Tampere University

Niko Jäntti

DETECTING ANOMALIES IN SERVER PERFORMANCE

Master's thesis
Faculty of Information
Technology and Communication
Sciences
Kari Systä
David Hästbacka
May 2020

ABSTRACT

Niko Jäntti: Detecting anomalies in server performance
Master's thesis
Tampere University
Computing Sciences
May 2020

This thesis studies ways to detect anomalies in server performance and tests simple implementations on detect two different anomalies: CPU related anomalies and memory leaks. The goal is to find a way to implement a lightweight analysis component. This thesis was done as an assignment for Patria Aviation Oy.

Common system performance metrics are presented of which CPU and memory utilization are also used later in thesis. A literary study was done to find different strategies on anomaly detection, including both statistical methods and machine learning methods. The simplest strategies, z-score and regression analysis using ordinary least squares, were used for implementation and testing.

Testing the implementations showed that z-score was suitable to detecting the anomalies that we were looking for. However, simple linear regression was not robust enough even with smoothed data.

Keywords: z-score, performance, anomaly

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Niko Jäntti: Palvelimen suorituskykypoikkeamien havaitseminen
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2020

Tässä diplomityössä tutustutaan palvelinjärjestelmien suorituskyvyn kannalta olennaisiin metriikoihin ja niiden hyödyntämiseen poikkeamien havaitsemiseksi. Työssä toteutetaan yksinkertaiset implementaatiot kahdenlaisten poikkeamien havaitsemiseksi: muistivuodot ja suorittimesta johtuvat suorituskykypoikkeamat. Tutkimus toteutettiin toimeksiantona Patria Aviation Oy:lle.

Työssä esitellään suorituskykyyn liittyviä metriikoita, joista osaa käsitellään myöhemmin poikkeamien havaitsemiseksi. Kirjallisuuskatsauksessa tutkittiin aikaisempia tutkimuksia aiheesta ja tutustuttiin sekä tilastollisiin menetelmiin että koneoppimisen menetelmiin. Toteutusten pohjaksi valittiin z-score sekä pienimmän neliösumman menetelmä niiden yksinkertaisuuden vuoksi.

Toteutukset testattiin ja huomattiin että z-score pohjainen menetelmä sopii löytämään työssä etsittyjä poikkeamia. Sen sijaan pienimmän neliösumman menetelmä osoittautui virheherkäksi datalle tehdystä silottelusta huolimatta.

Avainsanat: z-score, suorituskyky, poikkeama

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

This thesis was written while working for Patria Aviation Oy. I would like to thank Jouni Malinen for having faith in me and my co-workers for making the workdays feel less like workdays. Also, I would like to thank Kari Systä for supervising this thesis.

Lastly, I would like to thank my wife for support.

Tampere, 3 May 2020

Niko Jäntti

CONTENTS

1. INTRODUCTION 1
2. BACKGROUND 3
2.1 System metrics 3
2.1.1 CPUs 4
2.1.2 Memory 4
2.1.3 Disks 4
2.1.4 Network 4
2.2 Performance anomalies 5
2.3 Existing solutions 6
2.3.1 Nagios 6
2.3.2 Datadog 7
2.3.3 New Relic APM 7
2.3.4 dynatrace 7
3. ANOMALY DETECTION 8
3.1 Statistical methods 8
3.1.1 Student's t-test 8
3.1.2 Z-score 10
3.1.3 Regression analysis 10
3.2 Machine learning methods 11
3.2.1 Clustering 11
3.2.2 Neighbour-based 12
3.3 Previous studies 12
3.4 Selecting suitable methods 12
4. IMPLEMENTING AN ANOMALY DETECTION COMPONENT 14
4.1 Third-party libraries 14
4.1.1 Shogun 14
4.1.2 ALGLIB 14
4.2 Implementation 14
4.3 Input Data 15
4.4 Output 15
4.5 Data Storing 15
4.6 Data Processing 16
4.6.1 Model forming 16
4.6.2 CPU related performance anomaly detection 16
4.6.3 Memory leak detection 19
5. EVALUATION 24
5.1 Test system 24
5.2 Test Data 24
5.3 Results 27
6. CONCLUSIONS 30
REFERENCES 32

LIST OF SYMBOLS AND ABBREVIATIONS

CPU	Central Processing Unit
GiB	Gibibyte, 1024^3 bytes
GPU	Graphics Processing Unit
Hz	Hertz, unit of frequency
IOPS	Input/output operations per second
ms	Milli second
PaaS	Platform as a Service
RAM	Random Access Memory
SaaS	Software as a Service
TCP/IP	Transmission control protocol / Internet protocol

.

1. INTRODUCTION

As the number of different computer systems is growing, the importance of fault detection grows too. While manual monitoring on some systems might be reasonable, autonomous performance analysis not only offers possibilities to detect performance issues but also to pinpoint the root cause.

Companies, services, even health care relies on some type of computer systems. Such systems require monitoring in order to ensure uninterrupted uptime. Developing systems with proper computational capacity requires knowledge in performance requirements. Hunting down a cause for long latencies in an online store adds costs in system administration as well as in lost sales.

This thesis is done as an assignment for Patria Aviation Oy. There is a need for a software component that detects anomalies in server performance in run-time based on resource utilization. Those anomalies should be automatically detected so system administrators can be alerted.

The main goal of this thesis is to find answers to the following questions:

- How to detect anomalies in server performance?
- How to pinpoint the root cause?
- How to implement an adequate analysis component with minimal performance overhead?

The main server system is rather unique and resembles an embedded system, and that's why this thesis only refers to the more universal qualities of the system to make the results more widely usable. However, the implementation will be tested based on the behaviour that is relevant for the main system.

In chapter 2, background on system performance evaluation is presented. This includes different measurable metrics and what kind of performance anomalies one could look for. Also, some commercially available solutions are presented. In chapter 3, few different theoretical approaches on anomaly detection are presented. Chapter 4 describes the implementation of the chosen approach and briefly discusses some existing solutions for the algorithms. Also, a description of the underlaying server system

is given on a level that is relevant for the thesis. In chapter 5, the implemented component is evaluated for both, the performance overhead and its ability to detect anomalies. Lastly in chapter 6, the evaluation results are analysed and the thesis is briefly summed up.

2. BACKGROUND

This chapter goes through theory on system metrics and performance anomalies and briefly presents few system monitoring solutions. The following definitions are based on Brendan Gregg's definitions in [5].

2.1 System metrics

In a computer system, system's resources are the physical components, such as CPU, GPU, memory, network interface and hard or solid-state drives. Each component can have different measurable metrics, such as utilization, throughput and saturation. These metrics are usually provided by an operating system.

Resource **utilization** can either be time-based or capacity-based. Time-based utilization expresses how long a resource was active during a time period. For example, if CPU has utilization of 75 % with a sampling rate of 1 Hz, it means that CPU was active for 750 ms and in idle for 250 ms. Capacity-based utilization expresses the consumed capacity of a resource. For example, if a system has 32 GiB of memory and memory utilization of 75 %, it means that the system has 24 GiB of memory in use and 8 GiB of memory free. Another difference between time-based and capacity-based utilization, is how the resource behaves under full utilization. When a resource's time-based utilization reaches 100 %, the resource can still accept more work. Instead, capacity-based utilization of 100 % means that the resource cannot accept any more work. Both definitions are used in this thesis, CPU utilization being time-based and memory utilization being capacity-based.

Saturation begins when a resource's utilization reaches 100 %, meaning that work must be queued. As stated before, time-based utilization and capacity-based behave differently under full utilization. When writing to a disk that has capacity-based utilization of 100 %, all new work is being queued, whereas a CPU with time-based utilization of 100 % may be able to accept new work and thus might not be saturated even with 100 % utilization.

Disk I/O is measured in **IOPS**, or Input/output operations per second. Input and output operations refer to writes and reads respectively.

Throughput, or more precisely data throughput in this case, is usually measured in bits per second. It measures systems data rate and is usually used on networking communication but could also be used to describe disk IOPS.

2.1.1 CPUs

CPUs are the centre of a system. Modern systems have multiple CPUs and can utilize parallelism.

A **processor** is a hardware chip that contains one or more CPUs. CPUs can be either cores or hardware threads. A **core** is an independent CPU whereas some CPU architectures enable having multiple CPU instances on one core as **hardware threads**. An example being Intel's Hyper-Threading [9].

A **run queue** is a queue of processes or threads that are waiting to be run on a CPU. A run queue is managed by the kernel scheduler, that assigns software threads to run on CPUs. Software threads usually keep running on the same CPUs to preserve cached data in the CPU cache.

2.1.2 Memory

The running applications and their data is stored in memory, often referred to as RAM. Systems use an abstraction of memory called **virtual memory** to extend the usable memory beyond physical memory. When the physical memory is filled, the system starts **paging**, i.e. transferring memory pages between memory and storage devices.

Memory saturation refers to physical memory saturation. Memory becomes saturated when system runs out of available physical memory. This means that paging is a sign of memory saturation.

2.1.3 Disks

Disks are the primary storage devices on a system. They can be either mechanical hard disk drives (HDDs) or solid-state drives (SSDs).

Systems can also have **virtual disks**. Virtual disk is an abstraction of one or more physical disks, that are shown as one.

2.1.4 Network

Network is a fundamental part of modern systems, especially cloud systems. Network can be divided to hardware and software. The hardware side usually contains network interfaces, switches and gateways. The software side is the used protocol stack, most commonly TCP/IP.

2.2 Performance anomalies

Statistical anomalies are unpredictable occurrences in data. In terms of server performance, this could be abrupt drops or rises in CPU utilization over time, as shown on Figure 1.

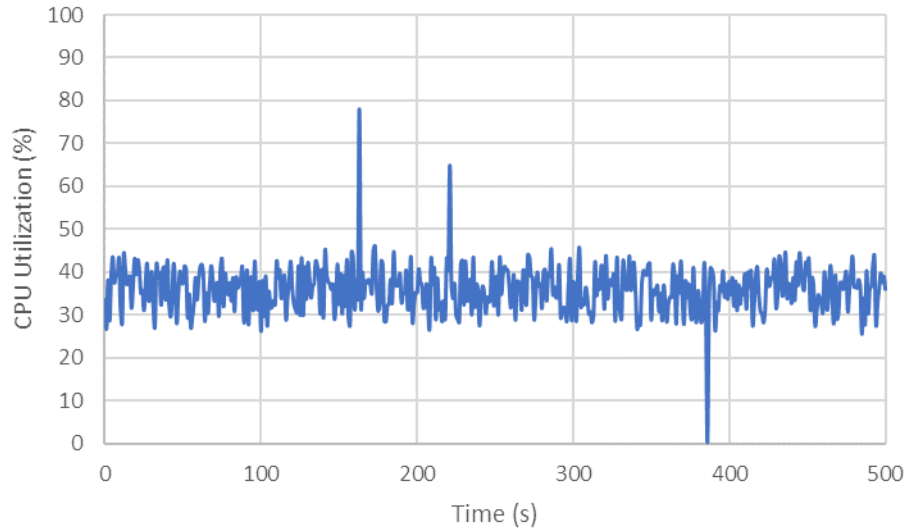

Figure 1. *CPU utilization with three anomalies.*

Bottlenecking is a situation, where a resource is exhausted so that it limits the performance of a system [5]. A plotted CPU utilization of an exhausted CPU can be seen on Figure 2.

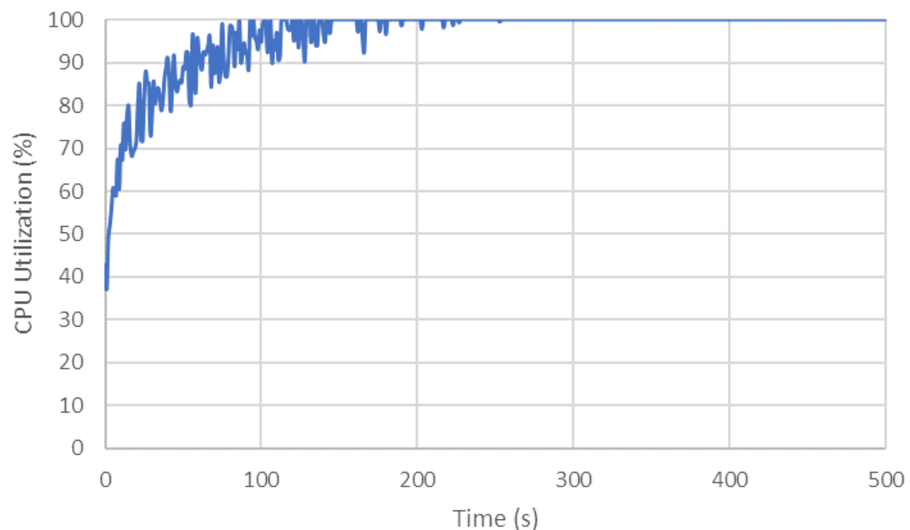

Figure 2. *CPU utilization of a saturated CPU.*

It's commonly stated that CPU could be saturated when the utilization goes over 60 percent [5], especially when the sampling rate is slower than 1 Hz. This is because the

sampling rate could hide short bursts of 100 % utilization. That's why also the CPU run queue length should be checked. The run queue length tells the amount of processes waiting for CPU time as well as the amount of processes currently running. This means that CPU is saturated if the run queue length is above the number of CPU cores.

Memory leak is a situation where an application allocates memory but doesn't release it after it is no longer needed and the reference to the memory is deleted. Memory leaks are common in software made with programming languages, such as C or C++, where the programmer responsible of releasing memory before deleting the reference.

Similarly to memory leak, memory hogging is a situation where an application keeps on allocating memory without releasing it, either on purpose or due to poor software design. The difference to memory leak being, that the references to memory are not deleted. Instead, the application keeps the rarely used references, for example due to improperly large cache.

Both memory leaks and memory hogging could create a rising slope in applications, or systems memory leading to excessive memory utilization. This could lead to a situation where there is not enough memory for applications to keep running correctly. Meaning that applications fail to allocate memory and thus being terminated or limiting their operation in some way, or the operating system starts to page memory to a hard drive. As memory has considerably faster access times, this has a negative effect on systems performance. Later in this thesis, both situations, memory leaks and memory hogging, are referenced as memory leaks.

In this thesis performance anomalies refer to unusual negative effects on system performance. Unusually high or low CPU utilization isn't considered a performance anomaly unless it also affects some performance metric, such as throughput or latency, negatively.

2.3 Existing solutions

There are commercial products available for system performance monitoring. Many products offer a wide range of features on monitoring for different use cases. Here are some of the products as part of the background study.

2.3.1 Nagios

Nagios is a popular infrastructure monitoring tool. It offers monitoring of applications, operating systems, network protocols and infrastructure. Nagios can be used for performance anomaly detection with static threshold values, but it also supports third-

party addons so it could be enhanced with an analysis component. Nagios can also analyse Linux logs to detect failed processes and other application and system level failures. [13]

2.3.2 Datadog

Datadog offers cloud-based solutions on system monitoring SaaS for per month pricing. It has integration possibilities with popular cloud services, applications and databases. Datadog is targeted at developers as well as system admins to get a full view of the infrastructure. It offers a customizable real-time dashboard and API access, alerting system and visualization. [2]

2.3.3 New Relic APM

New Relic APM (application performance monitoring) is one of New Relic’s system monitoring products. Like Datadog, it offers a real-time visualization and customizable monitoring. However, it is meant for monitoring an application’s performance instead of system level monitoring. [14]

2.3.4 dynatrace

Dynatrace offers an all-in-one solution for system and application monitoring. Advertising automatic anomaly detection, monitoring from infrastructure to application level and integration possibilities. [3]

3. ANOMALY DETECTION

This section describes theoretical methods that can be used on anomaly detection. There is a previous literary study made on performance anomaly detection methods that presents a wider spectrum of studied performance anomaly detection methods. These presented methods were found during a literary study. Each method is evaluated for suitability for the described system. Two main categories were found: statistical methods and machine learning methods.

3.1 Statistical methods

In statistical methods, a profile of system's current behaviour is compared to a profile of correctly working system. Statistical methods often expect that the data follows Gaussian distribution, or normal distribution. This means that anomalies, or outliers, can be interpreted as values that are significantly far for populations normal.

3.1.1 Student's t-test

Student's t-test is a method developed to test whether there is an actual difference between the means of two samples [11]. The t-test can be used to do one-sided or two-sided test. Two-sided test can be used to tell if the sample is deviated from population to either direction. One-sided test can be used to tell whether the sample is above or below the population, but not both.

Student's t-test defines the statistic t by the formula

$$t = \frac{\bar{X} - \mu}{\hat{\sigma}/\sqrt{n}}, \tag{3.1}$$

where $\bar{X}$ is the mean of tested sample, n is the sample size, $\hat{\sigma}$ is standard deviation and μ is the mean of the population. Student's t statistic follows Student's t-distribution that is a probability density function

$$f(t) = \frac{\Gamma\left(\frac{v+1}{2}\right)}{\sqrt{v\pi}\,\Gamma\left(\frac{v}{2}\right)}\left(1 + \frac{t^2}{v}\right)^{-\frac{v+1}{2}}, \tag{3.2}$$

where Γ is the gamma function and v is degrees of freedom [6]. With t-test the degrees of freedom is $n - 1$. A cumulative distribution function can then be calculated by integrating over the probability density function

$$F_n(x) = \int_{-\infty}^{x} f(t)\, dt. \tag{3.3}$$

Applying the calculated statistic t or its absolute to the $F_n(x)$ tells the p-value of the test sample. The p-value can then be used to tell whether the test sample's mean is significantly different from the population's mean. For example, testing whether the sample's mean is different from population's mean with a probability level of 95 % is done by testing

$$1 - F_n(|t|) > \frac{\alpha}{2}, \tag{3.4}$$

where α is significance level calculated by $1 - 0.95$, when using probability level of 95 %. The sample's mean is significantly different if the equation 3.4 isn't true.

The t-test is simple and designed to work with limited sample sizes. However, Student's t-test assumes that the samples have equal variances. Hence a *Welch's t-test* was developed. Welch's t-test can be used with both unequal variances and unequal sample sizes. Welch's t-test defines the statistic t by the formula

$$t_w = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}, \tag{3.5}$$

where $\bar{X}_1$ and $\bar{X}_2$ are means, s_1^2 and s_2^2 are variances and N_1 and N_2 are sample sizes of samples one and two. Welch's statistic t follows a distribution that is more complicated than Student's t-distribution. To overcome this, Bernard L. Welch came up with an approximation

$$t_w \dot{\sim} t(df_w), \tag{3.6}$$

where degrees of freedom is calculated by

$$df_w = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}. \tag{3.7}$$

The p-value can then be calculated using the previously presented cumulative probability function.

Hence the Student's t-test compares means, it cannot be used to analyse singular values. Instead, it could be used to compare larger samples to pre-known populations, i.e. comparing CPU utilization to a known baseline.

3.1.2 Z-score

The standard score, or z-score, is a number that tells observation's distance from populations mean. More precisely, z-score tells how many standard deviations an observation is above or below the population's mean. A positive z-score value indicates that the observation is above the mean and respectively a negative z-score value indicates that the observation is below the mean.

Z-score can be used to compare new data points to existing data. Z-score of an observation x is calculated by

$$z = \frac{x-\mu}{\sigma}, \tag{3.8}$$

where μ is the population mean and σ is the standard deviation. [4]

There is a robust peak detection algorithm that is developed around z-score [20]. The algorithm takes a timeseries data input and three user inputs: *lag* as the length of data used for smoothing, *threshold* as a value that defines which observations are considered peaks and *influence* as factor for how much a peak affects the smoothing. Each new observation is compared against the smoothed data by calculating a z-score. If the absolute of the z-score is greater than the defined threshold, then the observation is considered a peak.

3.1.3 Regression analysis

In regression analysis past events are used to form a regression model. The formed regression model can be used to evaluate whether there is a rising or declining trend in data. Regression analysis also reveals whether the data points are too far from the model for a trend to be considered. Considering the nature of input data and the goal in this thesis, we are interested in Simple Linear Regression.

Simple Linear Regression is a statistical model that simplifies data to linear function, i.e. first-degree polynomial

$$y_i = \alpha + \beta x_i + \varepsilon_i, \tag{3.9}$$

where α is the *y*-intercept, β is the slope of the line, ε_i is the error term and y_i and x_i are the compared variables [12]. The Simple Linear Regression model can be calculated using Ordinary Least Squares. With Ordinary Least Squares the aim is find α and β that minimize the sum of squared errors, error being the distance between sample y values and predicted y_i values.

The slope can be calculated by

$$\beta = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - \sum x_i \sum x_i}, \tag{3.10}$$

where n is the number of samples. The slope value β tells whether there is a rising of decreasing trend in the regression line. Leaving out the error term, the intercept can then be calculated with equation 3.9 using sample averages

$$\alpha = \frac{\sum y_i}{n} - \frac{\beta \sum x_i}{n}. \tag{3.11}$$

Although not very robust, Simple Linear Regression is chosen for detecting memory leaks and memory hogging for its simplicity. The implementation is presented in 4.6.3

3.2 Machine learning methods

Machine learning algorithms can be divided roughly into three categories: unsupervised, supervised and semi-supervised. In unsupervised learning patterns in the input are learned without explicit feedback. The most common example is clustering, where clusters of potentially useful data in input are detected. In supervised learning algorithms are manually taught the links between inputs and outputs, thus making the learning data labelled. Semi-supervised learning is a mixture of the two above. The learning is done with both labelled and unlabelled data. Due to their popularity, clustering and neighbour-based algorithms were chosen for closer inspection. [17]

3.2.1 Clustering

In clustering algorithms, the goal is to detect clusters of potentially useful data in the input. The most popular clustering method currently is k-means clustering. In the k-means algorithm, clusters are formed with k being the predefined number of clusters and their centroid being the mean of the values in cluster. The algorithm first selects k objects from a given group, forming initial clusters. Then each remaining object is assigned to a cluster whose centroid is closest, commonly based on Euclidean distance. After all objects have been assigned to a cluster, the cluster centroids are recalculated. The last two steps are repeated until neither produce changes. [7]

K-means aims to minimize a squared error function

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} d(x_i, c_j) \tag{3.12}$$

where k is number of clusters,n is number of samples $d(x_i, c_j)$ is the distance between point x_i and centroid c_i. The Euclidean distance with n dimensions is calculated by

$$d(x_i, c_j) = \sqrt{\left(x_{i_1} - c_{j_1}\right)^2 + \left(x_{i_2} - c_{j_2}\right)^2 + \cdots + \left(x_{i_n} - c_{j_n}\right)^2}. \tag{3.13}$$

The algorithm needs a proper k value and a lot of data to be able to identify the actual clusters in data. That's why the k-means algorithm was rejected in this thesis.

3.2.2 Neighbour-based

Neighbour-based algorithms classify input data based the closest data point of the population. A common neighbour-based algorithm is the kth-nearest neighbour.

The kth-nearest neighbour algorithm is a non-parametric classification method. The algorithm compares a test sample to it k nearest training samples based on the Euclidean distance. The test sample is then assigned to the category with the most samples. The Euclidean distance is calculated with equation 3.13. [18]

As with k-means clustering, also the results of kth-nearest neighbour depend highly on the choice of the value k and it need a lot of data. That's why the kth-nearest neighbour algorithm was rejected in this thesis.

3.3 Previous studies

Performance anomaly detection has been studied a lot. A literary study showed wide range of methodologies and approaches. There are also studies in the current situation of performance anomaly detection, such as [8] and [15]. Here we present some studies done on the field.

Many of the studies are done on cloud platforms. A solution for a PaaS system is introduced in [10], in which service level objectives are observed and violations trigger workload and application runtime data analysis for root cause identification. They use statistical methods to detect anomalies.

Another study presents their Performance Anomaly Detector in [16]. Their solution is a configurable tool that helps developers and administrators troubleshoot multi-server systems.

3.4 Selecting suitable methods

During the literary study, it became clear that there is a great deal of papers and studies done on the field of anomaly detection. Newer studies seem to lean towards cloud-based systems. As the aim on this thesis is to keep the implementation as light as possible and the analysis is done on running data, the machine learning methods were rejected as,

based on the literary study, they gave the impression of being too heavy for the task on hand.

Regression analysis with ordinary least squares was chosen as the memory leak detection method because of its simplicity and computational lightness. To make ordinary least squares more robust, z-score is used to smooth out noisy data.

Z-score was chosen for CPU related performance anomaly detection. The robust peak detection algorithm that is based on z-score, seems promising for detecting drops in throughput and is used as a basis for CPU related performance anomaly detection.

Z-score is chosen for CPU related anomaly detection. The implementation is presented in 4.6.2.

4. IMPLEMENTING AN ANOMALY DETECTION COMPONENT

The underlaying system consists of servers that are connected using unreliable connections. It can't be presumed that the status data produced by the servers is constantly available to a dedicated server. Hence the anomaly detection must be done locally, and the higher-level information is reported onwards to ensure more reliable results.

As the component is a part of a larger application, it is implemented with C++. The operating system running on the servers is Debian 10. The choice of using no third-party libraries was originally made to avoid issues with licensing and to ensure that the evaluation results aren't affected by any third-party implementations. Also, the chosen methods are simple and using third-party libraries was decided unnecessary.

4.1 Third-party libraries

Third-party libraries were studied as a part of the background study. Here are some existing libraries found for anomaly detection and machine learning. The libraries were not tested so they are presented just to knowledge that there are already existing solutions that could be used to solve these problems.

4.1.1 Shogun

Shogun is an open source machine learning toolbox implemented in C++. It offers interfaces for many popular programming languages. It has implementations for regression, clustering, neural networks and so on. [19]

4.1.2 ALGLIB

ALGLIB is a data processing library, that provides implementations for general statistical algorithms, decision forests, time series analysis, clustering and so on. ALGLIB is available in free and commercial editions. [1]

4.2 Implementation

While designing the component, several challenges needed to be solved. Challenges such as amount of data to store and what kind of anomalies are relevant. To solve these

and other challenges, the system’s throughput and memory utilization were studied, to get a basic idea of systems behaviour. These need to be adjusted accordingly if the system’s expected behaviour changes.

The component has two threads, a worker thread for data processing and data output and another for input data handling. This is, because the input data is passed on after reading, to make sure that it won’t be delayed by data processing. This is also to support different input intervals. The input data handler simply reads the input and stores the values.

4.3 Input Data

The component receives resource utilization data every five seconds. The data includes CPU utilization in percent per core, CPU run queue length and memory utilization in bytes. All the values are averages measured every five seconds. The resource utilization data is collected from the /proc pseudo-filesystem. In addition, the component receives the core application’s throughput as bytes in three seconds with the same interval.

The data includes both system wide resource utilizations and process specific resource utilizations. The process specific data differs from system wide data only on CPU utilization. The system wide data includes utilization (or load percentage) per core, whereas process specific data only has total CPU utilization.

4.4 Output

After analysing the input data, the component produces results per system resource. If a CPU related anomaly in throughput values or a possible memory leak is detected, the component sends a warning based on the criteria specified in subsections 4.6.2 and 4.6.3. The component also passes on the input data to other components, for example to be displayed in a graphical user interface.

4.5 Data Storing

The component stores data only in memory to make analysis, so the component doesn’t perform any disk i/o operations. To have a fixed length container for resource utilization history, *deque* was chosen as the datatype. This allows a fixed length, circular buffer type of storage, that adds new values to the end and removes the oldest from front. To avoid race conditions on writing and reading data, the deque is wrapped inside a class with thread safe read and write operations.

4.6 Data Processing

There are two different performance issues that we are trying to detect: CPU related anomalies in throughput and memory leaks. Therefore, two different approaches were implemented. There are some adjustable parameters on both approaches that require some knowledge on the analysed system.

4.6.1 Model forming

It's possible to use preformed models from a known data with no anomalies and compare that to sample data or use history to form a model during runtime. Using predefined models is more robust but requires more manual work. Each known good state of the system would need a model, but in contrast the models would be more reliable and thus the analysis results would be more predictable. Runtime model forming is less robust but doesn't require updating models manually in case changes to the system are made. Unfortunately, this usually also means that the model will get used to anomalous data after a while.

To keep the component flexible, the model for CPU related performance anomaly detection is formed runtime. However, the algorithm will try to prevent the model from getting used the anomalous data. Memory leak detection algorithm uses a simple predefined model of a with a straight horizontal trendline.

The component could be modified to use preformed models, with small effort. However, with the chosen approach that would mean that all known situations would have to be modelled.

4.6.2 CPU related performance anomaly detection

For CPU related performance anomaly detection, the CPU utilization is coupled with the systems throughput. We are interested in situations where the server's throughput is lower than it is supposed to. That's why the z-score based peak detection algorithm is only used to detect negative peaks. If a negative peak is detected and CPU utilization is above 60 percent on next, current or previous sample, then the run queue length is checked for CPU saturation. The anomaly detection algorithm's flowchart is presented in Figure 3. This algorithm is executed in a loop with a pre-set rate.

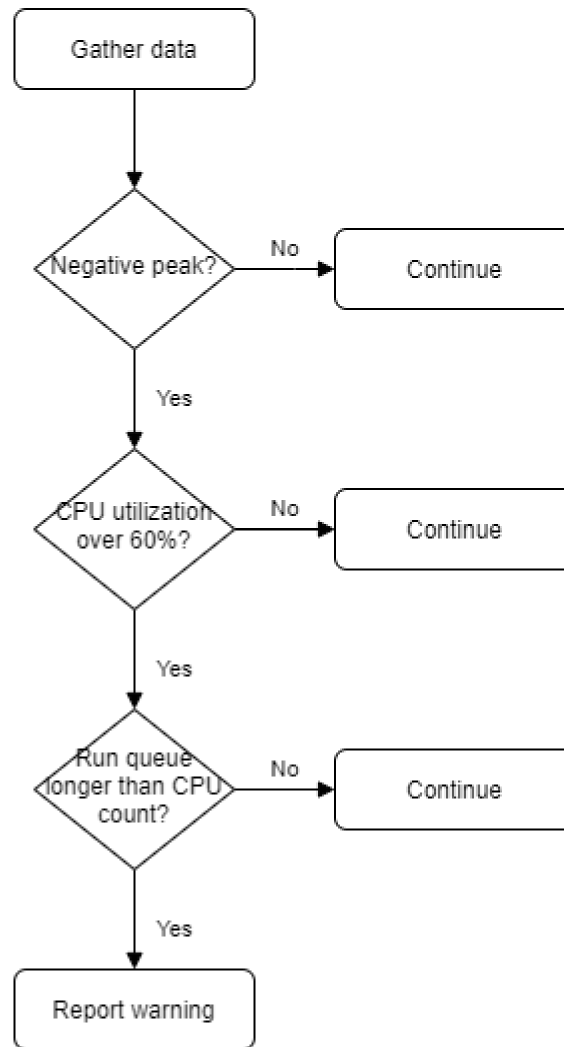

Figure 3. *CPU related performance anomaly detection flowchart.*

On first run the peak detection algorithm must be initialized. The initialization of the peak detection algorithm is shown in Program 1. The initialization function takes five input arguments. First argument is a deque type container *data,* that holds the data to be analysed. The second argument is another deque type container *filtered,* that holds the data filtered with the algorithm. The third argument *windowSize* determines the “lag” of the algorithm, i.e. the amount of history that new values are compared against. The fourth argument *influence* determines the amount of influence a value that is labelled as a peak has on the filtered history. It can have a value between zero and one. The fifth argument *threshold* determines how many deviations away a value must be in order to be considered a peak.

The *data* container must have at least the amount of *windowSize + 1* of data*,* because that amount of history is needed to form a base model before the results are meaningful. The first *windowSize* values of unfiltered data form the base model. The next values in

data are then analysed one by one. If the absolute of the z-score of a new value against the filtered history is greater than *threshold*, the new value is considered a peak. The non-peak values are added to the *filtered* container as is and the peak values are blended with the last value in *filtered* with the ratio of *influence* and the blended value is then added to *filtered* container. With *influence* of one, the peak values affect the filtered history just like non-peak values and with *influence* of zero the last value in *filtered* is duplicated. Lastly the function's return value tells if the last value was a peak and whether it was positive or negative.

The code uses functions *mean()* and *stdev()*, that are simple template functions for calculating the average and standard deviation of the values in a container. For their simplicity they are not included in the code samples. The original algorithm stores the calculated signal values, i.e. whether there is a peak, but this implementation only reports the result with the latest value. This is because on runtime analysis we are only interested of the current situation.

```
 1  void PeakDetectionInit(std::deque<double>& data,
 2                         std::deque<double>& filtered, size_t windowSize,
 3                         double influence, double threshold)
 4  {
 5    int signal = 0;
 6
 7    if (data.size() > windowSize)
 8    {
 9      double avg = 0;
10      double sdev = 0,
11
12      //initialize with unfiltered data
13      filtered = std::deque<double>(data.begin(),
14                                    data.begin() + windowSize);
15      for (auto i = windowSize; i < data.size(); i++)
16      {
17        avg = mean(filtered);
18        sdev = stdev(filtered);
19        if (std::fabs(data.at(i) – avg) / sdev > threshold)
20        {
21          if (data.at(i) > avg)
22            signal = 1;
23          else
24            signal = -1;
25          filtered.push_back(influence * data.at(i) +
26                             (1 - influence) * filtered.back());
27        }
28        else
29          filtered.push_back(data.at(i));
30      }
31    }
32    return signal;
33  }
```

Program 1. *Peak detection algorithm initialization.*

After initialization, all new values are evaluated with the same algorithm. On the subsequent rounds the new values are passed to the function shown in Program 2. The function in Program 2 takes the same input arguments as the initialization function. The *filtered* container now contains the history of filtered values, so the initialization is skipped and all the values in the input argument *data* are analysed against the existing history in *filtered* container. As stated earlier, the function will only report whether the last value is considered a peak.

```
1  int PeakDetection(std::deque<double>& data,
2                    std::deque<double>& filtered, size_t windowSize,
3                    double influence, double threshold)
4  {
5    for (auto i = 0; i < data.size(); i++)
6    {
7      avg = mean(filtered);
8      sdev = stdev(filtered);
9      if (std::fabs(data.at(i) – avg) / sdev > threshold)
10     {
11       if (data.at(i) > avg)
12         signal = 1;
13       else
14         signal = -1;
15       filtered.push_back(influence * data.at(i) +
16                          (1 - influence) * filtered.back());
17     }
18     else
19       filtered.push_back(data.at(i));
20   }
21   return signal;
22 }
```

Program 2. *Peak detection for new values.*

If either function reports a negative peak also the CPU utilization is checked. If CPU utilization is above 60 % also the run queue length is checked. A run queue length greater than the systems CPU count yields a warning on CPU related performance anomaly. Although there is an anomaly in throughput, this algorithm only reports a warning if it is caused by CPU bottleneck.

4.6.3 Memory leak detection

The memory leak detection discussed here includes memory hogging detection. In practice the algorithm is a simple upward slope detection algorithm based on simple linear regression with least squares. Because the least squares algorithm is not very robust, a simple algorithm was developed around it. When there is a predefined amount

of data, the data is smoothed, and peaks are removed. The smoothed data is then used to form a trendline with ordinary least squares. The target system is designed to work without swap partition to avoid paging. That’s why an upward slope is reported as a memory leak and used to forecast remaining time before running out of memory. If the forecast shows less than 30 minutes, it also gets reported. The algorithms flowchart is presented in Figure 4.

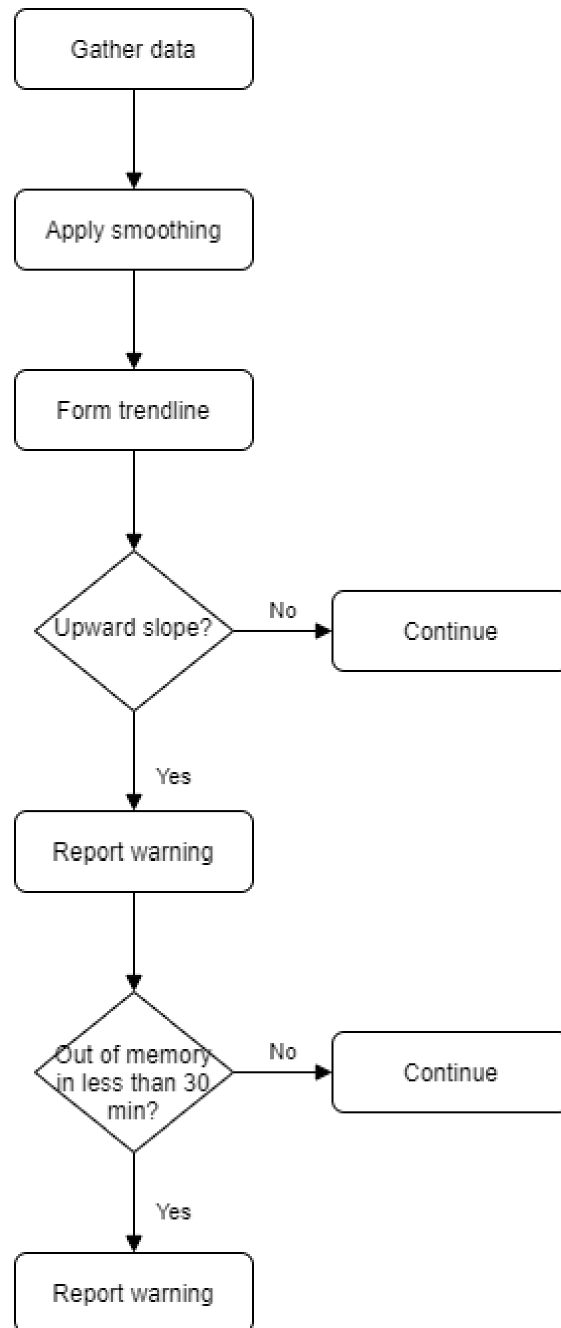

Figure 4. *Memory leak detection algorithm.*

Smoothing is done using first the same data filtering that was used in the peak detection algorithm introduced in 4.6.2. So first the data is fed to the peak detection algorithm. The filtered data is then smoothed even further by calculating the average of the filtered data. The average is calculated once every *windowSize* round so the averages are not calculated from overlapping data. For example, with a *windowSize* of 90 and input data rate of 1/5 seconds an average is calculated once every 450 seconds. This way we can store data on longer time span with less memory usage. This will also affect the least squares algorithms performance positively.

The implementation of the trendline forming algorithm is presented in Program 3. This implementation is for time series data, but it could be easily modified to analyse correlation of two different metrics. The function takes four input arguments. The first argument is a deque type container *yValues,* that holds time series data. The last three arguments, *slope, SEE* and *intercept* are used to pass the calculated slope, intercept and standard error of the estimate. The function simply forms a simple linear regression model using ordinary least squares on lines 12-22. Then it calculates the standard error of the estimate.

```
1  void leastSquares(std::deque<double>& yValues, double& slope,
2                    double& SEE, double& intercept)
3  {
4    double xSum = 0.0;
5    double ySum = 0.0;
6    double x2Sum = 0.0;
7    double xySum = 0.0;
8    double yEst = 0.0;
9    double yAvg = mean(yValues);
10   double yEESum = 0.0;
11
12   for (size_t i = 0; i < yValues.size(); i++)
13   {
14     xSum += i;
15     ySum += yValues.at(i);
16     x2Sum += i * i;
17     xySum += i * yValues.at(i);
18   }
19   slope = yValues.size() * xySum - xSum * ySum;
20   slope /= yValues.size() * x2Sum - xSum * xSum;
21   intercept = ySum - slope * xSum;
22   intercept /= yValues.size();
23
24   for (size_t i = 0; i < yValues.size(); i++)
25   {
26     yEst = slope * i + intercept;
27     yEESum += (yEst - yValues.at(i)) * ( yEst - yValues.at(i));
28   }
29   SEE = std::sqrt(yEESum / (yValues.size() - 2));
30 }
```

Program 3. *Trendline calculation using least squares.*

After forming the simple linear regression model, the model is then analysed. The analysis is shown in Program 4.

```
1    leastSquares(smoothedData, slope, SEE, intercept);
2
3    double errorMargin = SEE / smoothedData.size();
4
5    if (slope > errorMargin)
6    {
7      sendLeakWarning();
8
9      slope /= windowSize;
10     double secondsLeft = (RAMTotal - intercept) / slope;
11     secondsLeft *= dataInterval;
12
13     if (secondsLeft < 30 * 60)
14       sendTimeWarning(secondsLeft);
15   }
```

Program 4. *Interpreting least squares regression analysis.*

After executing the *leastSquares* function the standard error of the estimate is used to determine a threshold value for upward slope warning. If the *slope* is greater than the *errorMargin*, we assume there is a suspicious upward slope in memory utilization, and we send a warning. Then the forecasted time to out of memory in seconds is calculated using the *slope, intercept* and *RAMTotal,* that holds the total memory available. If *secondsLeft* is less than 1800 seconds (30 minutes), another warning is sent.

5. EVALUATION

Firstly, the component was evaluated by measuring detection rates and false positive rates under different situations and parameters. Secondly, the effect on system's performance was tested by measuring the time it takes to make the analysis. The component's detection ability was tested with simulated data to cover wider spectrum of cases in shorter time span. The data is fed to the algorithms just like I would be with real time data, i.e. one data point at a time.

The components functionality was tested for both, the CPU related anomaly detection and memory leak detection.

5.1 Test system

The test system is a Lenovo P50 laptop with Debian 10 running on a virtual machine. The virtual machine has 4 CPUs and 4 GiB of memory. These values are also used when testing the component.

5.2 Test Data

The system's behaviour was first observed to get an understanding of its usual resource utilization. This includes normal utilization levels and how the utilization changes, i.e. does the utilization increase and decrease smoothly or rapidly or does it stay steady. Based on this, the test data was generated to include anomalies that are relevant for the specific main system.

The CPU utilization and run queue length over time were simulated to have three models: low, medium and high. The model *low* has CPU utilization of around 30 % and run queue length topping at 2. The model *medium* has CPU utilization of around 65 % and run queue length topping at 6. The model *high* has CPU utilization of above 90 % and run queue length constantly above 4.

The simulated throughput over time was simulated to range from 149 800 bits per second to 150 200 bits per second with some drops. The throughput over time is plotted in Figure 5.

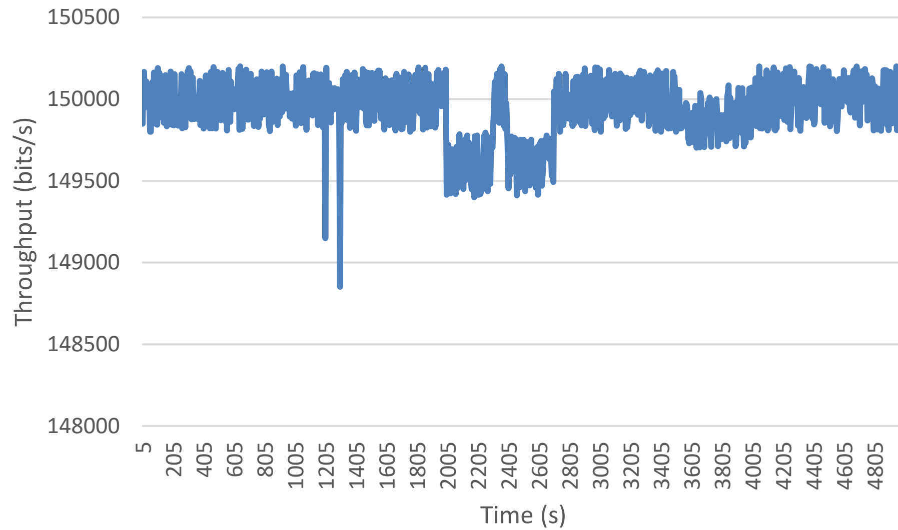

Figure 5. *Throughput over time.*

The CPU utilization and run queue length are coupled with simulated throughput over time. The throughput data is the same on all tests and the CPU utilization and run queue length model changes between tests.

The memory utilization over time was simulated with three models. The first model *slope* is a model with upward slope starting from around 3 042 MiB and ending to 4045 MiB with a time span of 8 hours and 20 minutes. The first model is plotted in Figure 6. The model also has some sudden upward and downward deviations. This model is simulating a situation that should raise both warnings.

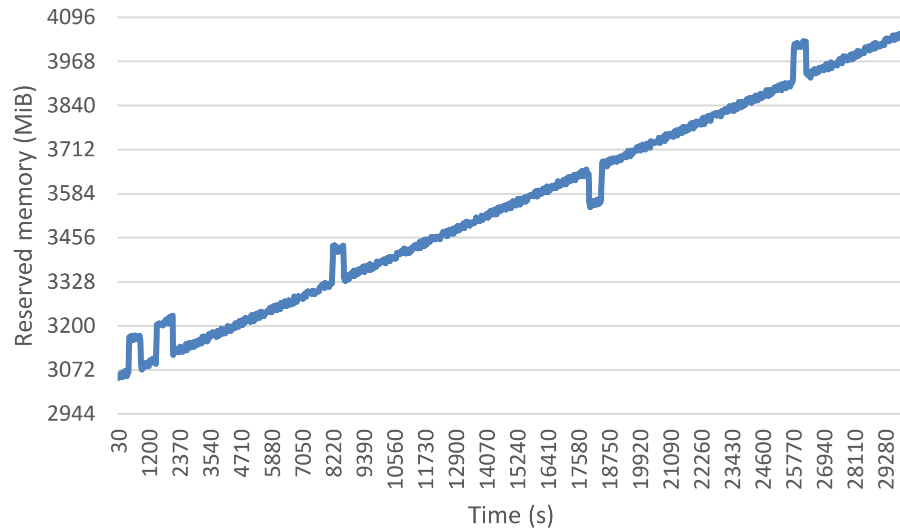

Figure 6. *Reserved memory over time with upward slope.*

The second model *decline* has a declining slope showing a situation that isn't considered a memory leak. The second model is plotted in Figure 7.

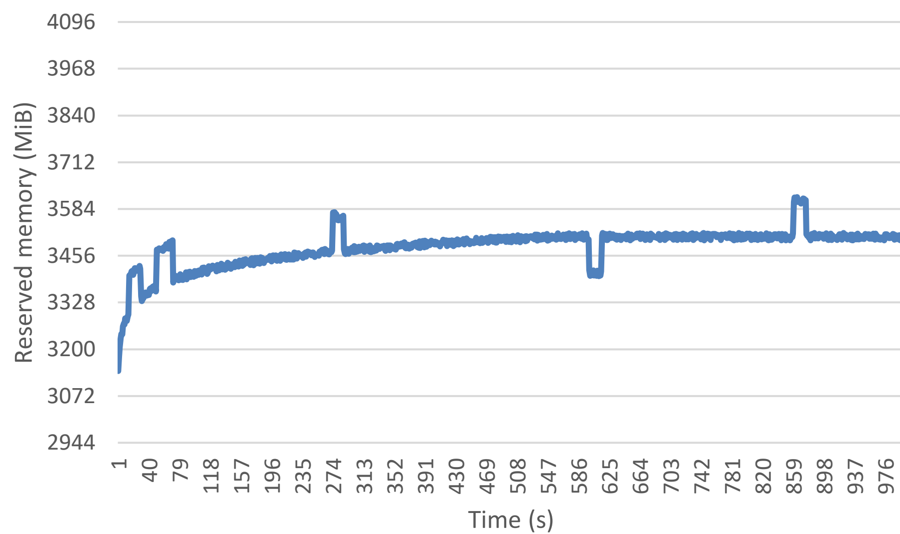

Figure 7. *Reserved memory over time with declining slope.*

The third model *flat* has no slope and the utilization follows basically a horizontal line. However, there is some noise and the last value is peaking. This is not supposed to cause a warning. The third model is plotted in Figure 8.

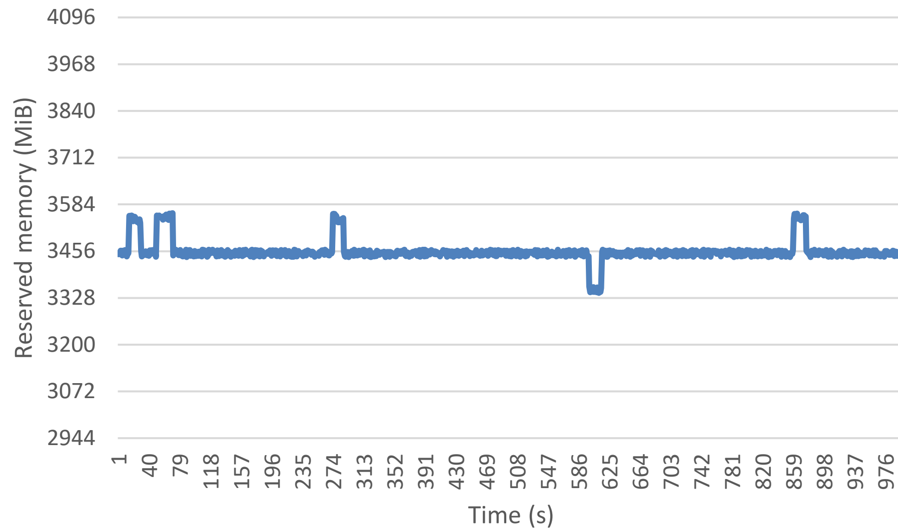

Figure 8. *Reserved memory over time with flat line.*

5.3 Results

The test results of the CPU related performance anomaly detection on model *low* are shown in Table 1.

Table 1. *Results of CPU related performance anomaly detection on model low.*

Window Size	Influence	Threshold	Anomalies	Detections	False Positives
30	1	3	0	0	0
30	0.5	3	0	0	0
60	1	3	0	0	0
60	0.5	3	0	0	0

The test results of the CPU related performance anomaly detection on model *medium* are shown in Table 2.

Table 2. *Results of CPU related performance anomaly detection on model medium.*

Window Size	Influence	Threshold	Anomalies	Detections	False Positives
30	1	3	5	4	0
30	0.5	3	5	3	0
60	1	3	5	4	0
60	0.5	3	5	3	0

The test results of the CPU related performance anomaly detection on model *high* are shown in Table 3.

Table 3. *Results of CPU related performance anomaly detection on model high.*

Window size	Influence	Threshold	Anomalies	Detections	False Positives
30	1	3	5	4	0
30	0.5	3	5	3	0
60	1	3	5	4	0
60	0.5	3	5	3	0

The component was able to detect 3 out of the 5 anomalies in the test data on both models *medium* and *high* with influence of 0.5 and 4 anomalies with influence of 1. In addition, the component reported no false positives. The component was unable to detect the fifth anomaly.

The test result of the memory leak detection on model *slope* is shown in Table 4.

Table 4. *Results of memory leak detection on model slope.*

Window size	Influence	Threshold	Detection	Out of memory warning
90	0.5	3	yes	yes

The test result of the memory leak detection on model *decline* is shown in Table 5.

Table 5. *Results of memory leak detection on model decline.*

Window size	Influence	Threshold	False Positive	Out of memory warning
90	0.5	3	yes	no

The test result of the memory leak detection on model *flat* is shown in Table 6.

Table 6. *Results of memory leak detection on model flat.*

Window size	Influence	Threshold	False Positive	Out of memory warning
90	0.5	3	no	no

The component was able to detect the memory leak and report the out of memory warning on the first simulated cases. Also, the component reported a false positive on the model *decline*.

The time to execute both CPU related anomaly detection and memory leak detection was measured 1000 times after the data containers reached their set limits. Running both algorithms took less than 10 ms on average. The CPU related anomaly detection analysis took less than 4 ms on average and the memory leak detection analysis took less than 4 ms on average.

6. CONCLUSIONS

The aim of this thesis was to familiarize with system performance analysis and find ways to detect anomalies in server performance. The background study gave answers to the first two research questions on detecting anomalies and finding the root cause.

The literary study on anomaly detection and testing the implemented component gave a partial answer to the last research question on implementing an adequate analysis component. The question was answered only partially, because although the component was light enough performance wise, the component's ability to detect memory leaks wasn't good enough.

The CPU related anomaly detection worked well on the problem in hand. The implementation was able to detect 4 anomalies out of 5 with appropriate parameters. It also didn't report any false positives. The one anomaly that was left undetected was below the threshold value. Having a more aggressive parameters would probably have caused the anomaly to be detected, but it would have caused the component to report false positives. The algorithm does require knowledge on system's resource usage, but also knowledge in what kind of anomalies need to be detected. This solution also detects very specific kind of anomalies so it's not very universal approach. The implementation was considered light enough with average run time of less than 4 ms.

Even with smoothing the memory leak detection algorithm wasn't very robust, as expected. Although being simple to implement and follow, it also requires a lot of knowledge of the systems performance to be able to set the parameters right. It is also rather unreliable with smaller data sizes and could lead to high false positive rates. With inappropriate *windowSize* the algorithm cannot distinguish declining raise from an actual suspicious raise in memory utilization. Although the algorithm performed quite well on forecasting remaining time before system runs out of memory in our simple tests. The algorithm also requires data from quite a long period of time before the results are relevant. The implementation was considered light enough with average runtime of less than 4 ms. These results will be used to continue the development of a system performance analysis component.

This thesis leaves room for further study on automatic system state evaluation. The method used to detect CPU related performance anomalies could be further studied to link other resources and performance metrics, for example by linking disk input and output operations to throughput. Linking CPU utilization, temperatures and fan speeds

could possibly be used to predict unusual temperature conditions and prevent possible hardware failures.

The literary study showed that there are plenty of ways to detect anomalies. The real challenge appears to be defining the actual metrics that give relevant results for the specific system. Designing a performance anomaly detection component for more universal systems would require less specific approach.

REFERENCES

[1] ALGLIB, https://www.alglib.net/ (visited 29.4.2020)

[2] Datadog, https://www.datadoghq.com (visited 29.4.2020)

[3] Dynatrace, https://www.dynatrace.com (visited 29.4.2020)

[4] R.J. Freund, J. Rudolf, W. J. Wilson, Statistical methods, 2nd ed, San Diego, Calif: Academic Press, 2003

[5] B. Gregg, Systems performance enterprise and the cloud, Upper Saddle River, NJ: Prentice Hall, 2014

[6] B. Grigelionis, Student's t-Distribution. Lovric M. (eds) International Encyclopedia of Statistical Science, Springer, 2011

[7] P. Hartono, Competitive Learning. Seel N.M. (eds) Encyclopedia of the Sciences of Learning, Springer, 2012

[8] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth, Performance Anomaly Detection and Bottleneck Identification, ACM Computing Surveys (CSUR), vol.48, no.1, Sep 2015, p. 1–35

[9] Intel, Intel Hyper-Threading Technology, Intel Corporation, website, Available: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (visited 29.4.2020)

[10] H. Jayathilaka, C. Krintz, R. M. Wolski, Detecting Performance Anomalies in Cloud Platform Applications, IEEE Transactions on Cloud Computing, Feb 2018, p. 1-1

[11] D. Kalpić, N. Hlupić, M. Lovrić, Student's t-Tests, Lovric M. (eds) International Encyclopedia of Statistical Science. Springer, 2011

[12] D. C. Montgomery, E. A. Peck, G. G. Vining, Introduction to linear regression analysis, 5th ed, Hoboken: Wiley, 2012

[13] Nagios, https://www.nagios.com/ (visited 2.5.2020)

[14] New Relic, https://www.newrelic.com (visited 29.4.2020)

[15] S. Nousiainen, J. Kilpi, P. Silvonen, M. Hiirsalmi, Anomaly detection from server log data. A case study, VTT Tiedotteita – Reseach Notes 2480, 2009, Available: https://www.vttresearch.com/sites/default/files/pdf/tiedotteet/2009/T2480.pdf

[16] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, C. Konig, PAD: Performance Anomaly Detection in Multi-server Distributed Systems, 2014 IEEE 7th International Conference on Cloud Computing, June 2014, p. 769–776

[17] S. Russell, P. Norvig, E. Davis, D. D. Edwards, D. Forsyth, N. J. Hay, J. M. Malik, V. Mittal, M. Sahami, S. Thrun, Artificial Intelligence: A Modern Approach, Third edition, Pearson, 2016

[18] T. Seidl, Nearest Neighbor Classification. LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems, Springer, 2009

[19] Shogun, https://github.com/shogun-toolbox/shogun (visited 29.4.2020)

[20] Stack Overflow Peak signal detection in realtime timeseries data, https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data/22640362#22640362 (visited 29.4.2020)