sudo sysctl  w net.core.rps_sock_flow_entries=8611
sudo sysctl  w net.core.somaxconn=3103
sudo sysctl  w net.ipv4.tcp_no_metrics_save=1
sudo sysctl  w net.core.flow_limit_table_len=8192
sudo sysctl  w net.ipv4.tcp_fin_timeout=18
sudo sysctl  w net.ipv4.tcp_keepalive_probes=8
sudo sysctl  w net.core.busy_read=14
sudo ifconfig eno2 mtu 2215
sudo sysctl  w net.ipv4.tcp_mem='14114693 11064845 9406878 '
sudo sysctl  w net.ipv4.tcp_tw_reuse=0
sudo sysctl  w net.core.wmem_default=320824
sudo ifconfig eno2 txqueuelen 1292
sudo sysctl  w net.ipv4.tcp_window_scaling=0
sudo sysctl  w net.core.tstamp_allow_data=0
sudo sysctl  w net.core.dev_weight=381
 = 120.54
```

Listing B.2: Slow parameter combination on server

On VM

```
Fastest:
 sudo sysctl  w net.core.tstamp_allow_data=0
sudo sysctl  w net.ipv4.tcp_mem='11631319 6235725 16064622 '
sudo sysctl  w net.ipv4.tcp_tw_reuse=0
sudo sysctl  w net.core.somaxconn=2456
sudo sysctl  w net.core.rmem_default=382029
sudo sysctl  w net.ipv4.tcp_timestamps=1
sudo sysctl  w net.core.netdev_max_backlog=3644
sudo sysctl  w net.core.rmem_max=6948953
sudo sysctl  w net.ipv4.tcp_wmem='4588 552707 6579218 '
sudo sysctl  w net.core.bpf_jit_enable=1
sudo sysctl  w net.core.busy_poll=40
sudo sysctl  w net.core.optmem_max=10014646
sudo sysctl  w net.ipv4.tcp_keepalive_intvl=46
sudo sysctl  w net.core.wmem_max=11607647
sudo sysctl  w net.core.wmem_default=409160
sudo sysctl  w net.core.wmem_default=298922
sudo sysctl  w net.ipv4.tcp_tw_reuse=1
sudo sysctl  w net.core.optmem_max=24339982
sudo ifconfig ens3 mtu 1843
sudo ifconfig ens3 txqueuelen 3795
sudo sysctl  w net.ipv4.tcp_window_scaling=1
sudo sysctl  w net.ipv4.tcp_no_metrics_save=0
sudo sysctl  w net.ipv4.tcp_mem='14995330 1187345 11231161 '
sudo sysctl  w net.core.netdev_max_backlog=3079
sudo sysctl  w net.core.busy_read=14
sudo sysctl  w net.core.tstamp_allow_data=1
sudo sysctl  w net.ipv4.tcp_keepalive_intvl=74
 = 272.53
```

Listing B.3: Fast parameter combination on a VM

```
Slowest:
  sudo sysctl  w net.ipv4.tcp_rmem='6102 603703 10665932 '
sudo sysctl  w net.ipv4.tcp_fin_timeout=35
```

```
sudo sysctl  w net.ipv4.tcp_moderate_rcvbuf=1
sudo sysctl  w net.core.rmem_default=361314
sudo sysctl  w net.core.busy_poll=48
sudo sysctl  w net.ipv4.tcp_keepalive_intvl=51
sudo sysctl  w net.core.optmem_max=21383203
sudo sysctl  w net.core.tstamp_allow_data=1
sudo sysctl  w net.core.busy_read=11
sudo sysctl  w net.ipv4.tcp_keepalive_probes=9
sudo sysctl  w net.core.netdev_max_backlog=1570
sudo sysctl  w net.ipv4.tcp_sack=0
sudo sysctl  w net.ipv4.tcp_mem='16662110 15537604 1517732 '
sudo ifconfig ens3 txqueuelen 4637
sudo sysctl  w net.ipv4.tcp_no_metrics_save=0
sudo ifconfig ens3 mtu 2043
sudo sysctl  w net.core.flow_limit_table_len=4096
sudo sysctl  w net.core.rmem_max=7862294
sudo sysctl  w net.ipv4.tcp_tw_reuse=0
sudo sysctl  w net.ipv4.tcp_window_scaling=1
sudo sysctl  w net.ipv4.tcp_timestamps=1
sudo sysctl  w net.core.wmem_default=224800
sudo sysctl  w net.core.somaxconn=3880
sudo sysctl  w net.ipv4.tcp_wmem='6767 245945 11590058 '
sudo sysctl  w net.core.rps_sock_flow_entries=32105
sudo sysctl  w net.core.wmem_max=8275502
sudo sysctl  w net.core.dev_weight=179
 = 101.33
```

Listing B.4: Slow parameter combination on a VM

Appendix C

The algorithm

```
import matplotlib.pyplot as plt
import numpy as np
import random, time
import csv
from collections import OrderedDict
import itertools
from random import randint
import subprocess
import tqdm
import math

parcomb = []
throughspeed = []
parent = []


with open("nyparams.txt") as p:
    read = p.read()
    wrds = read.splitlines()
    para = []



    for w in wrds:
        if len(w) != 0:
            para.append(w.split(";"))


def nyparams():

    del parent[:]
    for p in range(len(para)):
        nyverdi = "'"

        stripmin = para[p][1].strip("'")
        stripmaks = para[p][2].strip("'")

        parmin = stripmin.split(" ")
        parmaks = stripmaks.split(" ")

        if len(parmin) == 3:
            for i in range(len(parmin)):
```

```
                nyrange = random.randint(int(parmin[i]), int(
    parmaks[i]))
                nyverdi += "%s " %str(nyrange)
            nyverdi += "'"

        else:
            mini = int(para[p][1])
            maks = int(para[p][2])
            nyverdi = str(random.randint(mini,maks))

        nycomb = para[p][0] + nyverdi
        parent.append(nycomb)


#generate parameter combo
def gen_paracombo(ant, combo, pop):
    perm = []

    for i in range(0,pop):

        randperm = random.sample(range(0,(len(para))), ant)
        perm.append(randperm)

    return perm


def convert(combo):

    cnt = 0

    for com in combo:
        #randomize parametervalue in every parameter
        nyparams()
        paracombo = []

        for i in com:
            paracombo.append(parent[i])
        speedtest(paracombo, cnt)
        cnt += 1

def speedtest(params, cnt):

    paramset = []
    combspeed = []

    process = subprocess.Popen(['sh','old.sh'], stdout=subprocess.
    PIPE, stderr=subprocess.PIPE)
    process.wait()

    open('paratest.sh', 'w').close()
    with open('paratest.sh', 'a') as file:
        for p in params:
            file.write('%s\n' % (p))

    process2 = subprocess.Popen(['sh','paratest.sh'], stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
    process2.wait() # Wait for process to complete.


```

```
    for p in params:
        paramset.append(p)

    parcomb.append(paramset)


    # ssh barcli@10.0.0.2  p 2222  i ~/.ssh/master "sh scr.sh" |
    tail  1 | awk '{print $NF}'
    thro = subprocess.Popen(['sh','throughput.sh'], stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
    thro.wait()

    for line in thro.stdout.readlines():
        throughput = line

    throughput = throughput.decode('UTF 8')

    throughspeed.append(float(throughput.strip("\n")))
    print (throughput)

def printpam(combo):
    alt = " "
    for c in combo:
        alt += c + "\n"
    return alt

def selection(ant, genn, gens, avrg, selprob, crossprob, mutprob):
    genn += 1
    print ("GENERATION %d" % genn)
    print ("                                ")

    print "throughspeed\n%s"%throughspeed

    parcomb2 = parcomb[:]
    del parcomb[:]
    del throughspeed[:]

    for i in parcomb2:
        speedtest(i, 0)

    print "throughspeed\n%s"%throughspeed

    bestfit = []
    kortest = np.array(throughspeed)
    best = kortest.argsort()
    top.append(kortest[best[  1]])

    print ("best: %s" % kortest[best[  1]])
    print ("verst: %s" % kortest[best[0]])

    #testing the default parameters for every iteration
    oldtest()

    cntr = 0
    for p in parcomb:
        for pp in p:
            l = "2"
        cntr += 1

```

```
156     gens  = 1
157
158
159
160     print ("%.0f prosent av parents crosses:" % (selprob*100))
161     xpop = int(len(parcomb) * selprob)
162
163     print ("xpop =")
164     print (xpop)
165
166     if(xpop%2 != 0):
167         xpop  = 1
168
169     # velger de xpop raskeste kombinasjonene
170     doublecomb = [] # par av parents som er raskest fra beste par
        til verste
171
172     for p in range(int((xpop/2))):
173         bestpar = []
174         for i in kortest.argsort()[ xpop:]:
175             bestpar.append(parcomb[i][:])
176         doublecomb.append(bestpar)
177
178
179     #sletter xpop treigeste
180     slet = slett(xpop, best, kortest)
181
182     if slet:
183         print "empty"
184         print slet
185         kortest = slet[0]
186         best = slet[1]
187
188     bottom.append(kortest[best[0]])
189
190     avrg.append(np.mean(kortest.astype(np.float)))
191     print ("AVERAGE:")
192     print (avrg)
193     print ("Fastest:\n%s = %s" % (printpam(parcomb[best[  1]]),
        kortest[best[  1]]))
194     print ("\n")
195     print ("Slowest:\n %s = %s" % (printpam(parcomb[best[0]]),
        kortest[best[0]]))
196
197     #om det har gatt x antall generasjoner stopper lokka og den
        beste
198     #ruta blir printet ut etter x antall generasjoner
199     if gens <= 0:
200
201         #velger de 2 raskeste kombinasjonene
202         for i in kortest.argsort()[  2:]:
203             bestfit.append(parcomb[i][:])
204
205         stdv = []
206         kort = bestfit[  1][:]
207         kort.append(bestfit[  1][0])
208
209         for i in (range(len(parcomb))):
210             stdv.append(float(throughspeed[i]))
```

```
211
212         print ("Fastest:\n%s = %s" % (printpam(parcomb[best[  1]]),
     kortest[best[  1]]))
213         print ("Slowest %s = %s" % (printpam(parcomb[best[0]]),
    kortest[best[0]]))
214         print ("Standard deviation: %.2f , Mean: %.2f" % (np.std(
    stdv), np.mean(stdv)))
215
216         return (avrg)
217
218
219     for j in range(len(doublecomb)):
220         crossover(ant, doublecomb[j], best, kortest, crossprob,
    mutprob)
221
222     selection(ant, genn, gens, avrg, selprob, crossprob, mutprob)
223
224 #sletter to treigeste
225 def slett(ant, best, kortest):
226
227     slett = []
228     for i in range(ant):
229
230         parcomb.remove(parcomb[best[0]])
231         throughspeed.remove(kortest[best[0]])
232
233         kortest = np.array(throughspeed)
234         best = kortest.argsort()
235
236     return kortest, best
237
238
239 def test():
240     cntr = 0
241     for p in parcomb:
242         for pp in p:
243             print (pp)
244         print ("                    ")
245
246 def crossover(ant, bestfit, best, kortest, crossprob, mutprob):
247     index3 = []
248     index3val = []
249     rand3 = 0
250     rand = random.sample(range(0,ant), 1)
251
252     parent1 = bestfit[  1][:]
253     parent2 = bestfit[  2][:]
254
255     size = min(len(parent1), len(parent2))
256     for i in range(size):
257         if random.uniform(0, 1) < crossprob:
258             parent1[i], parent2[i] = parent2[i], parent1[i]
259
260     p = randint(0, ant  1)
261
262     if (randint(0,100) < mutprob):
263         print ("Mutation on parent1")
264         parent1[p] = mutationparam(ant)
265     if (randint(0,100) < mutprob):
```

```
266         print ("Mutation on parent2")
267         parent2[p] = mutationparam(ant)
268 
269     #legge til offspring 1 og 2 i parcomb
270     speedtest(parent1, 0)
271     speedtest(parent2, 1)
272 
273 def oldtest():
274     process = subprocess.Popen(['sh','old.sh'], stdout=subprocess.
    PIPE, stderr=subprocess.PIPE)
275     process.wait()
276 
277     thro = subprocess.Popen(['sh','throughput.sh'], stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
278     thro.wait()
279 
280     for line in thro.stdout.readlines():
281         oldthrough = line.decode('UTF 8')
282 
283 
284     oldspeed.append(float(oldthrough.strip("\n")))
285 
286     print (oldthrough)
287 
288 def mutationparam(ant):
289 
290     p = randint(0, ant)
291 
292     nyverdi = "'"
293     print (len(para[p][1]))
294     print (len(para[p][2]))
295     stripmin = para[p][1].strip("'")
296     stripmaks = para[p][2].strip("'")
297     #print "stripmin %s" % stripmin
298 
299     parmin = stripmin.split(" ")
300     parmaks = stripmaks.split(" ")
301 
302     if len(parmin) == 3:
303         for i in range(len(parmin)):
304             nyrange = random.randint(int(parmin[i]), int(parmaks[i
    ]))
305             nyverdi += "%s " %str(nyrange)
306         nyverdi += "'"
307 
308 
309     else:
310         mini = int(para[p][1])
311         maks = int(para[p][2])
312         nyverdi = str(random.randint(mini,maks))
313 
314     nycomb = para[p][0] + nyverdi
315     return nycomb
316 
317 
318 def divergence(F):
319     """ compute the divergence of n D scalar field `F` """
320     return reduce(np.add,np.gradient(F))
321 
```

```
322 def indexadder(arr):
323     addedarr = []
324 
325     for i in range(len(arr[0])):
326         tot = 0
327         for j in range(len(arr)):
328             tot+=arr[j][i]
329 
330         tot=(tot/len(arr))
331         addedarr.append(tot)
332 
333     return addedarr
334 
335 def convergence(arr):
336     minst = []
337     for i in range(len(arr)):
338         dev = []
339         check = 0
340         for j in range(10):
341             if i+10 < len(arr):
342                 dev.append(arr[i+j])
343                 check = 1
344 
345         if check == 1:
346             minst.append(np.std(dev))
347             check = 0
348 
349     nymin = []
350     for i in range(len(minst) 1):
351         # 1: compares 2 results right after eachother, if the
    difference between them is not bigger than that of the
    smallest stdv
352         # 2: does the same, but now check if the value is not
    negative
353         # 3: checks if the current stdv value is less than twice
    as big as the lowest stdv value
354         # Overall, first smallest stdv found which doesn't have
    too much difference compared to the smallest one, no extreme
    values
355         if (minst[i+1] minst[i]) < min(minst) and (minst[i+1]
    minst[i]) > 0 and minst[i] < (min(minst) + min(minst)):
356             print (minst[i+1] minst[i])
357             minstindex = i
358             print arr[i]
359             print i
360             print "return"
361             return minst, minstindex
362         else:
363             minstindex = minst.index(min(minst))
364 
365     return minst, minstindex
366 
367 start = time.time()
368 
369 av = []
370 
371 ant = 27
372 gens = 40
373 pop = 80
```

```
selprob = 0.1
crossprob = 0.5
mutprob = 16

totalavrg = []
totalold = []
totaltop = []
totalbottom = []
div = []


for i in tqdm.tqdm(range(10)):
    time.sleep(0.01)
    del parcomb[:]
    del throughspeed[:]
    del parent[:]

    print "                          Iteration number %d
                            " % i
    top = []
    bottom = []
    oldspeed = []
    avrg = []
    combo = []

    combo = gen_paracombo(ant, combo, pop)
    convert(combo)
    (selection(ant, 0, gens, avrg, selprob, crossprob, mutprob))

    totalavrg.append(avrg)
    totalold.append(oldspeed)
    totalbottom.append(bottom)
    totaltop.append(top)

    file = open('vmtests.txt', 'a')

    file.write("ant: %d gens: %d pop: %d  selprob: %.1f  crossprob
    : %.1f  mutprob: %d" % (ant, gens, pop, selprob, crossprob,
    mutprob))
    file.write("\n                    %d\ntotalavrg\n"%i)
    file.write("%s" % totalavrg)

    file.write("\ntotalold\n")
    file.write("%s" % totalold)

    file.write("\ntotalbottom\n")
    file.write("%s" % totalbottom)

    file.write("\ntotaltop\n")
    file.write("%s\n\n" % totaltop)


avrg2 = indexadder(totalavrg)
oldspeed2 = indexadder(totalold)
bottom2 = indexadder(totalbottom)
top2 = indexadder(totaltop)

print ("avrg2:")
print avrg2
```

```
430
431 print ("oldspeed2:")
432 print oldspeed2
433
434 print("bottom2")
435 print bottom2
436
437 print("top2")
438 print top2
439
440 print ('%.3f%s' % ((end   start), 'sec'))
441 end = time.time()
```

Listing C.1: Genetic algorithm