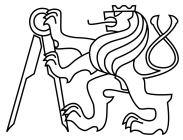
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Včasné varování před problémy infrastruktury informačního systému
Student:	Bc. Martin Beránek
Vedoucí:	Ing. Tomáš Vondra
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

V současné době je při rozvoji robustních informačních systémů nutné pracovat s monitoringem důležitých součástí systémů. Včasná reakce na selhání či vyčerpání prostředků je nutnou součástí správy.
Analyzujte současné možnosti monitoringu infrastruktury s důrazem na metody prediktivního monitoringu a agregace alarmů s určováním prapůvodní příčiny problému (root cause analysis).
Vytvořte prediktivní monitoring infrastruktury informačního systému na základě již vybudovaného dohledového systému. Na data sbíraná z existujícího systému aplikujte statistické nástroje, klasifikační techniky nebo strojové učení s cílem předpovídat alarmy dříve, než nastanou nebo zmenšit množství alarmů a obsluze prezentovat přednostně ty důležité.
Výsledek otestujte v reálném prostředí nebo alespoň zpětně na zachycených datech.
Výstupem bude aplikace napojená na dohledový systém schopná v reálném čase s dostatečnou pravděpodobností registrovat rizikové stavy systému a prezentovat je obsluze.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 22. srpna 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

Early Warning about Problems in Information System Infrastructure

Bc. et Bc. Martin Beránek

Supervisor: Ing. Tomáš Vondra (vondrto6)

5th January 2018

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Vondra for endless hours of consultations. I am very grateful to the company *Lukapo* which provided data and great environment for development. I have to thank Mall Group Czech Republic for providing data at the beginning of the project. My special thanks goes to Ing. Hanka Hurychová, because of the hours spent on grammar corrections.

Declaration

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

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

In Prague on 5th January 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Beránek, Martin. *Early Warning about Problems in Information System Infrastructure*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá tématy spojenými s předpovědmi dat a analýzou příčiny závad v informačních systémech. Používá statistické metody a metody strojového učení, které mohou být použity pro předpověď závad. Tyto předpovědi administátor může použít pro nápravu situace předtím, než dojde k poruše. Pomocí statistických metod je vytvořena analýza příčin. Ta může dopomoci k rychlějšímu odhalení zdroje problému a celkově opravy systému v kratším čase.

Klíčová slova Monitoring síťové infrastruktury, předpověď problémů systému, analýza příčin problémů

Abstract

This thesis covers topics of data prediction and root cause analysis in information system topology. It should cover basic methods which can be used to predict future failures in order to help administrators fix the problems before they occur. Statistical and machine learning methods are used to create analysis which can help administrators fix the problems at their root before they will affect the whole system. With that, the administrator can shorten the time of repairing the information system.

Keywords Monitoring of network infrastructure, predictions of problem in systems, root cause analysis

Contents

List of Figures

List of Tables

Introduction

We are now in the age of information systems. Wherever the user of the Internet can go, there is probably a complex and very robust system involved. There is hardly a person in the first world who is not using the Internet on a daily basis. This generates huge demand for availability of internet's resources. A huge demand for services spreads from e-shops to emails, from `IoT` to `GPS` and the number of internet users is still rising.

The evolution of technology brought us from a small server in the basement to a cloud containing thousands of servers. The home of the information system is getting bigger. This growth also foretells what the needs of users are. We can to this day find *The First Website* [4] of the Web which had only informational meaning. Today we are more concerned with e-commerce, search engines, etc. The biggest Czech search engine provider `Seznam.cz` provides about 15-16 million search requests a day [5].

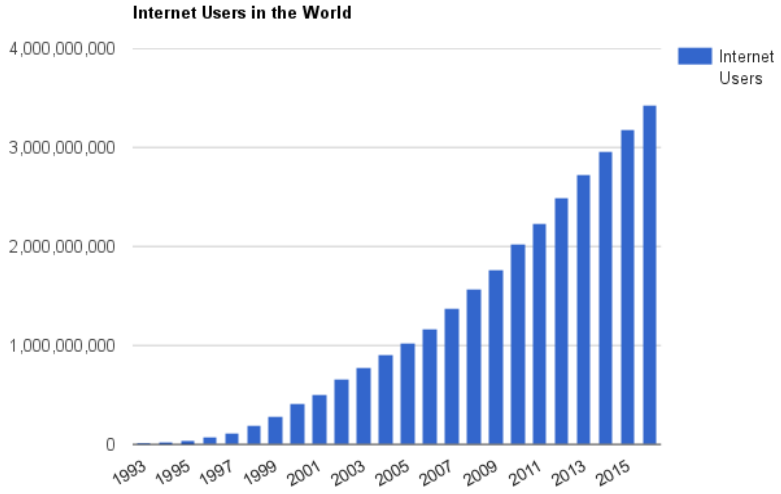


Figure 0.1: Number of internet users [1]

The hunger for services also generates another unpleasant issue. Internet outages have become a global threat. Last year one outage of Google `DNS` in the Central Europe [6] left millions of users without an internet connection. No cyber attack was involved in this outage. From the pieces of information given by Google authorities, it was just a technical issue. An open question how much are internet users dependent on an internet services comes up. The solution to apprehend the outages would be to use monitoring, which was without a doubt in place.

With internet evolving, monitoring grew as well. The basic idea [7, 8, 9] is to provide service designed to check on the other services. If a system goes through unexpected events, a human expert is notified to resolve the situation. This human expert is usually called the administrator. Yet very few of the monitoring systems can provide deeper analysis, yet alone forecast fail state before it occurs. In some cases, based on human expert specified triggers [10], relationships between fail states can be provided, but root cause analysis is not usually available, leaving administrators with their intuition.

Based on these basic ideas we can bring up several questions. Answering them can help create a better monitoring environment. Having a better understanding of upcoming fail states can have a significant impact on today's quality of services.

- What are the current options in monitoring?
- Can any system predict fail state of another system before it occurs?
- What are the relationships between fail states of a system?

Analysis of the current state of monitoring systems is mentioned in the first chapter – *Today's monitoring* (1). The chapter gives a basic overview of monitoring approaches. The main idea is to build a new system based on data provided by services already in use. That's why establishing thorough case study of monitoring services is crucial to the future of network monitoring. Second chapter *Predicting the future* (3) is focused on the time series analysis. Almost every monitored aspect of the information system is a time series (for instance `CPU` idle time or memory usage). Creating forecasts based on time series analysis can lead to recognizing trends or patterns. Having at least a slight idea about upcoming events can help administrator react fast to a future problem. Third chapter *Root cause analysis* (4) targets relationships between fail states. Usually, problems do not occur by their own. Having some time-based link between unusual conditions can help to provide fast return to normal state of a system on their own. Fourth chapter *Testing of application* (5) shows drawbacks and results of a resulting application of this thesis. It should cover situations which happened during testing of the application in production environment.

Chapter 1

Today's monitoring

After an initial research of resources for this topic, it is more than obvious that typology of monitoring is strongly driven by use case. Methods how to monitor state of resources across the whole topology are designed to ease the work of the administrator. That helps with the need for fast deployed solutions which are common in modern information systems[11]. New operations across information system topologies create new roles which profit from monitoring. Information about the system is needed not only for administrators but also for public relations, developers and in case of critical devices even for government institutions.

Definition of monitoring with its goals and much more was described at book *Effective Monitoring and Alerting* [2]. At the very beginning we have to understand two basic objectives:

Availability – stating the time and conditions on which the information system has to be ready to serve requests

Performance – opposite to availability, performance could be described as speed of response to the request, sometimes created from overall load time of a front page (`APDEX`)

Customer with back end administrators create analysis stating basic baselines of a system. Background knowledge of an expert in the field can improve base lining and help discover possible risks [12]. As process continues, both parties establish service level agreement (better known as `SLA`) which goes through availability and performance.

To better understand how these points are implemented, a small case study will be provided in the next section to demonstrate the point. As the typology, definitions and descriptions are part of referenced literature [2], there is no need to analyze it any further. What seems to be important is to investigate current situation of company focused on back end administration. Based on case study further steps can be discovered and applied. After initial study of

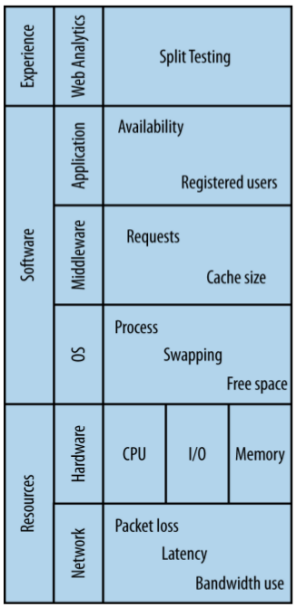


Figure 1.1: Monitoring coverage of system stack [2]

the needs of company, it should be obvious that company needs some kind of thorough analysis in terms of root cause of the problems. That is why we will analyze in section *Existing RCA tools* (1.2) what are current options and what is missing in them for small size projects.

1.1 Case study

1.1.1 Focus of the study

The object of the case study is a company named *Lukapo*. Main target of the company is to provide SLA for customers who own information systems. This also covers assistance with deployment and other developer operations. Everything which the company covers is part of a back end solution.

Main part of the investigation is to discover how company monitors information systems. To assure SLA company needs to supply availability and performance. That creates the need for gathering data on items of each part of the system. Parts can be summed up by system stack of each machine in the topology.

Company *Lukapo* promised to be a subject of this thesis. That is why we could have used them as a subject of the case study and also use their data for later work. Resulting application is also going to be tested in the environment of *Lukapo*.

The company takes care of the parts of stack in different manners. Main coverage is done by monitoring systems without human interaction. In the

following section we can observe how the company deals with items from the system stack in case of problem state (crash, depleting resources, ...).

1.1.2 Methods

The monitoring coverage of the system stack is the main part of this study. Company needs to monitor every part of it. In this case, methodology is going to analyze how the company monitor each part of the stack and how the company reacts in case of failure. As it seems the chosen methodology of this case study is going to be an observation of the process of fail handling. Main points which are going to be observed are:

Part of the stack coverage – discovers what are the tools used to monitor a particular part of the stack

Availability assurance and problem management – analyses how company takes care of problems in the information systems

Problem avoidance – analyses how company deals with future problems based on past experience

By observing these points in each stack part, we can determine what can be improved or what is missing.

1.1.2.1 Resources

Part of the stack coverage: The company covers resources in monitoring software *Zabbix* [7]. The software takes care of basic monitoring. Data from the resources are collected with static sampling period which is usually different for each resource. Based on the inputs from the data, triggers are fired stating abnormal state of the resource. Further notification is done with *OpsGenie* [13] which is responsible for reaching directly to the administrator on call.

Availability assurance and problem management: The company have basic trigger levels of possible problems. For example memory occupancy is divided into multiple levels to create state of warning to complete error state. Availability is handled based on the reached level.

Problem management in case of failure is done by the administrator who is currently on on-call duty. This actually leads the fail management into skills of the human expert. If the knowledge of the administrator is insufficient, chain of other administrators is in place. That means the situation is handled with more and more human experts based on rising time of unresolved emergency. If all fails, customer is informed and a different solution is proposed, based on negotiation between company and customer.

Problem avoidance: Company for low layer resources basically avoids problems with scaling. This is done according to the cost analysis and the information system capabilities. In some cases, only vertical scaling is available since the managed system can't scale across network topology.

Company actually keeps a database of known problems which happened in the past, or problems which are already known and may happen again. This helps the administrator on duty to fix the problem even if he is not familiar with the customer's information system. By creating such a database and improving it, company can avoid future problems.

1.1.2.2 Software

Part of the stack coverage: Company covers software items with *Zabbix* in some cases. The software is covered regularly with checks from outside the company. These checks are coming from third party information systems – for example Pingdom [14] and Statusdroid [15]. Therefore, overall availability is provided with simple pings on application layer. Performance is tracked with the load time of the monitored system.

Availability assurance and problem management: When the whole information system is not running, the `SLA` is usually very strict. Administrator searches for the errors in lower part of the stack in resources. If the fault is not part of the lower stack, programmers are being informed. Availability is also ensured by good functioning resources.

The middleware can be omitted, because company does not have that many customers who have any appliances of the middleware. If so, it is handled in the same manners. Most frequently used middleware is queue messaging (RabbitMQ [16]).

Problem avoidance: The company has an operations manual to fix the issues. If some new situation occurs, administrators with the customers establish baseline operations which should be done in the future. The procedure is almost the same as in resources. Main difference could be that developers are called much often to the rescue. Sometimes application level difficulties are directly part of the application without any possibility to control it outside or from tools of the operation system.

1.1.2.3 Experience

In this case, the company does not take any role. Customer usually collects metrics of classic user behaviour. That is:

- Average time spent on page
- The percentage of returning visitors

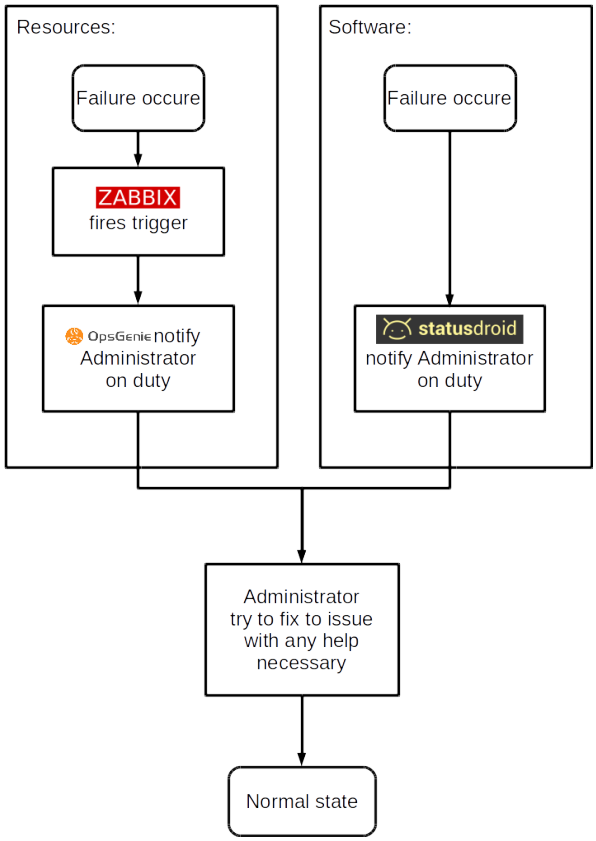


Figure 1.2: Monitoring operations based on part of the system stack

User behaviour is not part of the back end. That is the reason why company does not have direct influence on this stack part. Also, the topic of user experience is not covered in this theses for the same reason.

1.1.3 Discussion

From the description above we can establish basic outline of every failure handling. The company keeps an administrator on duty to solve any possible error which can occur. Each administrator on duty is online on the *OpsGenie* [13]. That means the reaction to the system failure is in a matter of minutes. Time to react in case of emergency is usually specified in `SLA`. The company keeps track of failures in the database and mitigates failures based on past knowledge. In case of new problems and no knowledge how to solve it, administrator contacts developers or directly customer who owns information system.

Based on the operation flow in case of emergency, some missing operations can be found:

There is no prediction in the process of monitoring:

Administrators strongly rely on good constructed trigger. If there is some gradually approaching disaster, administrator will only get the warning in case of immediate threat when the trigger fires. Estimating trends in time series from the monitoring could be essential for the fail avoidance.

There are only explicit triggers:

If any failure occurs and it's not specified as trigger, nothing is going to happen. There is no dynamic threshold estimation in place. This creates uncertainty in levels of failing resources.

Another issue is that there is hardly any monitoring of relationship between failed resources. Administrator can be fixing depleted memory yet the problem could be unwanted steal time of processor in the cloud.

1.1.4 Conclusion

Administrator can get more obvious information before triggers fire on fail states. There is of course some kind of heuristics which provides help with prediction in some cases. For example administrator can predict memory problem in case memory usage steady rises. The problem is that the administrator wouldn't apply his own judgment to every data taken. The company could profit out of more analysis of input data. Correlation between different data of resources can lead to create more non obvious judgment. Also creating some time based relationships can give bigger view on root cause.

There are concepts from applied economics which clearly help with analyzing root cause of a problem [17]. We can explore root-cause analysis with five hows methodology. That could help us create time based relationship view on a problems. This is very time consuming approach and not usable in information system topology. Administrators need to have this analysis fully automatized and see the results in reasonable short time. There are application which offer such a functions and we analyze them in following section.

1.2 Existing `RCA` tools

There are already solutions which create `RCA` in large clouds. One of them is *OpenStack Vitrage* [18] from *Nokia*. This platform is analysed, because there are a lot of similarities with this thesis. Keep in mind that *Vitrage* is growing fast. Conclusion of what could be done differently or what is missing in *Vitrage* and could be implemented can be based on the analysis of past versions.

Another very promissing project is *Dynatrace* [19] which promises full automation of root cause analysis. This platform is available under paid plan as cloud service. Because of its commercial nature, it is really hard to analyze

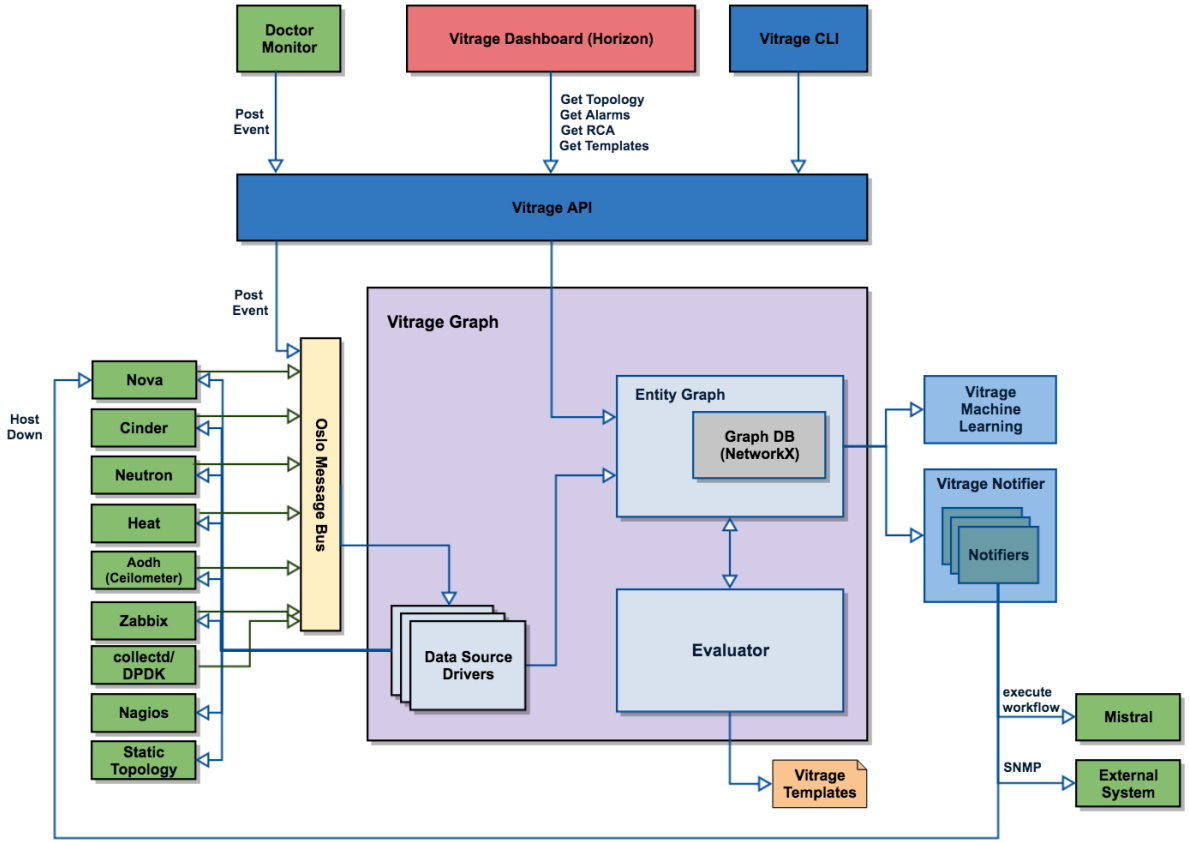


Figure 1.3: Vitrage Architecture

how the methods of root cause analysis function. The company keeps their methods partially secret and calls them *Dynatrace secret sauce* [20].

1.2.1 RCA *OpenStack Vitrage*

1.2.1.1 Description

From the description at *Vitrage* homepage [18] we can see that *Vitrage* performs analysis on the *OpenStack* platform. The whole idea is to expand alerts of cloud stack to give the administrator a better view of what should be the cause. The whole result of the analysis is displayed in the entity graph where the administrator can see what are the relationships between alarms and other *OpenStack* entities. The administrator can open a tab with alarms and see for each what *Vitrage* estimated as the root cause. The map of causes is defined in templates or dynamically generated with machine learning techniques (as they are promised to be ready in the new version). Machine learning results are actually offered to a user as new templates and user should decide if they are useful or not.

Vitrage offers an incredible number of integrations with other monitoring systems. In this thesis we have defined collectors for collecting data, *Vitrage* uses Data Drivers which collect the data and pass them to the Message Bus for later evaluation. During the writing time of this analysis, *Vitrage* has already successfully integrated *CollectD*, *Zabbix*, *Nagios* and others, which are not internal *OpenStack* monitoring resources.

Big advantage of *Vitrage* cloud usage is that *Vitrage* can collect data about the topology. This helps with the estimation of Entity diagram. *Vitrage* uses Data Drivers to contact the correct management system and extract information to estimate topology relationships.

Creation of analysis and graphs is done by Vitrage Graph. From the description [18] we can see that *Vitrage* performs graph searches like `BFS` and `DFS` algorithms. From project update [21] we can see that *Vitrage* also uses `MariaDB` to store some history. In the current available version the functionality is not added yet.

Another part of the system is *Vitrage* Notifier which can create feedback for entities in *OpenStack*. For example, if some host starts to fail in topology, *Vitrage* can decide from the analysis to inform Mistral to evacuate host. For external services *Vitrage* offers integration through `SNMP` protocol.

1.2.1.2 Drawbacks

There are some drawbacks which make the whole system less useful for a company like *Lukapo* (used in the case study). First of all, the company does not have any customer who uses *OpenStack*. As was already mentioned, *Vitrage* has integration for external monitoring systems and even for external feedbacks. The main problem would be the overhead of deploying such a monitoring for lightweight projects which are running on topology of a few servers. In that case, *Vitrage* would be the main consumer of the topology's resources.

In this thesis the resulting application should only use collectors to interact with monitoring. There are no constraints on where the application should run. We can say that the resulting application of this thesis does not have the same ambitions as *Vitrage*, because the target entity of analysis is not a whole cloud but a simple topology of a general customer who is able to use *Zabbix*.

Another problem could be seen in the output of the whole analysis. *Vitrage* uses graphs of entities which can get enormous. The author of the project mentioned multiple times that this is their main concern. After some major collapse, alerts can be generated in thousands. That would make the whole analysis useless during first minutes of disaster until the administrator can judge by themselves what is important and what is not. The root cause analysis is actually available in the `Alarm` tab for each alarm. That means there are thousands of analyses for thousands of alarms.

We would like to tackle this issue with alert aggregations. The resulting application is not going to display all the entities but only the alerts from the monitoring server and cluster them based on their time of occurrence. That would mean the administrator is going to observe a large cluster of multiple alarms in case of a disaster.

Vitrage offers an incredible number of options and configurations which can help improve the whole analysis. The administrator can set up from alarm severity aggregations [22] to whole templates for predefined cause relationships. Now let us consider basic use case of a project with a few servers. The administrator usually spends a lot of time just by setting up the monitoring itself. Creating another configuration to map causality and other relationships could be a matter of weeks. That might be just too much for a paying customer who wanted a few simple servers as backend for his project. Therefore, simpler the configuration can be, the better. Large number of descriptions and settings can be obtained directly from the monitoring servers, there is no need to ask for this information again.

The other problem is that deduced alarms in RCA are a matter of well prepared templates, not a result of some analysis – with the exception of templates prepared from machine learning. This thesis takes a different direction. As we mentioned before, our focus is to estimate prediction, explore clusters or analyze thresholds. None of these methods can give one hundred percent correct analysis, results still need to be evaluated by a human expert.

Vitrage in its current released version does not support history of entity graphs. During the last conference in the *Boston Summit* [21] the authors promised a better solution with sliders. In this work we will simply generate graphs and keep them for reasonably long time. The administrator can find them in the frontend page.

Another very important part of *Vitrage* is a possibility to create feedback. This is not going to be covered by this thesis. Creating feedbacks with large impact to the infrastructure could be an irreversible process. Companies (mainly *Lukapo*) which agree to provide data for this thesis did not give any permission to automatically alter infrastructure. It would be safe to say that target users do not need this function from the RCA analysis.

The last point to end this comparison: Vitrage does not support any predictions for data items. So far it does not even have its own time series database to keep the data. The author did not mention if they are going to add this function to the *Vitrage*. It is safe to assume that they are not going to.

1.2.2 RCA *Dynatrace*

1.2.2.1 Description

Very interesting platform is *Dynatrace* which is cloud application capable of analyzing the whole application stack. The solution is designed to be used by large enterprises. On the *Dynatrace* webpage in Customer stories we can see names like Volkswagen or Whirlpool and there are many more. *Dynatrace* promises much more than root cause analysis – it is connected to economical tools. For example root cause directly counts revenue loses in case of problem detection. This is promised to be done mainly without human interaction and mainly via artificial intelligence (AI).

Data extraction is done by one of the patented solutions called OneAgent. It is the application which is installed in the monitored hosts and collects as much information as possible. From further analysis, it mainly uses good knowledge of the application stack. By the description, OneAgent is able to dynamically find relationships of services called by the target application. Those are the lowest blocks of the cause analysis. *Dynatrace* can later estimate the problems and directly point to the services deduced from the relationships found by OneAgent. In practice, *Dynatrace* can explain system collapse from first click of the user to the failed database query.

Dynatrace also estimates entities diagram in dynamic infographic 1.4. Description promises estimation in a matter of seconds. Close inspection again reveals that this is done by a patented solution Smartscape. The solution promises no gaps and blind spots. The user has one hundred end-to-end visibility into all the application components. This is very impressive if we consider what can be expected form *Vitrage*.

Dynatrace states that their root cause analysis is one hundred percent accurate in case of dependency detection (example in 1.5 figure). They also state that they estimate this within seconds. Reason seems to be very robust OneAgent which auto-inject every part of application stack with it's own data collectors. Data collected from this process are needed for thorough later analysis. Another addition captures everything on the user experience level. This analysis could be provided to the users with heat maps. With that, *Dynatrace* can estimate the impact of a problem and count loses in revenue.

Another great feature of OneAgent is the automatic baselining. We used baselines in definition of SLA. Furthermore, *Dynatrace* uses it as dynamic thresholds and can deduce some alarms if the baseline is reached. From the description on the application page, smart baselining can remove almost ninety percents of false alarms.

It is more than obvious that without practical usage and exploration of *Dynatrace* we can hardly understand how the whole application functions. Even with the description from the website it is hard to deduce what can be expected form *Dynatrace* an how does it work. Since there is a large number

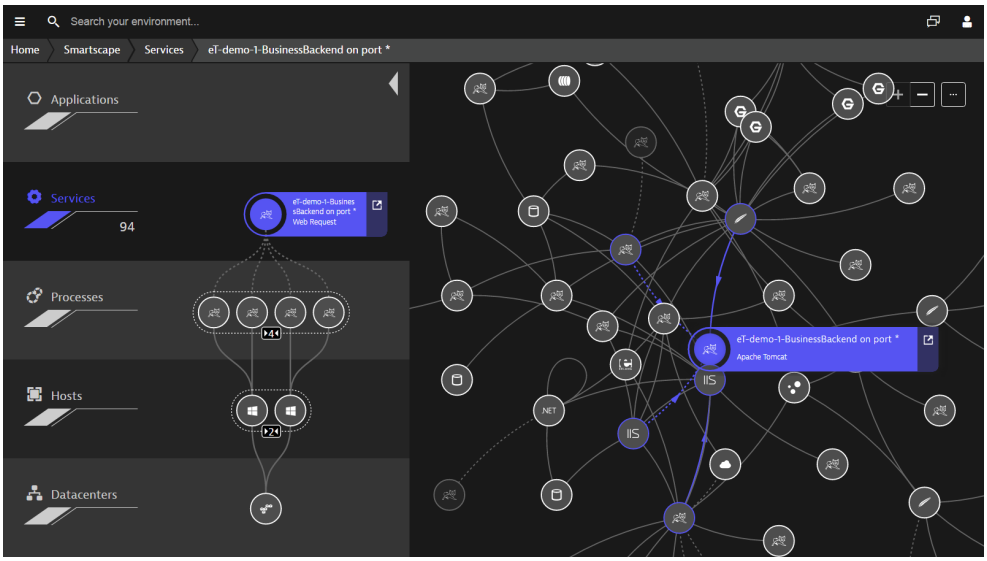


Figure 1.4: Dynamic entity diagram of *Dynatrace*

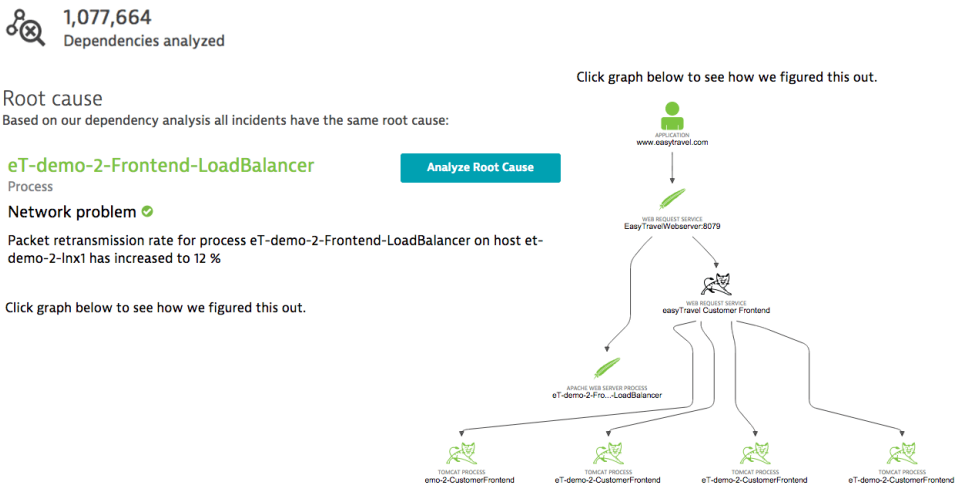


Figure 1.5: *Dynatrace* root cause

of satisfied customers, there can be hardly any criticism. Only drawbacks we could find are based on problems of pricing and close source.

1.2.2.2 Drawbacks

The biggest problem with an otherwise amazing solution is the price. *Dynatrace* offers fifteen days trial which is free. The pricing plan differs in size of enterprise. If customer chooses to pay a plan named `Pay-as-you-go`, the charge will be counted from the size of memory and hours of OneAgent runtime. For example, small server with 4 GB of memory would pay 8.4 US dollars a day. This is just for monitoring of performance, code-level visibility, deep process monitoring and cloud infrastructure monitoring. Other methods like user experience are paid extra. If we consider that month has thirty days, we will get roughly about 252 US dollars fee, that could be just too much for a small project.

We also have to consider that *Dynatrace* is a closed source. Using non open source platform can be unreasonable for some customers. The ability and assurance that everybody can check for programmer's errors could be helpful in designing new projects.

We also have to consider data transfers. Trusting a third party in case of monitoring means large amount of data traveling to the cloud for analysis. Nowadays, this behavior is not unusual. We have already mentioned services like OpsGenie and Newrelic which act exactly the same.

We have mentioned that *Dynatrace* is a closed source. There is no way how to install the platform on customer hardware or cloud. That is very obvious since *Dynatrace* is a private company which doesn't need to harm its income. The customer in this case needs to trust the company with every other aspect of running such a platform (backups, availability, performance, ...).

1.2.3 Comparison

To compare analyzed platforms, a summary table was created (figure 1.1). The application which should be the result of this thesis should cover the needs of target company, that's why can use the basic ideas in here in terms of *Our proposed solution*. The attributes composing the table should cover the main differences and better display the ambitions this work has. Attributes compared in the table were deduced from the case study. The main focus was given to these points:

Integration – attribute which should help the target company to better deploy the resulting application into the currently used topology of servers under monitoring

Methods of analysis – explores several possible methods used in analysis – finding them can help analyze possible data dependencies and help with application deployment

Predictions – analyze if the application has capabilities in prediction of data series

Entity diagram – analyze if the application can create network design to better display problems in topology

Housing – explores options of platforms which can be use to run the application

Adding our proposed solution helps with creating (function and non-functional) requirements in later application design.

The application resulting from this thesis faces several obvious challenges, which, however, cannot be tackled by the application. We do not have cloud platform to use for large scale computation as *Dynatrace* has. The target company is driven by the customer budget. Lease of cloud platform used only for advanced monitoring can strongly influence costs of the whole solution.

Also, since only the data used for analysis come from monitoring servers, the application can't have any information about topology. That degrades informational value of any entity diagram to entities from monitoring. This is a constraint given by not using any other monitoring agent.

On the other hand, we plan to have usable predictions of failures. Whole result application is open source and it is not bound to any cloud platform. This can improve deployment of the solution.

	Vitrage	*Dynatrace*	Our proposed solution
Integration	Integrated with all main monitoring systems.	Have its own DataAgent which collects variety of information based on a paid plan (differs for full stack and cloud infrastructure).	Plan to have integration to monitoring servers.
Methods of analysis	Have explicit templates which define relationships in the root cause analysis. Machine learning techniques are partially used.	Contains `AI` methods, the company has patented their solutions and calls them *Secret Sauce*.	Plan to use only statistical methods to estimate clusters, relationships, thresholds and correlations.
Predictions	Does not have any ambition in predictions.	It is hard to find out if *Dynatrace* uses any prediction.	Predictions are one of the main functions.
Entity diagram	Estimate entity diagram from topology and resources.	Estimate infographics which have the same meaning as entity diagram in *Vitrage*.	Do not estimate any entity diagram, because monitoring system does not keep any information about it (only source of information is the monitoring system) and there are no other systems to obtain it from.
Housing	Could be installed on local machine.	Provide frontend to the results which come from cloud computation.	Could be installed on local machine.

Table 1.1: Comparision of different root cause analysis implementation

Chapter 2

Application design

Based on the discovered topics we need to estimate what is going to be result of this thesis. That should be an application containing the tools covering the topics which were mentioned as insufficient in case study.

We have already covered some requirements, which can help administrators monitor topology better. From the observation provided we can estimate basic functional requirements. Those should cover the needs of the administrator for creating better judgment and analyze monitored topology better:

- Application has to provide the administrator with basic prediction of future
- Application has to create root cause analysis

From functional requirements we can create some non-functional requirements which are actually quite natural for monitoring systems. These should also be covered by simple `SLA` offering reasonable availability and performance. Non-functional requirements should support functionality of application. We can estimate them as follow:

- The application has to provide analysis in a reasonable, understandable frontend
- The application has to provide prediction in a reasonably short computational time
- The application has to create root cause analysis in a reasonably short computational time
- The application has to collect data from monitoring servers and keep them for analysis
- The application has to use data from existing monitoring applications

The first three points of non-functional requirements come from a practical point of view. Administrator can not profit from analyses which were estimated late. Point mentioning data collection is created for the needs of data analysis. Monitoring servers have different procedures of keeping data. These procedures can keep just averages or compressed data from the past – these data are then deemed useless in further analysis. Last point is very significant for the simplicity of deployment of the resulting application. The administrator can "plug in" whole application to the running environment without any additional agent.

Application layout needs to be divided into multiple subsystems which can communicate with each other. The whole system should be able to asynchronously work on predictions as well as collect data from monitoring servers or estimate root cause analysis. From these requirements we can estimate the components of the application:

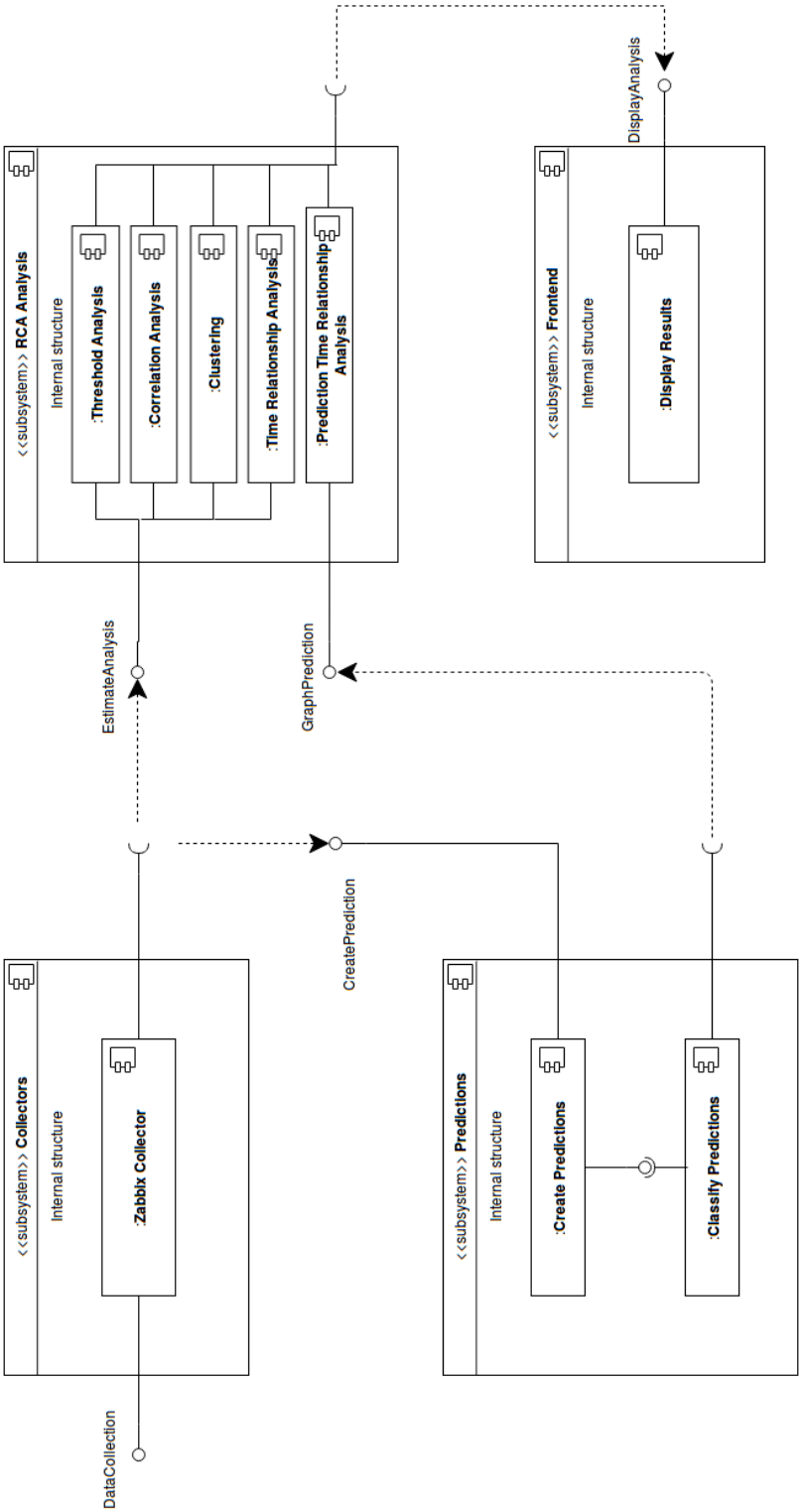


Figure 2.1: Components diagram

Figure 2.1 should describe subsystems and components which are independent in terms of computation. The system is divided into multiple subsystems performing different tasks. Results are presented to the users through frontend. The following section goes through every subsystem, describes its relationship in terms of other subsystems and analyzes possible algorithms which can be used for inner functionality of subsystems.

Collectors

Subsystem **Collectors** provides data to the subsystems **Predictions** and **RCA Analysis**. Input of data is provided by independent monitoring servers. It should serve as an independent layer between monitoring such as *Zabbix* or *Icinga* to the monitoring analysis.

Predictions

Subsystem **Predictions** gets data from **Collectors** and estimates the predictions. These predictions are then passed on the **RCA Analysis**.

In exploration of resources covering predictions we have found algorithms such as `ARIMA` and `LSTM` networks. Those are analyzed and compared as tools which can be used in the resulting application.

RCA Analysis

Subsystem **RCA Analysis** analyzes different aspects which can be useful for the administrator. Data inputs are from **Predictions** and **Collectors**, the output is passed to **Frontend** which displays results to the user.

In case of analyzing data for `RCA` which come only from monitoring servers, we have to consider what is their informational value. The amount of data needs to be reduced and put into context. There are some algorithms which can be found in literature [2] and some which can help compare data series. Without any entity diagrams exploring physical relationship, we can always map relationship between triggered alerts in terms of time.

Algorithms which can help with `RCA` could be following:

Time based relationships – alarms do occur in time order, analysis causality can help the administrator decide what was the cause of a problem. We are going to explore if different methods can help analyze causality better. For example, using logical time instead of real time can be promising, because it helps to sort every event in the distributed computing environment.

Clustering – in a disastrous scenario, alarms occur in large quantities. We can create clusters of them based on the time at which they occur. As we mentioned before, we can also consider logical time. Clusters can be estimated from density in time. Very simple clustering can be done by

sampling the time. For example, if we would sample every minute, our clusters would contain alarms which happened during that minute.

Thresholding – estimating constants which create a frame creating baseline of data. If data goes above or lower than threshold estimated from reasonable old data, it could be considered a problem.

Correlation – to compare two time series we would need to have a measure of distance. Correlation in form of the Pierson Coefficient seems to be very useful to capture such a distance.

Further analysis of implementation is presented in each chapter. Partitioning all the algorithms and methods to the separated subsystems proved to be a manageable way of implementation. In terms of components, we can imagine that each component is a microservice bound to other microservice with queues.

2.1 Prototype

For better representation of the result, a mock-up of frontend was created (figure 2.2). From the component point of view, this should be the output of subsystem **Frontend** if all the other components in the system work correctly. We can see that `serverXY1` failed due to a lack of free space on device. Another alarm was triggered by service `MySQL` which stopped working properly. Time related dependency was formed, stating that free space on the hard drive alarm was triggered before the `MySQL` alarm. The administrator can keep track of the alarms and consider time dependency to solve the issue. Another very significant analysis was provided via correlation estimation. We can see that time series of service `Free space on device` on server `serverXY1` and service `File backup` were significantly correlated. The administrator can decide to enlarge disk space on both devices. Based on prediction, analysis created an alarm on service `Website load time` which reached unbearable limits and was classified as prone to failure in near future. Another failing service could be `Number of web connection` which reached dynamic threshold of 95 percentil. Administrator should take it into consideration because the load time of webpage is rising and was predicted to fail (based on trigger).

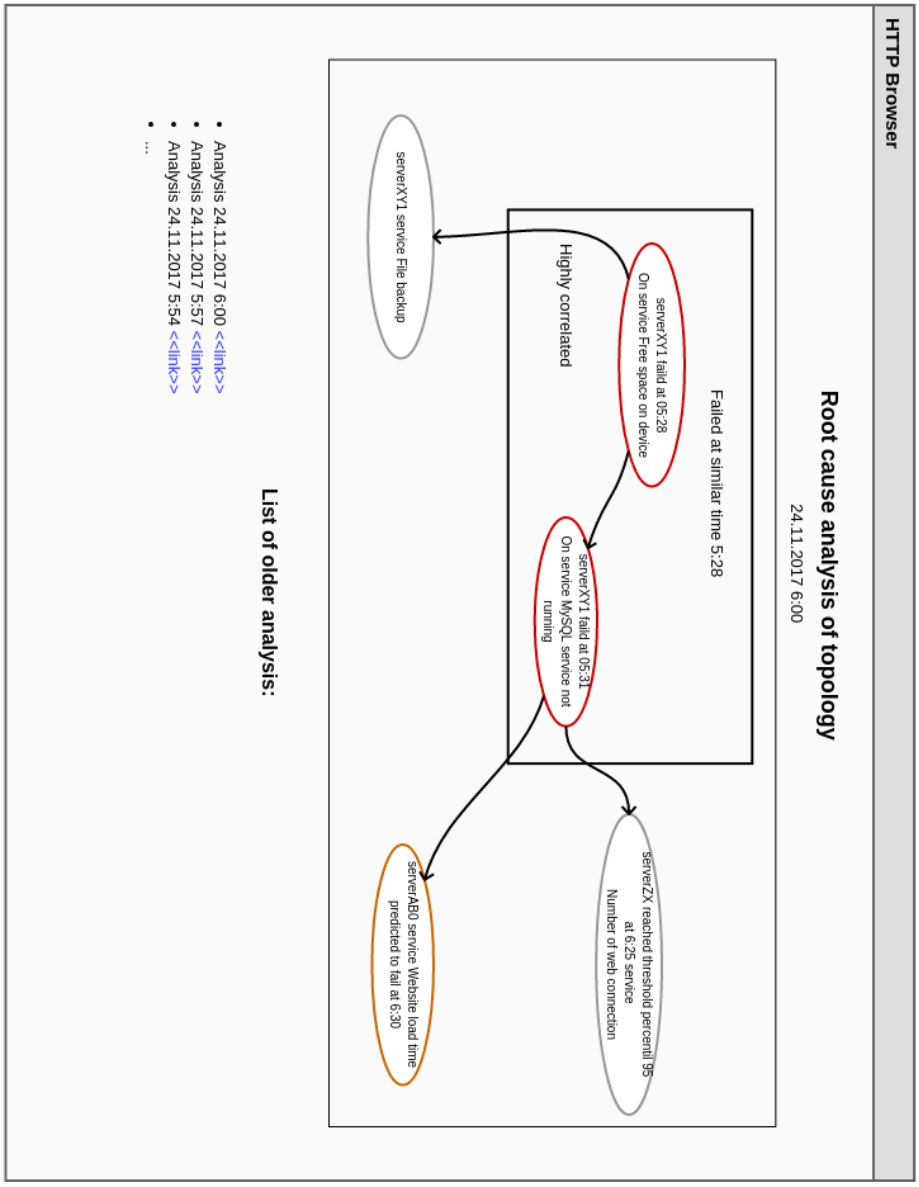


Figure 2.2: Mock-up of application frontend

Application design was used to show what should be the result of this work. We have defined the questions which should be answered in the introduction. Creating this application should lead to a better understanding of what methods can be used and how they should be implemented. The following text will go through all the subsystems and describe how they were implemented. We can also find created known solutions. There are already applications which create root cause analysis. On the other hand, non of them create predictions.

Chapter 3

Predicting the future

Nowadays there is not a single system with some commercial value that wouldn't be monitored – there is always some health check. Beyond simple ping, control covers many other resources. For example `CPU` load, memory usage or disk `I/O` operations. Some of the monitoring systems also store data to create graphs. Based on the data, further analysis can be done. The majority of monitoring services (*Icinga*, *Cacti*, ...) do not provide data forecasting. *Zabbix* can use module [7] which can give basic models to forecast the future based on past observations (linear, polynomial and exponential regression models). None of the monitoring systems mentioned earlier use complex models like `ARIMA` or Neural Networks with Long Short Term Memory (`LSTM` Neural Networks) – the reason is long computational time.

Furthermore, there are still some attributes of forecasting that need to be satisfied. If forecast takes unreasonably long time, future can be far gone, when data are ready. Another problem may be selecting which data should be analyzed. Monitoring systems cover almost every resource of the client server ranging from incoming `HTTP` requests till hard drive `I/O` count. If a forecasting system needs to cover every input data time series, there can be hardly any assumption about data. Data could vary in trends, mean, maximum, minimum and various other features. Without any expert knowledge, forecast needs to be as general as possible.

Forecast itself can help avoid outages of systems. With fitting statistical or machine learning models for crucial resources, system administrator can know in advance what are the possible future problems. For example, when an incoming bandwidth of data to server exceeds certain value, a server can become unreachable. Administrator can be informed about upcoming alert from our application and try to fix the problem before it happens.

Collection data section of this paper is about data retrieval from monitoring systems. There is no standard representation of the data, therefore implementation of a collector which can do some transformation was created. For the purpose of showing how difficult forecast can be, one particular time

series was selected. It has all the unwanted properties which could be found (no stationarity and a lot a peaks). Second section *ARIMA* applies `ARIMA` model on one selected time series. Further analysis is done, especially on deciding the correct order of the model. Third section *Reccurent Neural Long Short Term Memory Network* deals with the machine learning methods for forecasting. Testing is done primarily for the long term predictions, because the longer prediction, the better. Exception is `LSTM` network, because of different use case. `ARIMA` needs to be retrained every time with new data where `LSTM` can adapt and be used reasonably fast again and again. Therefore in *Practical use of `LSTM` networks* we focus on using `LSTM` for short term predictions.

3.1 Collecting data

Data from the monitoring systems needs to be accessed via a fast and reliable way. Creating models from old data could lead to an invalid conclusion. For some systems, access to a database is performed directly via `REST API`. All the monitoring used in this paper has some `REST API`. The problem is with systems which do not by default keep any memory and only work with the current state of resources. That is, for example, `Icinga` (with basic installation) which asks client machines about their states. With collected output comes logic. It decides on contacting the administrator to change any unwanted state. `Icinga` uses `Nagios` commands on client devices. On the other hand *Zabbix* keeps data about the past and uses moving average to compress data. Moving average can lose some features of time series. Both of these approaches are not usable for further analyses.

Therefore, new layer was suggested, its main role is to collect data. That is why it uses collectors. These collectors are different for each monitoring system. The reason behind that is absence of normalization over `REST` interfaces. Usually, there is no Atom Syndication Format or `HATEOS` which can help drive collector and make in more reusable. Each of the collectors is going to obtain data from monitoring software's `REST API` making a layer between data used for time series analysis and data in different formats. Collectors are going to save the data to a shared database (figure 3.1).

Sampling speed of data is strictly defined by monitoring system. Collector does need to make samples in unified time, because data items could be still pulled out from monitored systems. Therefore collector needs to find data item collection interval and create a schedule to retrieve it only in case monitoring system already got the data. Implementation in *Python* gives a lot of opportunities to create such a polling task. The problem was that a lot of these solutions used process for each task given.

That lead to server collapse with incredible number of processes. That's why collector uses library `apscheduler` [23] which provides multiple threads

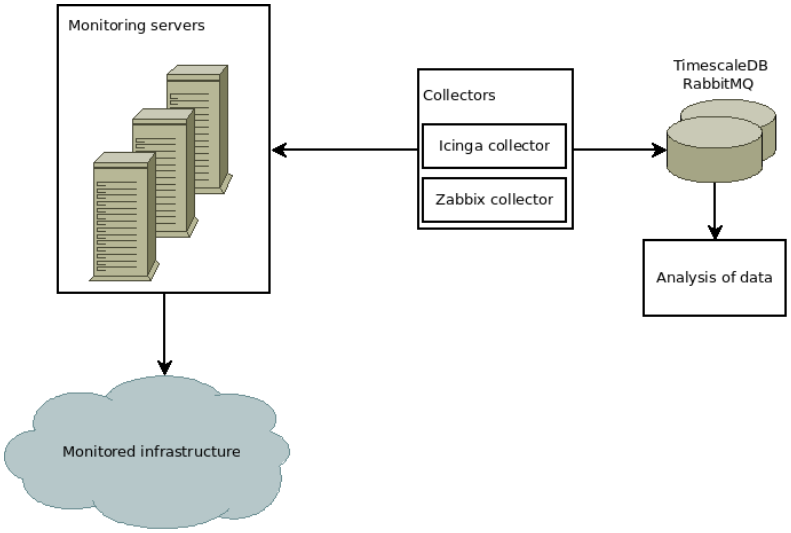


Figure 3.1: Data collection infrastructure

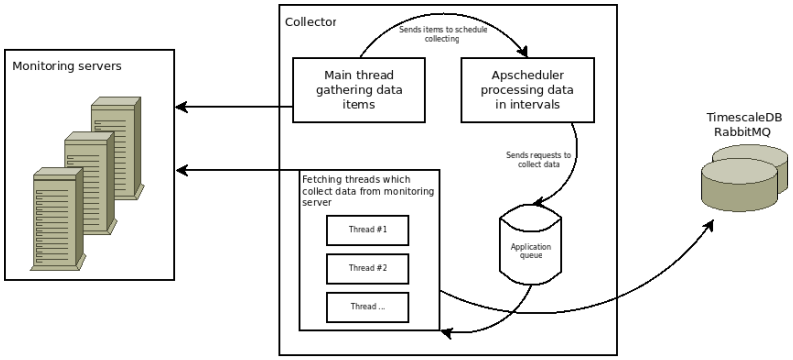


Figure 3.2: Inner architecture of collector

that work on jobs given in code. After collecting roughly about 2000 data items per 30 seconds, `apscheduler` created exception stating that execution time was too long and data were not collected in accurate time. This was apprehend with rewriting scheduled jobs (figure 3.2). Currently job gives only notice to queue that there are data to be collected and on the other side of queue consumers collect the data whenever they are ready. Data are than send to `RabbitMQ` queue for other jobs (analysis, prediction, ...).

The database has a pretty simple layout, there is no need for complicated relationships. Table for records contains only five columns, only the first normal form is enforced. Following list explains the columns:

id – is running iterator for later use in `ORM`, there is no need for it in any analysis

time_execution – contains Unix timestamp which is used in *Zabbix* and also in `Icinga` to cover time when the value was obtained

resource – contains the name of the monitored resource, for example `CPU load` or memory usage

hostname – of the machine which is monitored

result – is a value of **resource** taken at the **time_execution** from machine with given **hostname**

state – is a status of **resource**, for example when monitoring suggests that resource is in `warning` state, non zero value is stored

Columns **hostname** and **resource** are already named in the monitoring system. Collector only retrieves their names and saves them to records table. There is no need for user explicit knowledge.

Column **hostname** and **resource** can use bitmap index to speed up querying. Both columns have low cardinality. `PostgreSQL` does not offer explicit definition of bitmap index, it uses it in querying system. **Time_execution** has significant meaning in `TimescaleDB` [3]. The database can be partitioned not by the size of a table but by timestamp. This partitioning is called chunking. Fast access to the date is provided by a hypertable, which has an overview of all the chunks in the table. `TimescaleDB` is based on `PostgreSQL`. Developers can use normal `SQL` to obtain data. Much needed difference in speed can be seen in benchmarks provided from Timescale (figure 3.3).

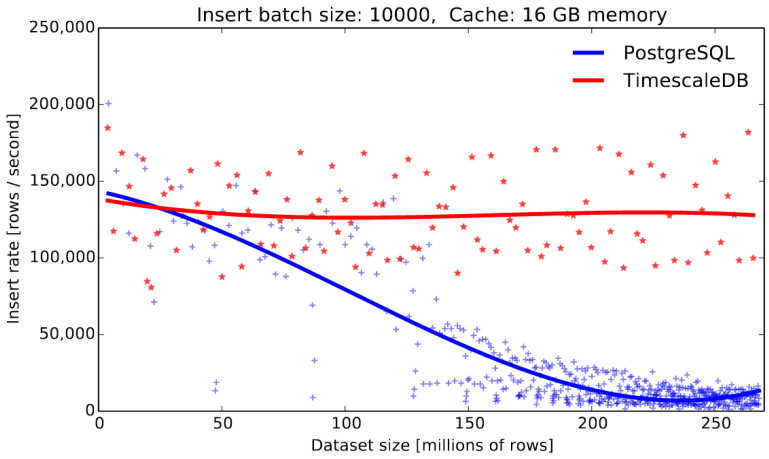


Figure 3.3: `TimescaleDB` benchmark on 16 GB of input data [3]

Furthermore, with collection of large set of time series comes a question of which of them are necessary to analyze. For example, collecting number

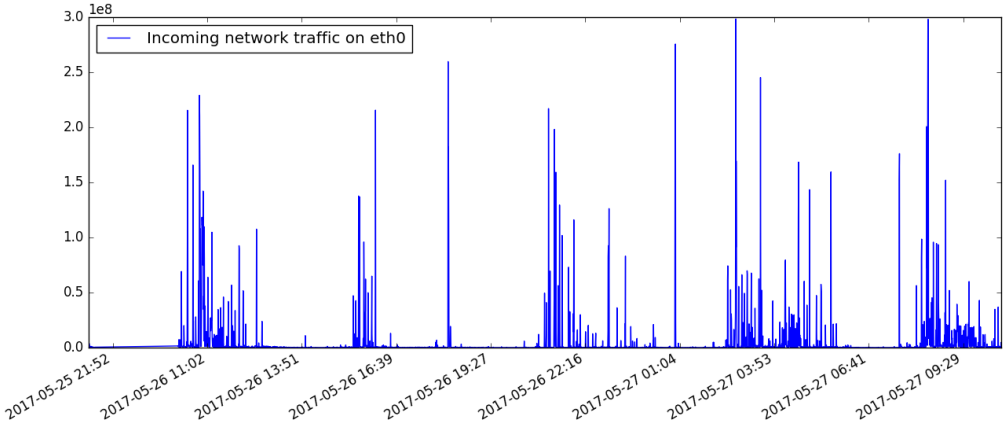


Figure 3.4: Incoming network traffic of ethernet device of message queue server in bits per second

of logged in users has much smaller meaning than counting incoming packets. Basic preprocessing should mark time series which do not give any information. The first idea would be to use entropy. But this is usually not the case since time series data are not discrete. Using a continuous version of counting entropy, which is called *Limiting density of discrete point* [24], is not reasonable either. Data do not need to follow any continuous function.

As a result, some other features need to be used. One of the fastest ways seems to simply check standard deviation. If the deviation is zero, there is no need to use the series in forecasts. Time series model, in this case, should be constant for all its values through the time. The reasonable forecast would be just to prolong constant values. Another attribute in selecting right time series to analyze would be to ask a human expert to mark what he is interested in specifically.

For the purpose of this paper's section, only data from one resource are used. It is the incoming network traffic on ethernet device of message queue server (figure 3.4). Data are taken from *Zabbix* which saved data of resource every minute for five days. There is a space in data at the beginning, meaning that the state of the resource hasn't changed in some time. No data were received during that time.

Zabbix collected 3780 pieces of data with the standard deviation of 22674710 and mean of 5010694. Data spans from `2017-05-25 21:52:21` to `2017-05-30 17:49:38`. Minimum value is 30776 (30 [Kbits]) with maximum 298492700 (284 [Mbps]). Simple look at the data shows that hardly any assumption can be made. There are a lot of peaks and no visible stationarity by far.

3.2 ARIMA

`ARIMA` is an acronym for *Autoregressive Integrated Moving Average* model. It got its fame from its application in predicting stock market [25]. There are two reasons why to use `ARIMA`. First is to understand better underlying signal of data (if there is any) and second to create a forecast based on already known data. As the acronym suggests, it is constructed from a combination of three base models.

3.2.1 AR

Autoregressive part implies that there is some autoregression in the signal which could be used for further forecast. It's defined as follows:

$$x_t = \alpha_1 X_{t-1} + \cdots + \alpha_p X_{t-p} + z_t \tag{3.1}$$

Where α are autocorrelation coefficients at lags $1, 2, \ldots, p$ and z_t is residual error term. A good model covers all the data and leaves $z_t = 0$. It is trying to fit data with a function using previous x. The function is adapted via optimizing coefficients to fit input data with the smallest error. In this case the residual sum of squares divided by the number of degrees of freedom was used (as it is usually default option). Input parameter of p states what order `AR` model is.

3.2.2 MA

Moving average can help with univariate time series. Furthermore smooth data, which have peaks with no informational value (for example current load of `CPU`). It is defined below:

$$x_t = \sum_{i=0}^{q} \beta_i x_{t-i} \tag{3.2}$$

Where β_i are the weights applied to prior data. The β coefficients cover weighted average from current and immediate values. Moving average targets trend of immediate data to create a forecast.

3.2.3 Differencing

Both `AR` and `MA` models have a problem with stationarity. If the model is not stationary, there is hardly any way how to fit these models (models are finite). Time series have to keep variance, autocorrelation and mean constant over time. This suggests removal of trends in data, therefore integration was introduced to make dataset stationary.

For our input dataset, we can create a test telling us something more about stationarity. *Augmented Dickey–Fuller test* (named after David Dickey and

Wayne Fuller [26]) has null hypothesis whether a unit root is presented in an autoregressive model or not. The alternative hypothesis states stationarity. We can run the algorithm on the dataset:

```
ADF Statistic: -4.297952
p-value: 0.000448
        5%: -2.862
        1%: -3.432
        10%: -2.567
```

For rejection of null hypothesis, the p-value is less than or equal to a specific level. P-value 0.000448 is much bigger than any level of 5%, 1% or 10%. Hypothesis couldn't be rejected. Dataset could not be labeled as stationary, therefore differencing is going to be used by `ARIMA` model. Whole model can be described as follows:

$$(1 - \phi_1 B - \ldots - \phi_p B^p) \cdot (1 - B)^d y_t = c + (1 + \theta_1 B + \cdots + \theta_q B^q) e_t \quad (3.3)$$

The first part of the equation is dedicated to `AR` with input degree of p. The second part is for differencing with the order of d. Last part in `MA` model is stated with the polynomial of degree q.

3.2.4 Deciding the parameters

As mentioned earlier, input parameters of p, d, q need to be given to `ARIMA` model. This could be achieved by observing autocorrelation function plot (figure 3.5). By carefully analyzing plots generated by statistical library we can derive optimal values.

Autocorrelation plot shows that the data stick at lag 0 to 1. There is a small hint of seasonality which occurs at peaks of autocorrelation in the frequency of 5 lags. From `QQ` plot we can summarize that series is a skewed normal distribution [27].

The problem is that this approach couldn't be reproduced automatically. The algorithm can check for autocorrelation just as well as an observer of a graph did. On the other hand, it couldn't notice trends suggesting some level of difference. There is a way to estimate the parameters through a full scan automatically. Since the model needs to be generated with an application, scan for parameters needs to be done anyway. To compare which parameters are better or worse than others we can use *Akaike Information Criterion* (`AIC`) [28].

`AIC` can help with comparisons of models. Suppose we have M of data x. Let k be a count of estimated parameters in the model, then $\hat{L}$ is the maximized value of the likelihood function for the model. That means $\hat{L} =$

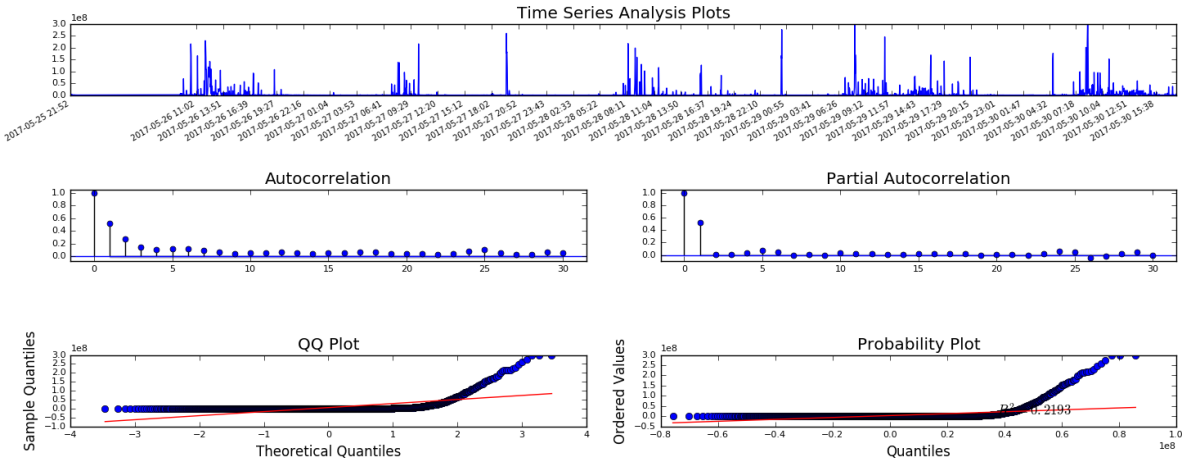


Figure 3.5: Input data statistics

$P(x|\hat{\theta}, M)$, where $\hat{\theta}$ are parameter values that maximize the likelihood of a function.

$$AIC = 2k - 2\ln(\hat{L}) \tag{3.4}$$

Let's say we have two models describing the same data with different parameter values. We want to decide which one did the description better. We perform count for both `AIC` and compare them. If `AIC` of the second model is smaller then for the first model, the second model is better for describing the underlying structure of data.

Since we already know autocorrelation function of our data, we can assume that there is going to be a really small size of parameters p, q. Full scan with `AIC` for the parameters estimated the best (p, d, q) as following $(1, 1, 7)$. Resulting `AIC` was 109940.967535.

3.2.5 Testing the model

Consider model parameters $(1, 1, 7)$. Let's do some preprocessing and divide data into testing and training. A ratio was set to 80% of data are used for the training phase, and 20% are left for testing. After that we can estimate how much different will the mode be in the future. To calculate error, we will use mean squared error function [29].

The figure shows (figure 3.6) by how much the model missed the real data in the long run. The graph also shows the underlying structure which the model established on input data before creating the forecast. This approach could help in further analysis. The resulting mean squared error was estimated to 25080375.5115. This shows a relationship with autocorrelation which

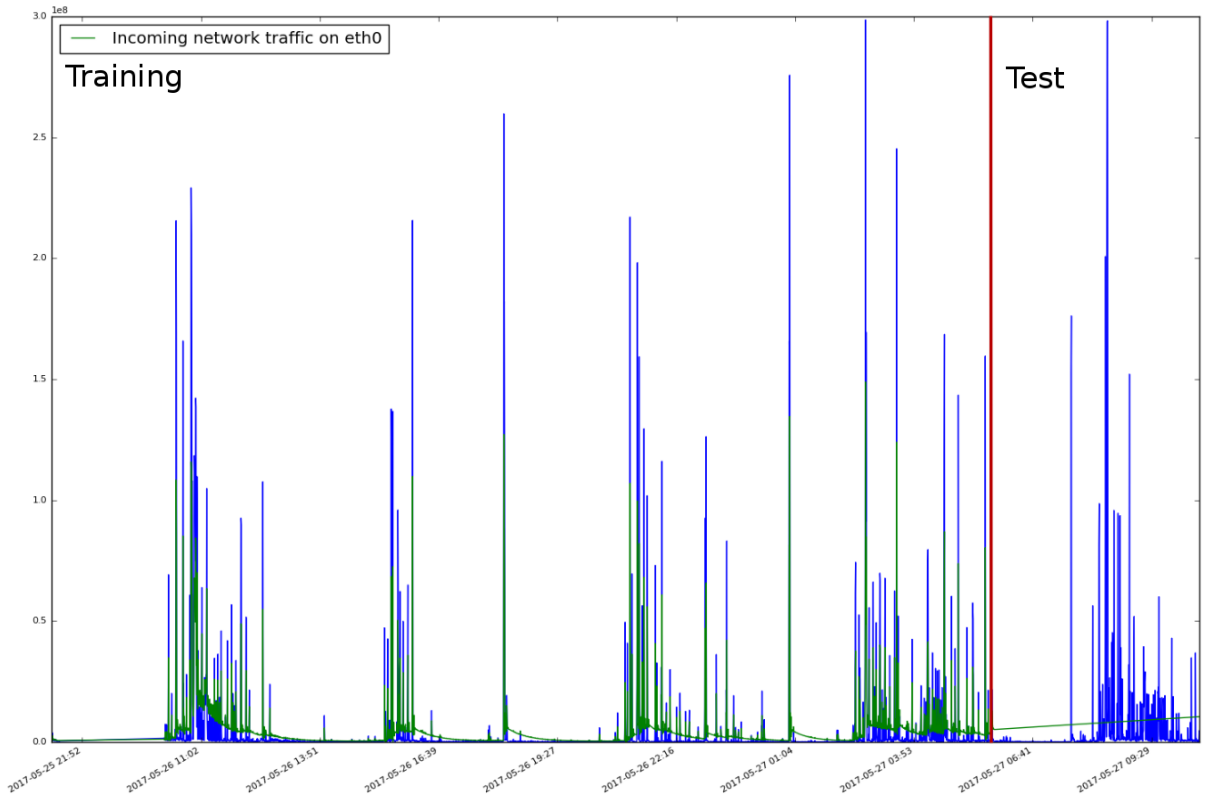


Figure 3.6: Real data in comparison with `ARIMA` model, green line is the prediction and blue line is showing real data

suggested that long prediction in the future couldn't be successful with this data.

3.3 Reccurent Neural Long Short Term Memory Network

`ARIMA` was introduced based on statistical features. What if there is a structure which could be learned and later used in the forecast? This learning could be done without knowledge about features like autocorrelation. Therefore some analysis regarding Recurrent Neural Networks needs to be done.

For the forecast of future values, we certainly need to understand the past. Standard neural networks do not provide any advantage in this situation. Their form of learning is not based upon creating persistent knowledge used for further forecasts. Recurrent networks address the issue. They use loops in form of a feedback allowing information to stay in more persistent mode.

In diagram (figure 3.7) the input x_t is coming in time t with output h_t

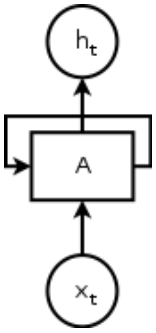


Figure 3.7: One node of reccurent network

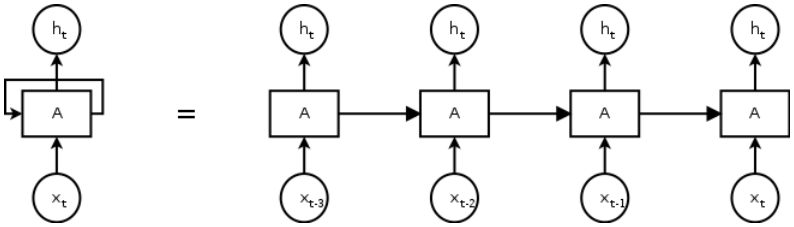


Figure 3.8: Unrolled reccurent network

(hidden state) which is later used as recurrent input in other iterations (figure 3.8). The network can be unrolled and can pass its outputs at time $t-1$ to another unrolled part at time t. Each loop can have old hidden states as well as new data on the input. This approach can lead to providing relationships in the network.

So far we haven't mentioned the issue of activation function, which is only one in the `RNN` node (figure 3.9). This leads to a problem of long-term dependency. It seems there is only explicit dependency from the last state to new one. That doesn't need to be true as we also observed at `ARIMA`. `RNN` can benefit from creating not only relationship between outputs directly from the last step but from more past steps which do not need to be in sequence. This creates a context which can be used in the network. There is also the limitation on how much past lags can create the context for new output [30]. The answer to this problem can be solved with `LSTM` [31]

Long Short Term Memory Networks are a specific kind kind of `RNN` with the ability to learn long-term dependencies. Instead of using one activation function, there are five of them, each has a different meaning (figure 3.10).

The node (figure 3.11) contains cell state in horizontal line with one multiplication and addition. The state is the way how the network transfers information with some minor linear interactions. The ability to remove or add information is modelled with gates. Each contains one sigmoid and a pointwise multiplication operation. The output of gates is between one and zero expressing the weight of how much information is relevant in the context

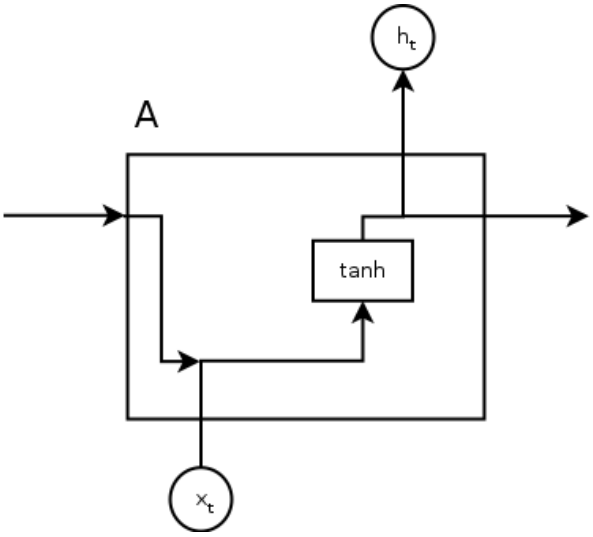


Figure 3.9: Activation function in one node

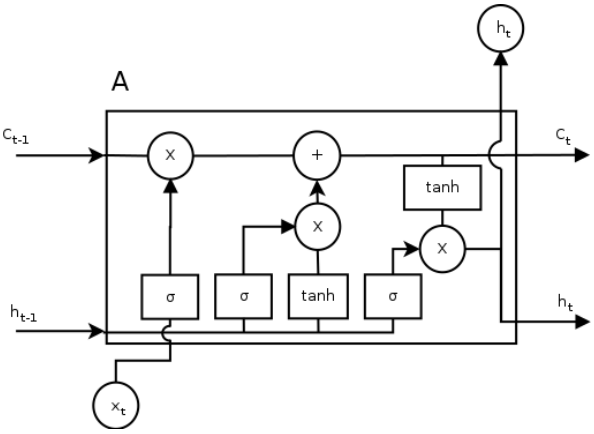


Figure 3.10: Activation function in one node of LSTM

of the network. There are only three gates to protect or control the cell state.

First step of `LSTM` (figure 3.11) is dedicated to what information should be left out. The sigmoid gate responsible for this is called "forget gate". Output is between one and zero where zero discards whole information and one keeps it all. Output of the gate is given by function: $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$.

Next step (figure 3.12) is to decide which information should be stored in cell state. First sigmoid is called "input gate layer" and decides which values should be updated. Next the tanh function creates vector of a new candidate for C_t. The output function is stated as follows: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ and $C_{tc} = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$.

Update of C_t is created in the last step into the cell state. It's formed

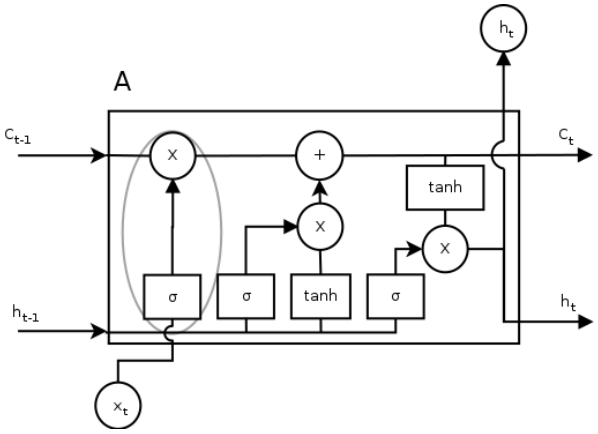


Figure 3.11: First step of `LSTM`

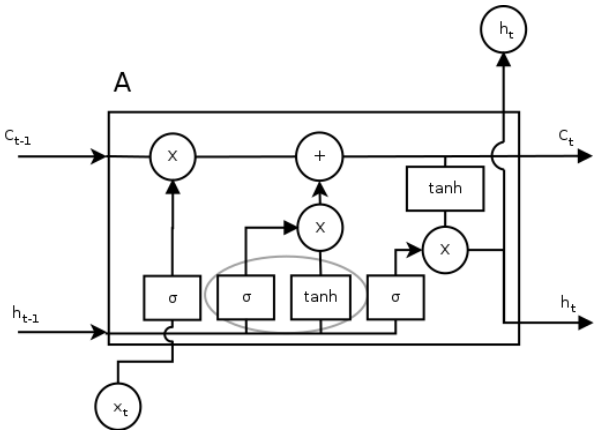


Figure 3.12: Second step of `LSTM`

out of all the gates from the steps before. New C_t is created from the next equation:

$$C_t = f_t * C_{t-1} + i_t * C_{tc} \tag{3.5}$$

Finally, the decision needs to be made about the output. It is based on the state of the cell. Also filtering is applied. Sigmoid layer decides which part of the cell is going to be outputted. Then the tanh function pushes the value from -1 to 1 and multiplies it with an output of the sigmoid gate. That will create an output of only selected values.

The output h_t (figure 3.13) is generated by the following equation:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.6}$$

$$h_t = o_t * \tanh(C_t) \tag{3.7}$$

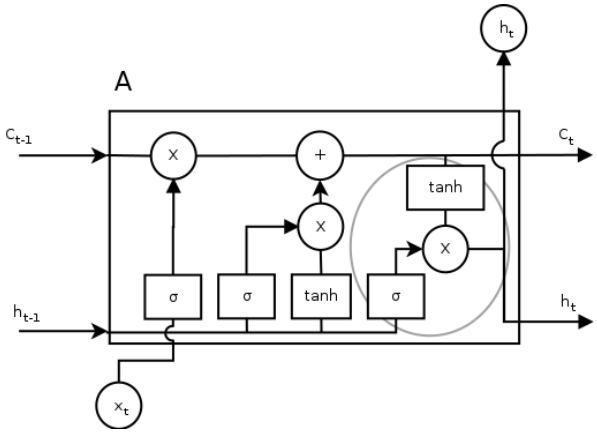


Figure 3.13: Output of h_t

3.3.1 Preprocessing

Input data have a standard deviation of 22674710 and mean of 5010694. Data for `LSTM` network have to be scaled from 0 to 1. Therefore to use model training properly, everything needs to be scaled down.

Another operation would be to change raw data into data with window. Every point of time needs to have a window containing data which have come immediately before this point. For better understanding see figure 3.14. The window size is one of the parameters which should be estimated on data. In this case, we start with the window size of 40.

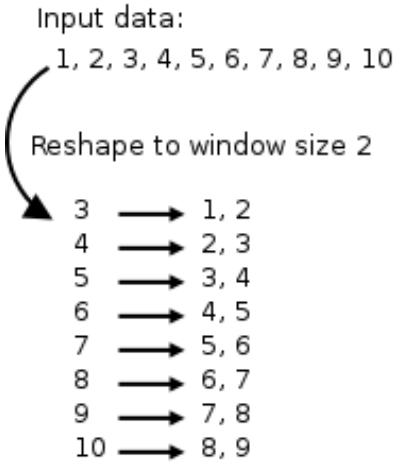


Figure 3.14: Reshaped data for `LSTM`

The last thing which needs to be done is to create two subsets of data. Since this is a time series, we can't use any cross-validation. The original context is to forecast future based on old data, which couldn't be done on

shuffled data. The ratio of 80% training data to 20% testing was used to see if the approach had similar results as in `ARIMA` case.

3.3.2 Creating models

3.3.2.1 Basic LSTM

Basic `LSTM` model contains one layer of nodes (figure 3.15). The output is forwarded to densifier which outputs one number as a result for each input window. There is no relationship between windows. That means there is no need to keep or transfer the state.

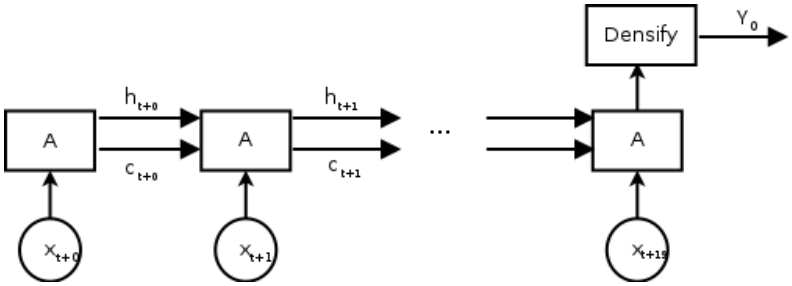


Figure 3.15: Basic `LSTM` model with on layer

Another property which needs to be set is loss function [32]. Training procedure needs to estimate how the model performs in comparison with real data [1]. For that process mean square error function is used. `ADAM` algorithm was used as an optimizer because it seems to perform the best in most of the cases [33]. As was mentioned earlier, we can't make any assumptions about data. Hence `ADAM` seems to be the best option.

[1] Names of dimensions are omitted, data are transformed to series of real numbers

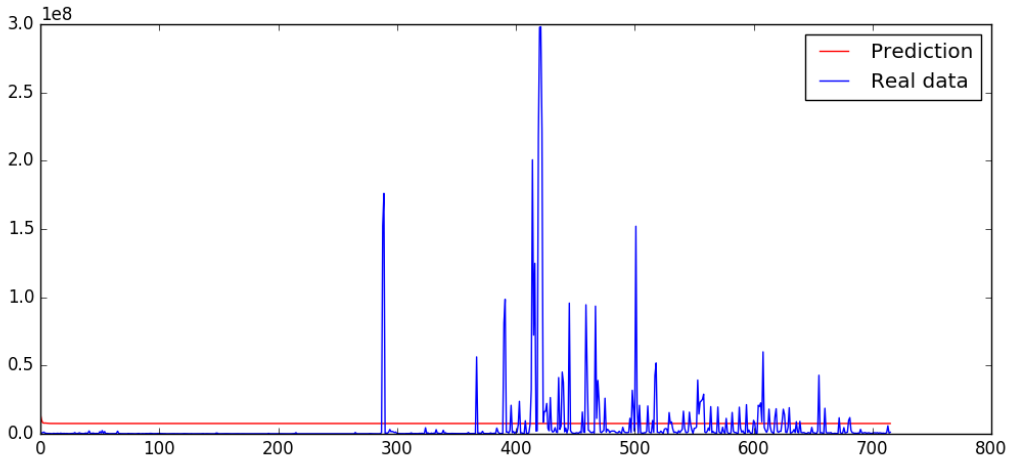


Figure 3.16: Result of basic `LSTM` network model

Figure does not show any improvement in forecasting in the long term (figure 3.16). The resulting mean squared error was estimated to 25759944.4567. That is a little bit worse (679568.9452) than `ARIMA` forecast.

3.3.2.2 Stateful `LSTM`

Stateful `LSTM` model contain one layer of nodes (figure 3.17). The output is forwarded to densifier which outputs result. Relationship between windows is propagated through the network with input state marked as C_{t*} (figure 3.17).

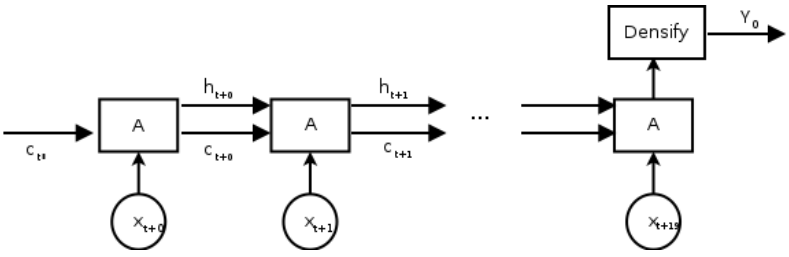


Figure 3.17: `LSTM` model with state input from past `LSTMs`

All the properties related to internal function (optimizer, loss, ...) stay the same. The only property which is added is statefulness. The problem of the proper size of the window is now less significant since we are propagating state through the network.

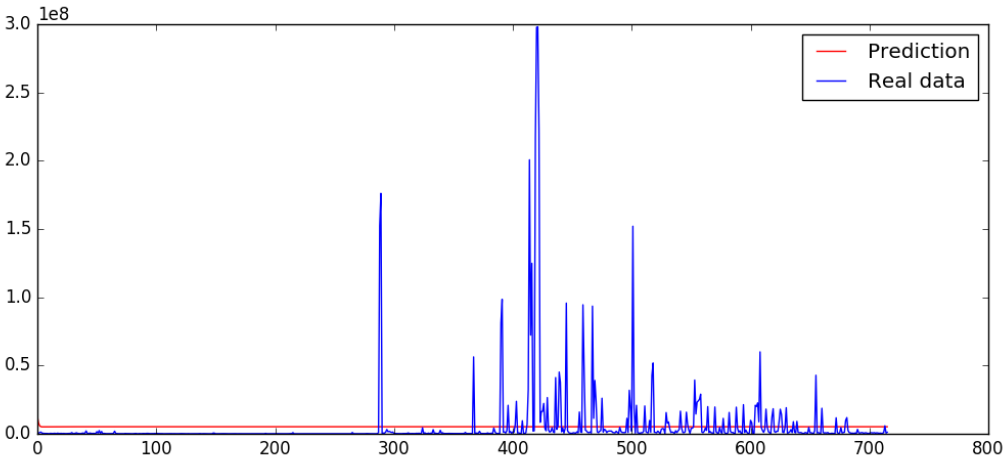


Figure 3.18: Result of stateful `LSTM` network model

Even though it seems that there no improvement, the mean square error shows that there was a change for better. Mean square error was 25723488.664 which is 36455.7927 off than basic `LSTM` model. Nevertheless, this result is hardly as good as `ARIMA` model.

3.3.2.3 Stacked `LSTM`

Another way to improve forecast should be to stack `LSTM` models (figure 3.19). Whole layers of hidden states can be forwarded to new layers which can better capture new features of data. For example, changing frequency of a signal can be divided into different layers. Since there is hardly anything we can assume about the input data, this seems to be the best approach for a forecast.

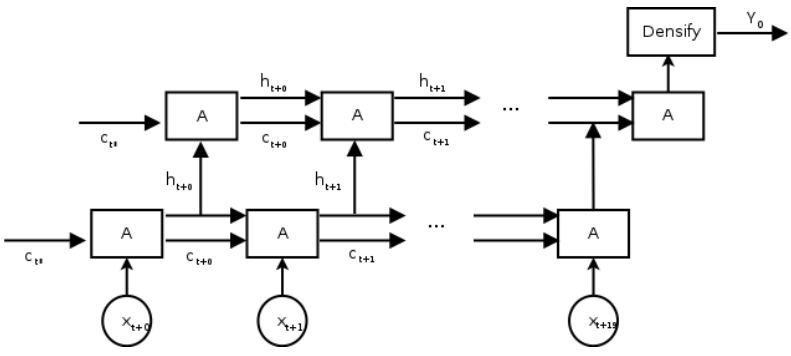


Figure 3.19: Stacked LSTM in two layers

The idea seemed promising, but results are not better than in one layer stateful `LSTM` network. Mean square error was 25752725.4935 which is worse

(the difference is 29236.8295). As the figure 3.20 shows, model performance does not significantly differ from others.

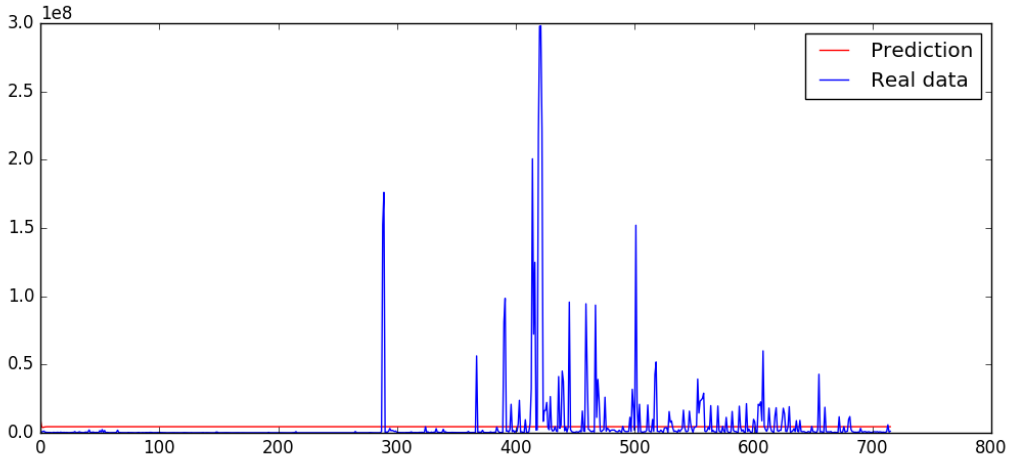


Figure 3.20: Results of stacked `LSTM` in two layers

3.3.2.4 Practical use of `LSTM` networks

We have so far considered `LSTM` networks on the worst data with a lot of spikes, questionable stationarity and no visible trends. However, there are much greater advantages if we consider different use cases. In case of `ARIMA` model we always had to retrain the whole model for next prediction. This is not the case for neural networks. Weights can be changed according to the new data without reconstructing the model. Classifier can be recycled and adjusted during runtime.

Another observation is that predictions `MSE` are getting bigger with the number of steps taken. This can not be avoided for general data. Let us keep in mind that if we have a trained classifier, we can create predictions very quickly. In that case, we can predict small steps in the future in almost constant time.

We will give a small window of data to the classifier which then provides us with a short prediction. If we use data item which has a collection interval of half a minute, we can create reliable predictions for five minutes in the future. That is a significantly shorter time interval than in the tests we did for each `LSTM` configuration.

For this case we will use simple `LSTM` network with one layer of neurons. Training will be done in ten iterations with the same configuration as in the last tests with a basic `LSTM` network. The data are a representation of bytes downloaded per minute from a very busy web server. They obviously create

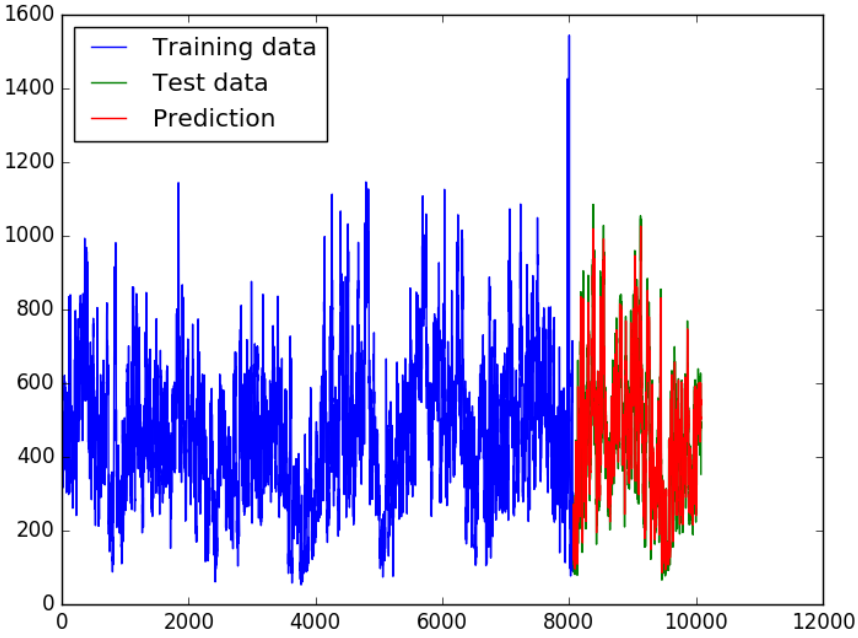


Figure 3.21: One step prediction with `LSTM`

some day to day trend but have a lot of spikes. We can see how the classifier did on a short prediction in the following graph (figure 3.21).

Results are, as expected, much better. Computational speed of the prediction was bearable. Training of such a model took a long time which was also expected (on platform `GeForce GTX 1060`). We can see that the classifier predicted each new step almost correctly without any problem.

That would mean we do not need to create long term predictions, we can simply predict quick, short term batches of data and use them for analysis. To compare how the performance of predictions deteriorates in the future, let us zoom at the end of the dataset and create a prediction for a long term of a hundred steps (figure 3.22). If we compare the results, we can clearly see that the prediction works for a few more steps and then loses its accuracy and continues only as a constant.

To get a better view of how `LSTM` captures the seasonality and how important it is for predictions, let us consider artificial data of repeating normal distributions. They can symbolize shoppers connecting to a virtual restaurant ordering food, showing large peaks during the middle of the day. Arrival of clients could start rising from the morning and decline to zero during the night. There is a clear day to day trend. These data are constructed to be simple for `LSTM` classifier – even if the classifier has one layer and is stateless.

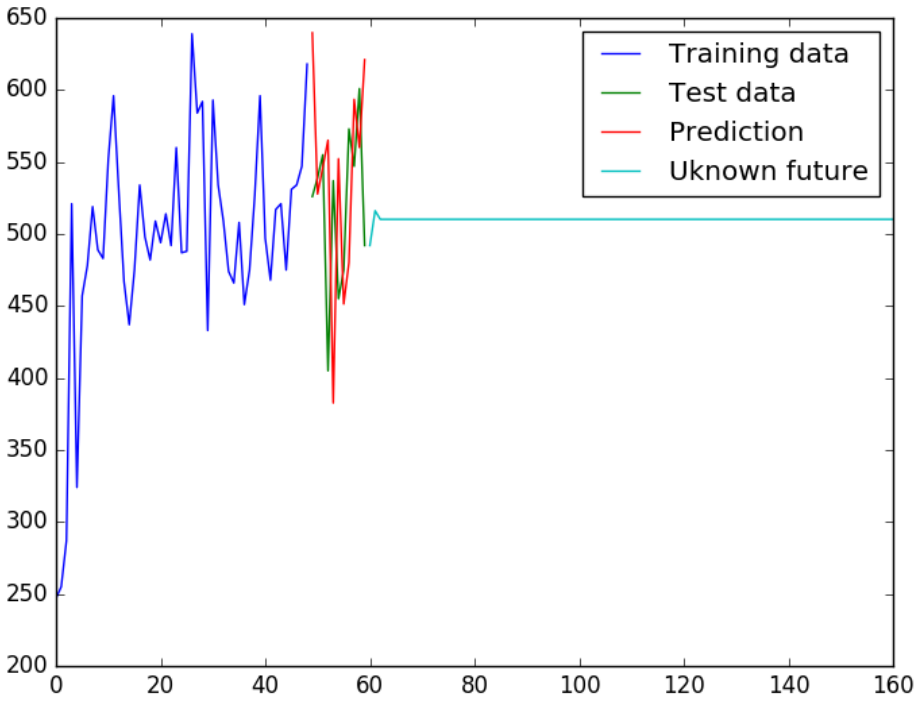


Figure 3.22: One step prediction with `LSTM` with further prediction of unknown data

For `ARIMA` another type of model named *Seasonal ARIMA* would need to be used. With these data we attempt to make new prediction with the same basic `LSTM` model and observe results (figure 3.23).

We can clearly see that `LSTM` found the seasonality. If we take into consideration that every peak represents one day, we have a reliable prediction for a few days (administrator can get a fairly good idea how the data will continue). This classifier is reusable and can adjust its weights the next time we use it. That means we have a great tool which can help with the monitoring of future problems.

3.4 Conclusion

Collecting data from resources in information systems is an easy task. Every significant part of the system usually has some representation in a monitoring system. It is reasonably harder to create collectors for all the different monitoring servers. This resulted in creating an independent layer where for

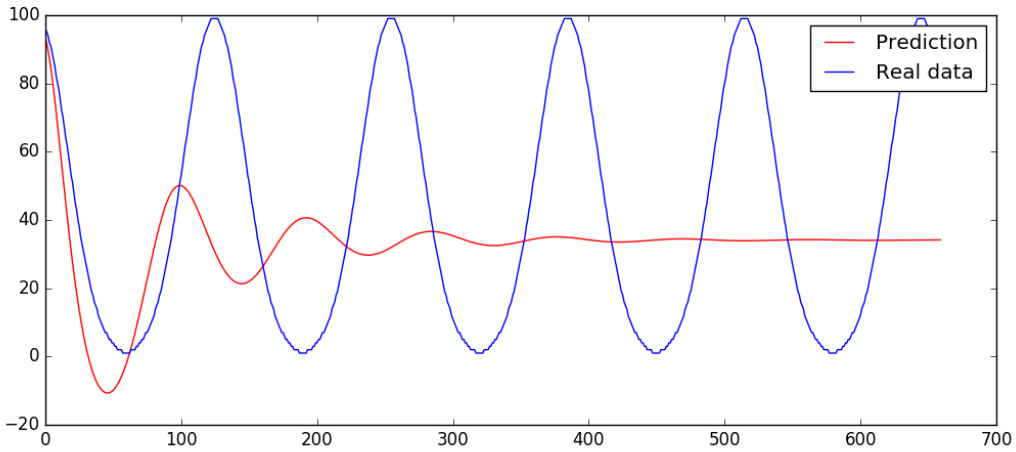


Figure 3.23: Future prediction with well trended data

every new system some application can be defined which is responsible for contacting `REST API` and getting the data which are needed.

Collecting all the data from all the resources showed to be unreasonable, usable data were filtered with standard deviation. Any method, which would compare multiple time series and state which should be used, was left out. There was a simple reason behind this. Removing series automatically without the knowledge of context could lead to removing a future source of information. Another reason could be the computational time of decision. Standard deviation, which could be estimated reasonably fast, was found sufficient.

The problem with the quality of data continued throughout the chapter. Basic analysis on first dataset proves that the small autocorrelation makes any long forecast quite impossible. Very promising was `MA` part of `ARIMA` model which by `AIC` estimation showed, that the model can use seven lags from the past. Testing proved that `ARIMA` couldn't create any long forecast.

Another possibility which is currently vital in stock prize analyses was to use some machine learning technique. The first idea was to use some recurrent neuron network. Due to gradient descent problem, basic `RNN` was skipped and `LSTM` network was used. Three models were constructed with different inner structure (one layer with long window, stateful and stacked in subsection 3.3.2) . Unfortunately, for generic data without any expert knowledge, the models didn't make any major difference in the long forecast (table 3.1). `ARIMA` proved to be faster and predicted better in case of mean square error.

The problem does not need to be in `LSTM` model itself but in the input parameters which need to undergo further tunning. In this paper there was a setting for window of lags 40, improvement can be done in setting window size to 120 (lags usually have a distance of one minute between each other,

Name	MSE
ARIMA	25080375.5115
LSTM Network	25759944.4567
Stateful LSTM Network	25723488.664
Stacked Stafeful LSTM Network	25752725.4935

Table 3.1: Mean square error in trained models

120 would be two hours long window) or even more. This could lead to much longer training time with an unclear result.

A more promising method would be to use LSTM network just to estimate few lags in the future. The network can adapt in runtime and the model does not need to be fully trained again and again. Adaptive training methods gave LSTM networks an advantage over ARIMA. Further work is going to be focused on adaptive learning in LSTM and to find out if ARIMA can cover long term predictions better than LSTM. This approach turned out to be very promising. With classifier already trained, creating predictions is a matter of short computational time. Because ARIMA can't dynamically adapt, it would be unreasonable to use it in the same way.

Another promising method would be using ARIMA as preprocessing. With cleared data, LSTM can capture trends much better. Again, the procedure of estimating forecast needs to be as generic as possible. That mean this approach should not be used.

The final statement would be to better select time series. With better quality of data comes better probability of a good prediction. There is no generic model which can create a forecast with a small error on every input data.

Chapter 4

Root cause analysis

We have mentioned that administrators do not have a very good picture of the problem. All of risk management can profit from a more extensive overview. There are methods known from applied economics [17] which can address the issue and root cause analysis (RCA) is one of them. Definition states that root cause analysis is applied methodology used to identify and correct a root cause instead of addressing the symptomatic results. In our case, symptoms can be e.g. failed service, yet root cause could be depleted hard drive memory.

To estimate root cause can be a matter of a good setup in monitoring. If all of the dependencies are correctly described, administrator can directly fix them. But this is not the case for every alarm and every trigger. That is why in this thesis we will try to analyze alarms by multiple methods which then lead to better decision making by the administrator.

4.1 Our approach

We have analyzed one of the major tools to create RCA in cloud environment (section 1.2). The purpose of our application was summarized in the application design. Exploring *Vitrage* showed that the application design was not that far from the already known solutions. In this section we will explore possibilities of RCA analysis based on statistical tools and machine learning methods. That is something not used by *Vitrage* by default. There are obvious reasons why. No statistical method or machine learning can create analysis which maps problems at one hundred percent accuracy. That means the administrator using the application of this work should always judge by themselves. Obvious advantage is that the administrator does not need to define any templates but also the administrator needs to know a little bit more about analysis and have good knowledge about of his monitored topology.

Finding the root cause from data gathered directly from monitoring software could be very tricky. Some of the issues were already addressed in the past chapter. Here is a small summary:

- A sampling frequency of the checks can create unusable data. The reason is mainly that monitoring software is not designated to collect data for further analysis but for on-time trigger evaluation. Sampling is therefore driven by the possibility of failing not for data analysis.
- Dependencies are not easy to find. Basic ideas like Principal Component Analysis (PCA) could lead to excessively load time of monitoring software. Finding correlation between all the items can be inconceivably hard.
- Administrators tend to gather data mapping only symptoms. It is much easier to test the whole stack with application check rather than monitor each item in the stack. That creates vast space for fails without anyone noticing.
- Data tend to be more white noise than stationary. For example load of a system can jump from zero to ten very quickly and fall back to zero. Graphs project this instability as spikes leaving system load as unusable data.

This chapter creates analysis of what could be done with such data and how to deal with them. To deal with spikes, we can estimate dynamic thresholds [2]. A more prominent structure can be created with time relationships. If some fails happen at the same time period, an administrator should see them all in one group. Also, in many cases correlation could provide much needed help.

4.2 Selecting thresholds

Input data are usually bound to some lower and upper limit. Utilization of CPU will never go above 100 percent as well as throughput of 1 Gb link never reach 1.1 Gb. Furthermore, data do not usually jump low and high if there is nothing wrong (except for CPU load and some other special cases). Underlying trends are steady rise or day to day spikes as clients connect to the system. Smart monitoring can help with creating these thresholds.

4.2.1 Static thresholds

Some of the thresholds are already known from SLA or from monitoring systems itself. Administrators define them to notice something which they consider abnormal. Regular ones are 10 % to 25 % limit of running out of disk space. These can be optimized yet they are not the case of this study.

4.2.2 Data driven thresholds

Time series, which are usually the data from monitoring, have evolving pattern. Thresholds are more dynamic and should be calibrated by the data's

most recent state. That is why data drive thresholds need to be adjusted periodically. This approach comes with some obstacles:

- We have to consider problems which are a part of such data. What if we pick a relatively extended period and estimate threshold based on past high values? That would render the whole threshold unusable. The value of the threshold can get so high it can no longer detect any failing state. The sensitivity would be just too unbearable.

- Another problem may occur if values of data item decrease to unrealistic levels – for example servers which are in the idle pool, ready to be used in production. These servers will have much lower utilization and can set dynamic thresholds low. If the threshold will get used, it will fire a lot of false alarms. Thresholds would be considered oversensitive.

To apprehend these difficulties, administrators should decide how old data look back should be used. If monitored data items are not valid, only explicit evaluation can help. More help can be offered in monitoring systems. For example, *Zabbix* can specifically disable whole client and not collect data when a server is not in production.

From empiric observation [2] one week of data seems enough to create thresholds. Based on the target monitored system, there can be apparent similarities which repeat itself. These trends in data can mean users attending some service at server doing their job or logging out of the system during lunch hours. There are many possibilities. On the other hand, we should consider some days which are different in case of trends. Weekends and holidays can create bigger space for the problem of lower utilization. If data shows a decline in some days, one week still seems to be enough to cover basic distributions.

Estimation of dynamic threshold could be covered in these steps:

1. Determine lower and upper limit. Based on the data distribution, a percentile should be also considered.

2. Sort data in ascending way where maximum is percentile 100.

3. Every other percentile needs to be counted out of rank, where rank is $(percentile * sizeofdata)/100$.

4. Resulting rank needs to be rounded up and through this procedure, we can select the desired threshold.

5. If given percentile exceeds minimum or maximum, it should be rounded up to the data's minimum or maximum.

After applying this approach to the data from monitoring, alerts started to occurred pretty fast. The reason was because the observed period was too

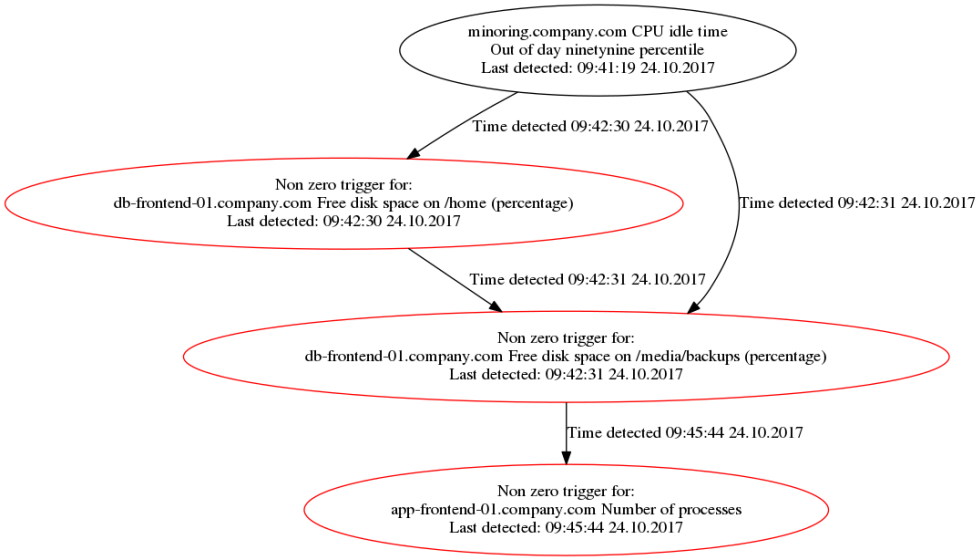


Figure 4.1: Over sensitive threshold

small and thresholds were over sensitive (figure 4.1). Following figure shows failed states in a red bubble. These bubbles are bound with arrows showing which items failed first.

The red bubbles are items marked by the monitoring system. The black one is stating data going over 99 percentile, which is almost maximum. Sampling period was one minute. Since the data should be gathered from the whole week, the threshold is valid only after getting more than 10080 values.

Creating better thresholds could not directly help with monitoring. We have keep in mind obstacles with a sensitivity of estimation. There are some more technical problems. For example, if there are missing data or data are collected with different sampling period. Smart monitoring should consider these issues. In practical use, this method proves to be helpful even if it generated some false alarms. The main usage could be to use them as simple classifier for data items which have simple level of failure. If the value exceeds threshold, it could be classified as abnormal. In application design we can see component which is responsible for classifying prediction, that could be done with thresholds.

4.3 Time-based relationships

Some time based relationships have already been mentioned (figure 4.1). The idea is to map failed states to a diagram which better explains what happened in the past. Each failed item would have some time consequence if the failure did not occur just by itself.

Collected data contains timestamp which gives enough information to estimate where the trigger fired. This could be one of the problems because timestamp can be given from system with different time notation (`UTC` vs. `CEST`). If we consider only Unix timestamp, we can get just too much information. By the standard [34] Unix timestamp counts seconds form 1.1.1970 – that is an enormous number in case of monitoring. Essential time relationships between failures do not have more significant interval than days. From literature, we can find that the Mean Time Between Failures (`MTBF`) was considered roughly about seven days [35].

Another alternative would be to use logical time [36] which can better map causality in a distributed system. For this use case, it promises better aggregation of events. If we consider counting logical time from the beginning of the monitoring, it will hardly reach the size of the Unix timestamp.

4.3.1 Logical time

In case of collecting information about the fail states with logical time, some conditions need to be met. The collector needs to start the counter for each received item and give it a default value of zero. In next sampling period, all the counters need to be iterated by one. The sampling period needs to be uniform for all the data items.

These generate several problems:

- The collector needs to gather all data or at least increment their logical time even if they will not be used in later analysis or data haven't changed. A preferable solution would be to gather this logical time only on failed states. If there was a smaller number of them, fewer counters would need to be adjusted.
- Another issue will be if the collector restarts. If the counters would not be saved in some persistent memory, all of them can appear as if they happened at the same time and also without any time relationship.

If the condition of saving logical time and collecting in uniform sample rate is met, an algorithm could go as follows:

1. Collect in $time = 0$ data from the monitoring system, give zero logical time for data items which are marked by the system as failing.
2. If there is a new item in the next sample, give it zero logical time and create relationship from old item to new detected item.

Based on the described algorithm, the result can look like this:

In figure 4.2, we can see the problems of logical time. The analysis considered data item `IO Wait` on server `minoring.company.com` with `Number of`

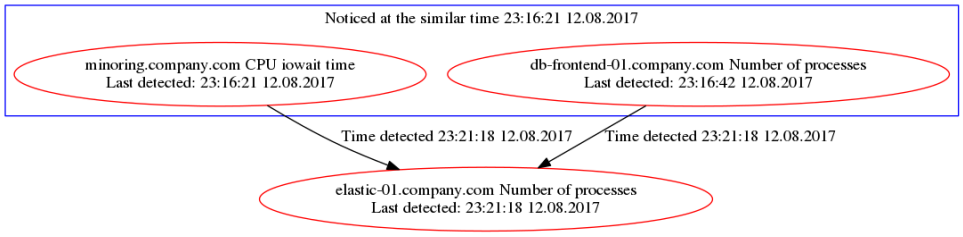


Figure 4.2: Diagram of failed items with logical time

`Processes` on server `db-frontend-01.company.com` in the same period, even though the difference between them is twenty seconds. Analysis correctly marked two relationships from items of the box of same time notice to item `Number of Processes` on server `elastic-01.company.com`. Of course, some administrators would consider the box as correct one because twenty seconds is not a significant time distance. But that wasn't the case. The problem was that collector noticed this item in the same sampling period which was thirty seconds. That's why monitoring created one cluster of two items which happened relatively close together.

4.3.2 Real-time

In real time, we have a timestamp for each failed item. These are numbers which can be easily compared and sorted. They do not give us an opportunity for creating fast clusters based on the sampling period, yet they change the collecting process much more.

For the logical time, we made a sample in fixed time delay. Monitoring systems do not sample the data items at fixed intervals. Some of the items have shown trend and do not need to be collected that often. Those can be system uptime or system time. That's why collector needs to keep separate interval for each item.

An algorithm will then sort the items and create relationships. Since there is hardly any item which happens at exactly the same time, it is always going to be a linear graph (figure 4.3).

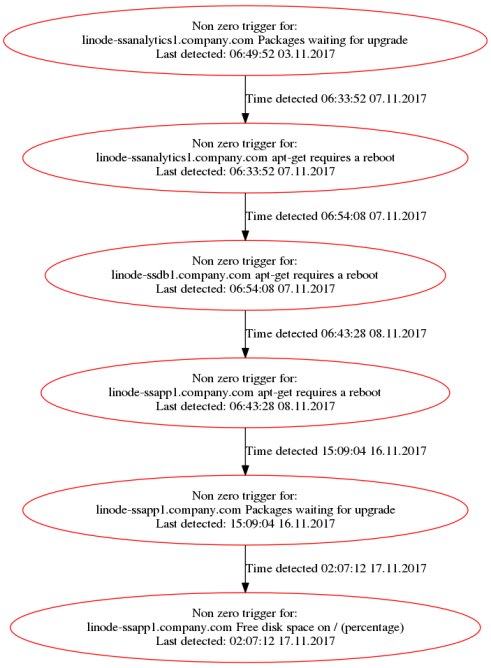


Figure 4.3: Diagram of failed items sorted in time

To reconsider the sampling period and maybe profit from easy clustering from logical time, each timestamp can be stripped from the last bits. That could create classes of data items noticed in the same logic period. For example, items seen in one minute can be marked as noticed at the same time because seconds do not give any extra meaning.

Stripping the bits from timestamp can be created with few bit operations:

```
int(
    int(item['time_of_last_notice'])
) ^ (
    int(item['time_of_last_notice']) & 63
)
```

This would in later analysis mark items noticed in one minute and few seconds as noticed at the same time. 63 is a mask for bits which can be omitted. To mark in the same way half a minute, mask would be 31. The following diagram creates relationships with strip bits for fifteen seconds (figure 4.4).

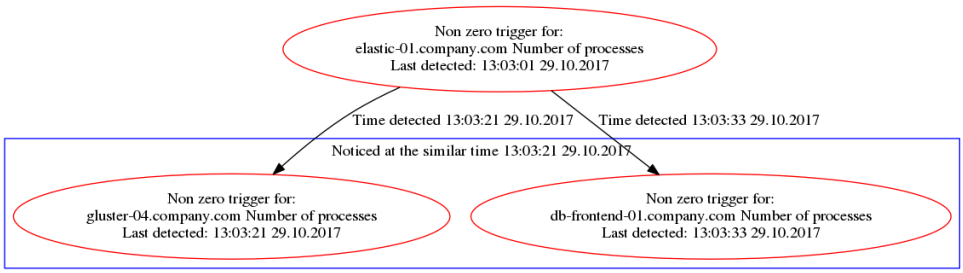


Figure 4.4: Diagram of failed items sorted in time with stripped bits

Since items in the blue box have a difference of ten seconds, the algorithm has put them in together.

Creating time-based relationship is not so complicated task. To establish some different graphs than linear relations, logical time was also considered. The time was incremented in sampling periods. That proved to be more tricky than helpful. A more convenient approach was to collect data items in different periods based on the monitoring system data items interval. This approach has not left any space for clusters creating a linear graph only. To tackle the issue, stripping few bits from a timestamp was introduced. Even after exploring these possibilities, a more significant question is if the graph is usable, the more user-based analysis needs to be provided.

4.4 Fail states aggregation

We have already shown some aggregation of fail states. In this case, we have multiple alerts which happened in different time periods. These alerts are only bound by time-based relationships. That means we have a long line giving no additional information to the administrator. What if we cluster those alerts and show just the clusters?

Let's have two clusters of alerts (figure 4.5). These alerts are connected to a slowly depleted resource, so the alerts gradually occur and when the requests peak, the alerts gradually go away. To make two clusters, this incident had happened twice. The time of occurred alerts will center around two peaks. Let's simulate this state with normal distributions.

4.4.1 Logical time aggregation

Logical aggregation would give a fast creation of clustering alarms based on sampling frequency. If we would set sampling period to thirty seconds the alarms would be divided into multiple windows. That would make much smaller number of all alarms. The problem is that clusters would not cover everything and with the higher sampling frequency, we would get exactly the

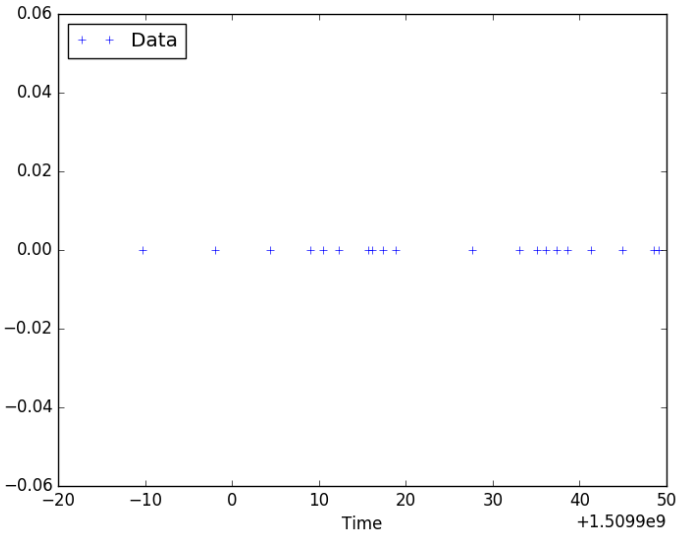


Figure 4.5: Two normal distribution escalating crisis in system

same result. Higher frequency is a desirable state, so it would be unreasonable to rely on this method.

The example (figure 4.6) clearly shows that two clusters would be formed. That is correct, yet the first one apparently cut peak of the first normal distribution in half. Since we can't make a decision when sampling happens, this state is normal.

4.4.2 Kernel density

Other way of finding these clusters would be to perform some clustering technique. There are plenty of them, but there is a catch. Clustering is usually performed in higher dimensions than just `1D`. Using such an algorithm would lead to unnecessary use of system resources.

That is why the right way is to use kernel density estimation functions. It is a non-parametric way to estimate the probability density function of a random variable. The occurrence of alarms indeed is a random variable.

Kernel density function can be defined as follows: Let $(x_1, x_2, \ldots, x_n)$ be a univariate independent and identically distributed sample collected from some distribution with an unknown density function f. The resulting function f has its kernel density estimator as:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

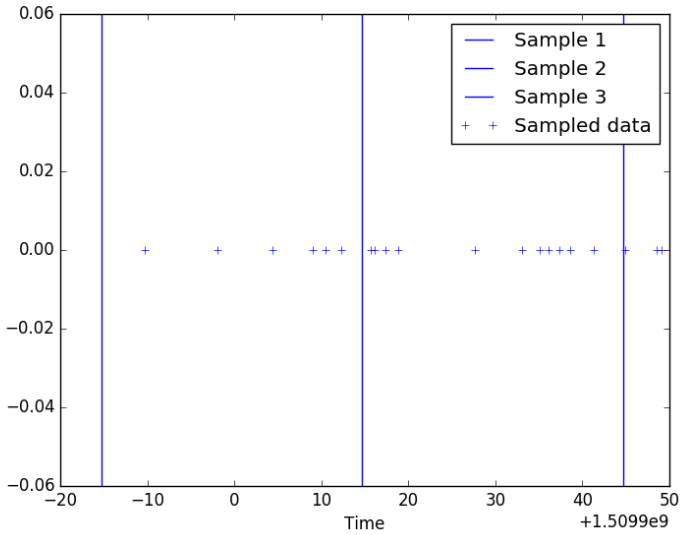


Figure 4.6: Uniform sampled data

Where K is the kernel function with constraints: function is non-negative with integrates to one and $h > 0$. h is called bandwidth and it is used as smoothing parameter. A kernel with h as subscript is called the scaled kernel and is defined as $K_h(x) = \frac{1}{h}K\left(\frac{x}{h}\right)$. Choosing the h according to the data is not a trivial task. In this paper we choose Gaussian kernel function [37].

$$K(x) = (1/\sqrt{2\pi})e^{\frac{-x^2}{2}}$$

There are constraints to use this kernel, and that is to have univariate data, like in our case. Another significant feature is that Gaussian kernel generates functions which can be straightforward derivative. That is going to be very useful for further cluster computation.

To estimate correct bandwidth, we can perform a full search starting from 0 ending at 1. If we neglect that $h \in \mathbb{R}^{[0,1]}$ we would need to search with discrete increment which would enlarge possible error. Intuition says the lower the bandwidth, the better. That does not need to be true. We can again use AIC [28] to estimate the best fit. Another way would be to use mean integrated square error (MISE) or asymptotic mean integrated squared error (AMISE) to find the smallest difference between a real density function, and that would be our best-fitted bandwidth. This is really time consuming and unnecessary for our task. The other way of estimating bandwidth would be to use some rule of thumb.

Silverman's rule [38] which also works only for one-dimensional data, estimate $h = \left(\frac{4\sigma^5}{3n}\right)^{\frac{1}{5}}$ where n is the size of data and σ is the standard

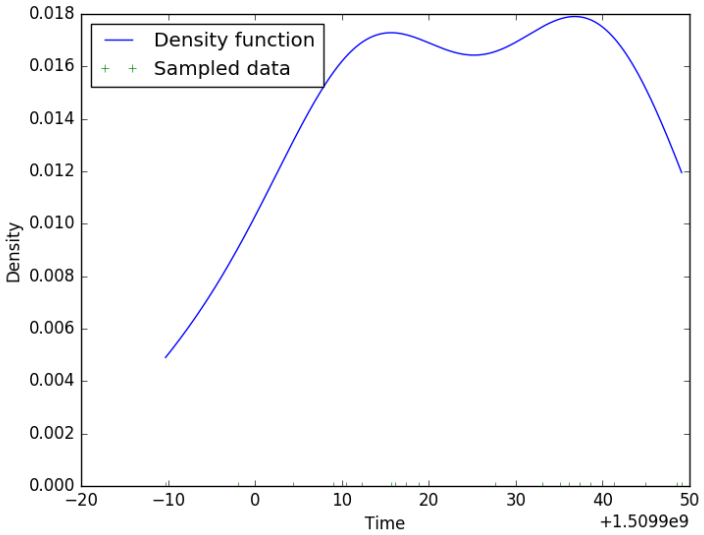


Figure 4.7: Kernel density estimation with bandwidth from Scott's rule of thumb

deviation. The method is based on the assumption that the underlying distribution of data is a normal distribution, which in our case is certainly true without any doubt.

Scott's rule does not have any constraints for higher dimensions, h is estimated as follows $\hat{H} = \frac{1}{n^{d+4}} \sum^{\frac{1}{2}}$ where $\sum^{\frac{1}{2}}$ is the Covariance matrix and d is dimension. This rule is the generalization of Silverman rule [39] for higher dimensions. The result would not give just one h but matrix of $\hat{H}$ which would show correspondent h for each dimension.

Using the rule of thumb speeds up whole computation and makes the usage of the result much faster. After applying the kernel density estimation to the data, we would get little difference in outcome:

As the figures (figure 4.7, figure 4.8) show, both methods of bandwidth estimation clearly discovered two normally distributed alarm events. The difference is in the sizes of spikes. This is the reason why an only Gaussian method is used, to create a function which has derivation in all of its course. The intuition again dictates the bigger the hills are, the better the following results.

To estimate the clusters, we will find local maxima and minima of a density function and divide data into groups by the minima of a function (figure 4.9). Apparently, from the figure (figure 4.7) we can see that Scott's rule of thumb prepared the function for such an analysis better.

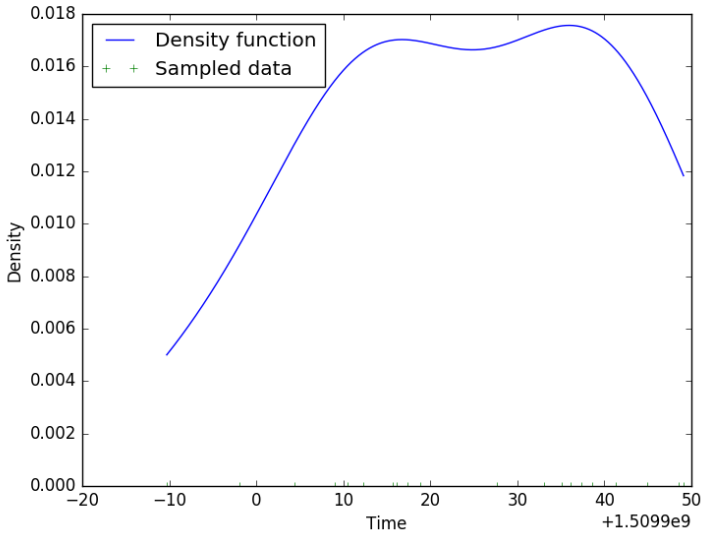


Figure 4.8: Kernel density estimation with bandwidth from Silverman's rule of thumb

So after finding the local minima and maxima, we would have two separated clusters:

Creating clusters of alarms can provide a better overview in time of crisis. Count of failed items can get to huge numbers. A human expert can get lost. Having some kind of aggregation can focus administrator directly on the problems. Logical clustering proved to be related precisely to the sampling period of the data items, that is not good enough because sampling does not give any context of data. Kernel density function estimation proved to be much better and with Scott rule of thumb helped discover the structure of input data far better. For further use, this approach will be the preferred one.

4.5 Correlations

Another compelling way how to compare two time series and decide what they have in common is correlation. Correlation is not causation, so the results should always be checked from the human expert who understands the meaning of data. Based on statistical distance we can decide if there is maybe something familiar. In this case, we will use *Pearson correlation coefficient*. There are some problems which can make correlation useless in the monitoring system analysis.

- Time needed for computation of *Pearson coefficient* could be a real drawback and create whole monitoring analysis too slow.

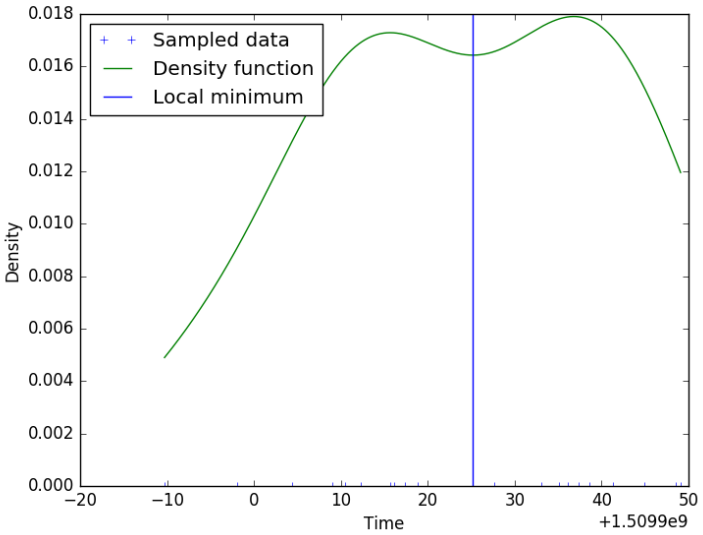


Figure 4.9: Minima division of data into two clusters

- Trends can give correlation much bigger significance than it has. Trend removal needs to be introduced to tackle the issue. This should remove linear trends which are very significant in correlation.

These issues need to be solved. In this case, the correlation represents the meaning of distance between time series. Therefore, the time plays a significant role in this computation.

4.5.1 Time of computation

Processing *Pearson correlation coefficient* is a well known process, and used platforms offer their implementation. Selecting time series from a database and then counting correlation could lead to unnecessary overhead. Used database platform `PostgreSQL` offers aggregation operator `corr` [40]. The basic idea was to select data from records and forward them for the computation.

Listing 4.1: Algorithm to select data over PostgreSQL database and compute *Pearson correlation coefficient* over them

```
WITH subst1 AS (
 SELECT cast(result as float) as res1
 FROM record
 WHERE hostname = 'elastic-01.company.com'
   AND service = 'Available memory'
   AND time_execution > 400556032 ORDER BY time_execution DESC
),
subst2 AS (
```

```
 SELECT cast(result as float) as res2
 FROM record
 WHERE hostname = 'db-frontend-01.company.com'
   AND service = 'Available memory'
   AND time_execution > 400556032 ORDER BY time_execution DESC
)
SELECT corr(subst1.res1, subst2.res2) from subst1, subst2;
```

There are some implementation drawbacks. `PostgreSQL` does not allow to use of bitmap indices [41] defined directly in table definition. That leaves space for query processing and sometimes can give much greater performance if the query is formulated correctly. This is not the case since we are selecting two subsets and later comparing them. For each `SELECT` in a subquery, filter scan is performed. There is no speed up in caching because the data are from different subsets.

Listing 4.2: Query plan of query

```
                                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=891754.15..891754.16 rows=1 width=8)
   CTE subst1
     ->  Sort  (cost=114078.98..114089.59 rows=4243 width=16)
           Sort Key: record.time_execution DESC
           ->  Seq Scan on record  (
               cost=0.00..113823.33 rows=4243 width=16)
                  Filter: ((time_execution > 400556032)
 AND ((hostname)::text = 'elastic-01.company.com'::text)
 AND ((service)::text = 'Available memory'::text))
   CTE subst2
     ->  Sort  (cost=114280.14..114297.51 rows=6947 width=16)
           Sort Key: record_1.time_execution DESC
           ->  Seq Scan on record record_1  (
               cost=0.00..113836.85 rows=6947 width=16)
                  Filter: ((time_execution > 400556032)
 AND ((hostname)::text = 'db-frontend-01.company.com'::text)
 AND ((service)::text = 'Available memory'::text))
   ->  Nested Loop  (cost=0.00..589676.75 rows=29476121 width=16)
         ->  CTE Scan on subst1  (cost=0.00..84.86 rows=4243 width=8)
         ->  CTE Scan on subst2  (cost=0.00..138.94 rows=6947 width=8)
```

As we can see, both queries performance have similar costs. Opposite to bitmap operation, execution of query instead chooses a filter which better maps the nature of `AND` expressions. The reason could be because after insertion of a vast portion of data, a cardinality of a column might become too big for bitmap operations.

Better results could be reached by using materialized views. Because correlation is counted each time when some analysis is needed, the view would need to be updated multiple times. This can generate big overhead on database processing and slow down other operations.

In the end, the fastest method was to transfer data from database to application memory and count the correlation during runtime. The application can memoize the subsets of data and use them multiple times before the task is finished. Another problem is that `PostgreSQL` does not have functions for signal processing which is needed for trend removal. The implementation would need to be created and generate another overhead.

4.5.2 Trend removal

If we closely think about *Pearson correlation coefficient* we have to consider that the correlation coefficient is between $+1$ (a perfect linear relationship) and -1 (perfectly inversely related) – zero state no linear relationship. Numbers bigger than 0.5 state significant statistical relationship and that is all we can deduce from the coefficient. It does not give us anything more and less. A big responsibility is put on human experts who should judge by themselves through by inspecting both time series.

In another case, we can introduce linear similarity just by adding trends [42]. If two time series have the same trend, the linear relationship is getting significantly larger. That can present some lousy interpretation. The internet is full of examples of poorly interpreted correlations between time series – those are usually referenced as spurious correlations [43]. The main idea should be first to remove the trends and then explore correlation. We are not going to consider secular trends which are underlying time series in long periods of time. In this case, the trend removal is going to be done in small time windows (five to ten minutes). Data were divided into small pieces because of long computation time, therefore administrator is going to see only the correlations in a span of a few minutes.

Of course, trends are not to be generally a bad thing. Obtained knowledge is always exploring the linearity of time series. In the process of exploring relationships between two time series, the main result is to discover variations of one series with others. Trends just make it more difficult.

So with the need for trend removal, we have two simple algorithms which can prepare the data:

- With a method of **first differences** we will remove trend by subtracting each point with the point of directly preceding it. This can be formulated as function mapping old values to new ones:

$$y_{new}(n) = X(n) - X(n-1)$$

- Another method is called **link relatives**. Opposite to **first differences** the method divides each point by direct last point.

$$y_{new}(n) = X(n)/X(n-1)$$

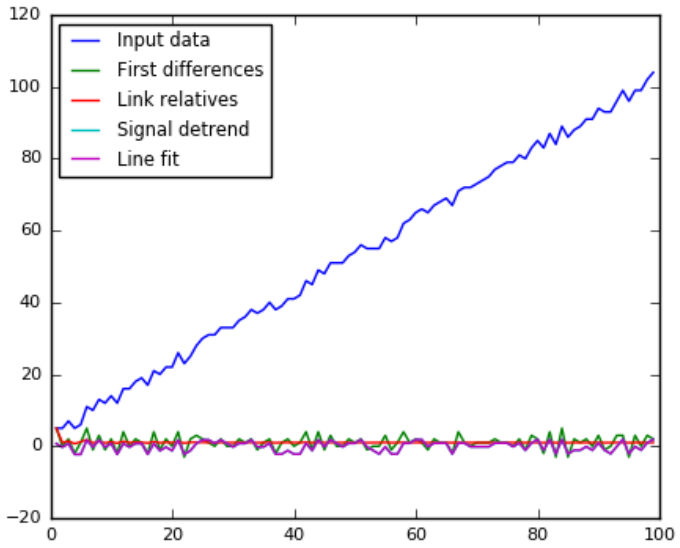


Figure 4.10: Comparison of trend removal functions

An obvious problem with **first differences** is that all the values have to be non zero. *Python's* library function `scipy.signal` [44] offers function for removing the trend which uses linear model of data. In a first step, a regression model is constructed with simple least square method. In a second step, the function subtracts data model points from real data.

Listing 4.3: Algorithm to remove trend in data with polynoma of first order (only $a \cdot X + b$) fitting

```
def detrend_avg(lst):
    z = np.polyfit(list(range(1, len(lst) + 1)), lst, 2)
    p = np.poly1d(z)
    print(p)
    new_lst = []
    for i in range(1, len(lst) + 1):
        new_lst.append(lst[i - 1] - p(i))
    return new_lst
```

To test the implementations of all three trend removal functions, a test data were generated. These data were taken from random function and added with linear trend. Notice (figure 4.10) that `Signal detrend` which is library function, is fully overlapped with `Line fit` which is a function from (listing 4.3) above.

The decision needs to be made. We need to estimate which function for removing the trend is the fastest. Another point which needs to be considered is which function removes the trend better. This can be done by measuring

Name	Time
First differences	16.43598 [seconds]
Link relatives	16.924389 [seconds]
Signal detrend	15.01764 [seconds]

Table 4.1: Speed of different functions of trend removal

Functions	Standard deviation
Original data	288674.99
First differences	1.99
Link relatives	0.0033
Signal detrend	1.414

Table 4.2: Standard deviation of data with and without trend

standard deviation of each function. If the function keeps reasonable big deviation it could be used for later analysis.

The speed was considered from a data sample of 10000000 length. Only the trend removal was considered. Library function `timeit` was used. The platform of testing was `Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz`. Keep in mind that these values are almost irrational. If we consider data item which is collected every half a minute from the monitoring server, the number of 10000000 sampled values would be roughly about two years of data. From *Amdahl law* [45] we can conclude that optimizing this part of code is not that important.

Another measurement which is necessary to obtain is standard deviation. If result from trend removal function removes all the variate from the data and leave them as constant, no further analysis would be able to establish any meaningful results. There are some obvious observations. Original data series should have the largest standard deviation – the smallest one is going to be the worst one.

We can see (figure 4.2) that signal trend removal with regression model was slightly worse in keeping the deviation than method of first differences. This observation is clearly visible in graph (figure 4.10). The decision of which function is going to be used seems to be unsolvable with methods of comparing standard deviation. That would also mean the human expert should decide which method should be used. That would be unreasonable for larger set of time series. That's why the most general function should be used.

Using library function proved to be the safest way. This was not decided because of the standard deviation of data but because of the speed. Another point is that counting the correlation is not slower than the speed of the trend removal. If we took data from speed testing and counted the speed of correlation computation, it never went under 27.40 seconds (estimate created

by more than forty measurements). After a lot of measurements, it seems that primary drawback is the speed of database. Retrieving the data of a time series proved to take too much time. Due to memoization this cost procedure is always done just once for each data item which is considered in computation. Because of other computations (future predictions, time distance relationships, ...) also need a lot of database connections, the whole process of relationships estimations is slowed down just because of correlation algorithm.

The essential question should be if the correlation is more important than other algorithms which should help the administrators in monitoring of the information system. Because of runtime overhead in computation, results took long to estimate. The administrator had the results after a long period of waiting – that means whole analysis was useless. The solution of the database slow down could be certain denormalization of database data model. This would create another overhead to the database processing. When should this denormalization happen? And which time series should be put into new denormalized tables? These are the questions which would mean total reformation in database models implementation. Because of `TimescaleDB` this process would be also computational difficult and at the end would generate another database overhead during the denormalization.

Chapter 5

Testing of application

Whole application was tested at the target company *Lukapo*. The data comes from their *Zabbix* monitoring server which collects data from the customer's servers. Following diagrams come from the application which is the result of this thesis. The diagrams are hand picked to show drawbacks and advantages of the mentioned analysis.

This chapter should explain what are the outcomes of the methods used in this thesis. We have mentioned multiple times that we have no ambition to have an application which is one hundred percent correct, but it is trying to give a bigger picture of the monitored topology. The issues were noticed during the testing on production data and especially with the administrators at their on-call shift. We will go through the issues in clustering which were noticeable only after collecting weeks of data. In the predictions section we will discuss problems of insufficient amount of data, which are marked as correct without any failure state. In thresholds we will analyse further what happens if a sudden change in data trend occurs. To see some positive perspective, we will go through some examples when the application created correct analysis and helped the administrator to fix or avoid failed situation.

5.1 Clustering

We have used estimation of density function on the occurrences of alerts during the time. We also considered use of sampling which could help in the case of speeding up the process. Both methods had their advantages and drawbacks. In case of sampling we have considered the length of the sampling period. That seemed to be unreasonable since we had to utilize sampling period as a parameter which should be chosen. On the other hand, estimating density proved to be reasonably fast and the results seemed legitimate.

There is one throwback which can have a serious impact on the result of the method. Consider two normal distributions, the first one is distributed in few minutes and the second one is distributed in one day. Those distributions

are far enough from each other to be recognized as independent (in sense of time). Our algorithm would mark them and create two clusters (figure 5.1).

The question is, how much is this going to help the administrator watching the diagram. From user experience point of view and personal feedback from the administrator of *Lukapo*, we can clearly say that the administrator has not considered it as an error, but tried to discover why such a case happened. After the discussion with the users of our application we found out that the clusters should have been labelled as clusters and nothing else. Labelling them with timestamp of the alert, which is in the center of the cluster or saying that alerts were detected in similar time proved to be misleading. The administrators usually have basic idea of what is a cluster, but they will never have any idea what is a frame containing a lot of bubbles labelled `Noticed at similar time` (figure 5.1) if the clusters are going to contain alerts with distribution in a matter of days.

On the other hand, this property is not wrong in most normal cases. We have captured in another observation different behaviour which seems to be correct. Two disasters happened in the monitored network. First one was pretty fast and generated cluster with items with average distance of seconds. The reason of the disaster was an error of the `VPN` server on the way to the system. The second disaster was closely bound to the first one (table 5.1). However, the alerts were spread out in minutes. The reason was that some application resources were unable to continue unless `VPN` came back up. The algorithm correctly marked two clusters.

Figure 5.2 shows how clusters were estimated. In this case we have omitted the graph which was the result of the analysis, we rather showed how the density function was established.

We can see that the non-parametric cluster estimation needs some heuristic which marks excessive distances between alerts as inapplicable. Those time distances can be days. There is hardly any disaster scenario which would justify clusters of alerts within the span of days. For the case of seconds, minutes and hours clustering proved to be working correctly. More improvements can be done in case of user experience. Further testing in the field of usability needs to be done.

5.2 Thresholds

There was one problem with thresholds which has not happened during short testing. If we took data which contained fail state and this state was left unnoticed for a long period of time, thresholds trained themselves on this failed state and considered it normal. After resolving the issue or with a sudden change in data, the threshold was fired and could confuse the administrator.

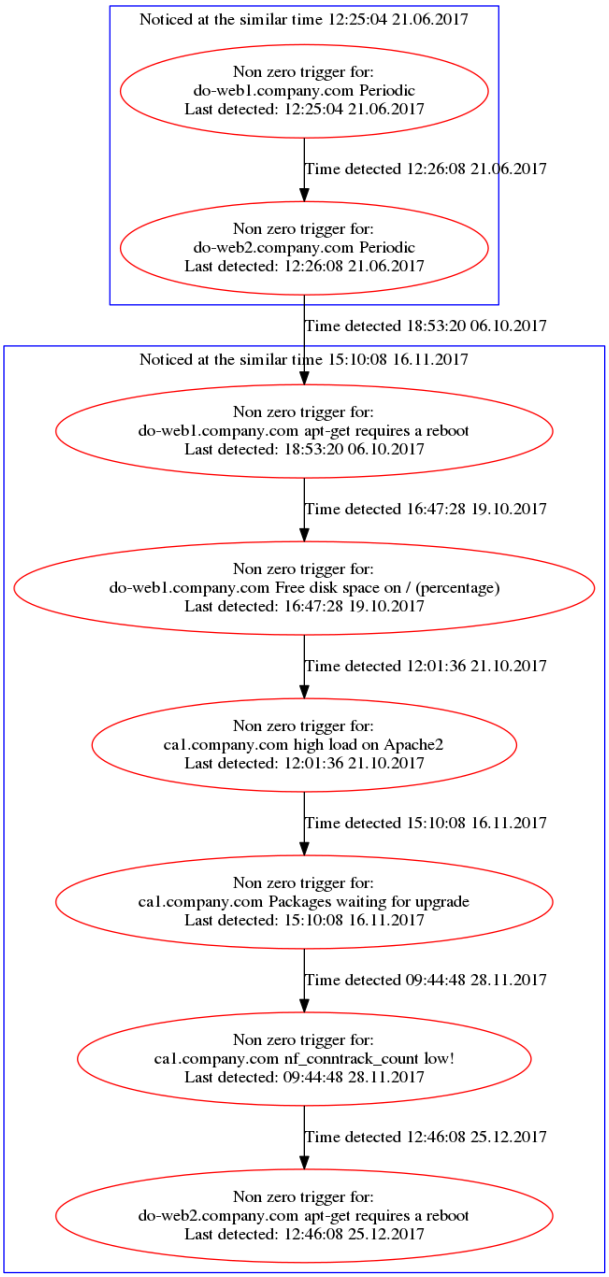


Figure 5.1: Two clusters with different density

Times	Comment
15:55:04	`VPN` server ssh connection failed
15:55:14	*Zabbix* agent not available
15:55:25	. . . alerts related to customer application
15:55:29	
15:55:34	
15:56:14	
15:56:24	
15:56:29	End of first cluster
16:05:16	Application check failed after almost 10 minutes of `VPN` shutdown
16:13:36	Application can not connect to some resource in `VPN` network
16:17:06	
16:21:56	
16:25:26	
16:27:06	
16:30:26	
16:32:06	
16:32:26	
16:32:46	Last alert related to the disaster situation, end of second cluster

Table 5.1: Times of alerts with different spread

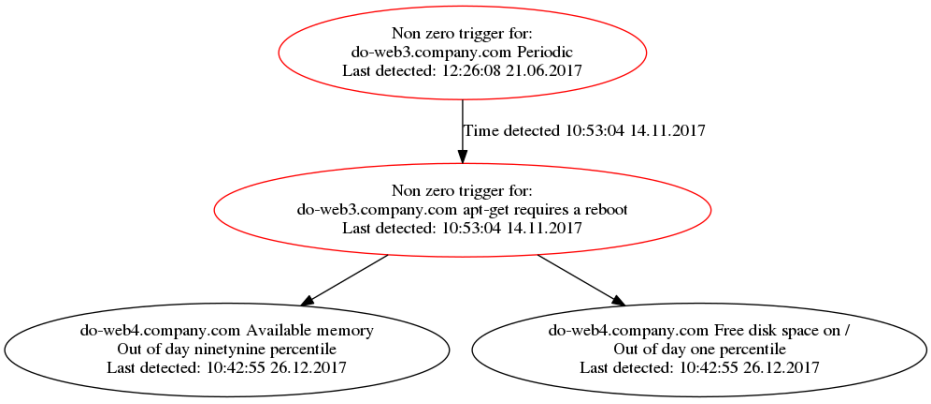


Figure 5.3: Thresholds comparison

In figure 5.3 we can see that one of the thresholds showed that `Free disk space on /` is lower than one percentile. In the real scenario the administrator just removed majority of the data from the server. It again depends on a real

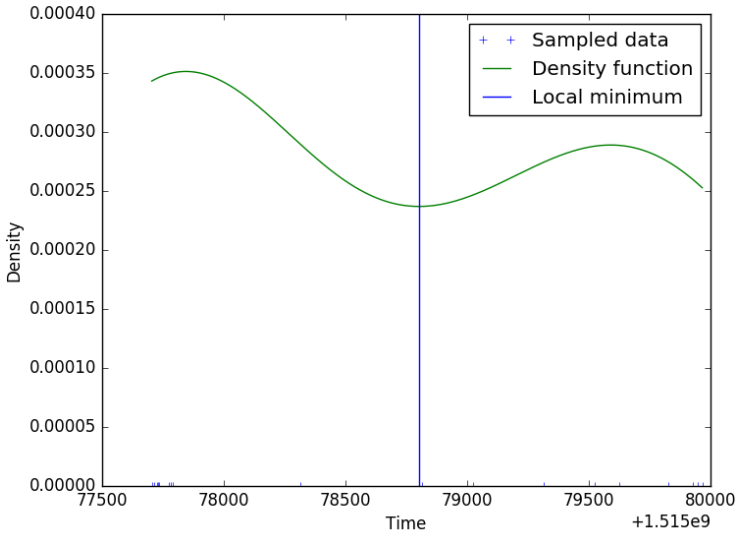


Figure 5.2: Density function estimation of two clusters with different spread distance of alerts

situation. If we consider that it could be a result of some malicious attack, we can hardly say that this is the fault of monitoring. We can clearly say that thresholds do not have one hundred percent accuracy.

To fix the issue, the application had to get more data from the *Zabbix* server. After a while, the threshold alert disappeared (figure 5.4) with new data collected from the server. That seemed to be a correct behaviour, because the administrator was warned about the sudden change in data series and could potentially react.

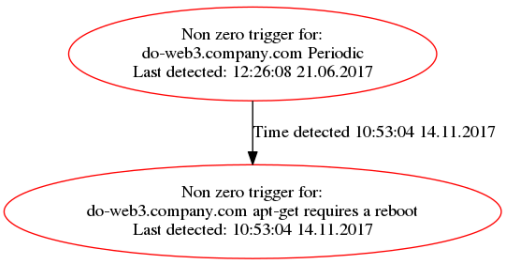


Figure 5.4: Threshold alerts have gone away with more data

This is the case of insufficient amount of data. The administrator should know about the issue and the thresholds should adjust themselves during a long period of time. For a better analysis, the administrator should be aware of maintenance setting and set it for the data item at their *Zabbix* server.

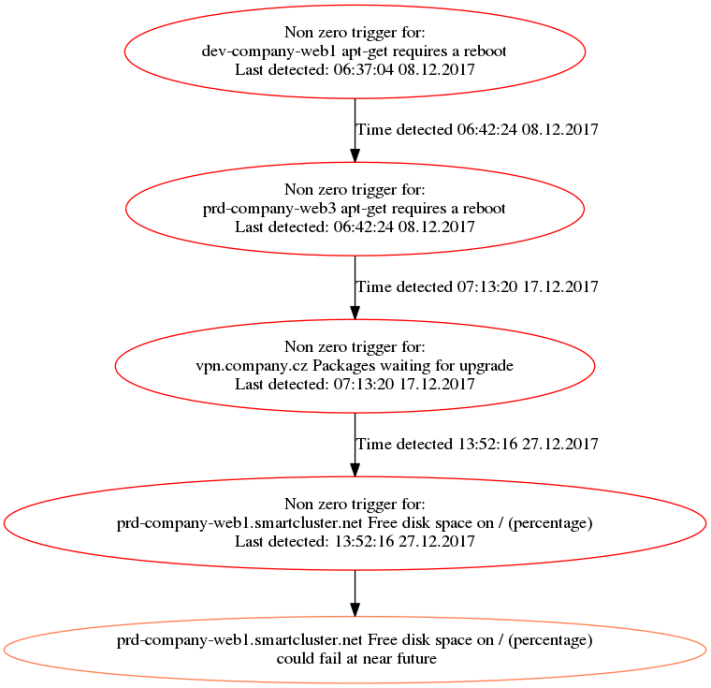


Figure 5.5: Alert on insufficient disk space and also predicted as failing

Items which are under maintenance are not used to compute thresholds.

5.3 Predictions

We always have to consider how much data we have to create for any prediction. Having data which mark future data correctly as an alert is essential for our solution. The problem we have encountered was that all the data from the past were marked as an error (figure 5.5). The classifier which marked future data as alerts had hard time to adjust to the fact that the data are correct again. Until we have collected sufficient amount of data, the classification failed to recognize that everything was alright again.

Now we can compare what happened if the alert was solved. Because all the data were marked as faulty, the prediction still marks the future as a fault (figure 5.6).

The solution was to remove classification and use thresholds only. This step had to be used only if there was a sufficient amount of data to create thresholds. Having small amount of data led to over sensitive thresholds.

Results were measured in another issue when the server slowly depleted all the available memory in local storage. We can see that the prediction had no problem to predict linear data and predicted alert in near future (figure

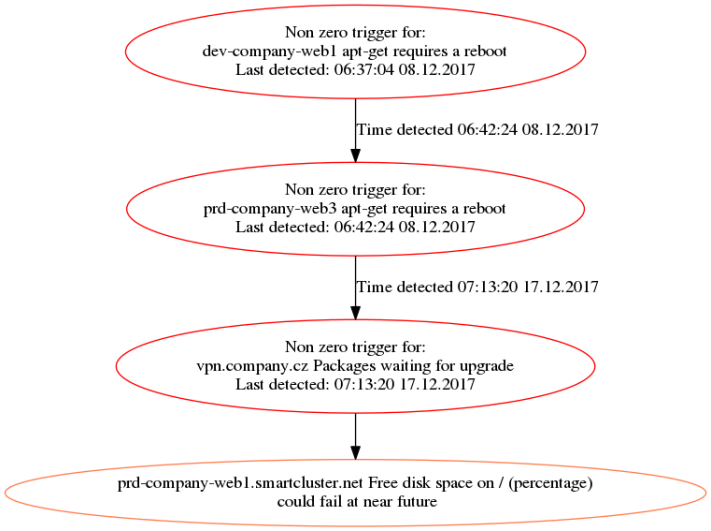


Figure 5.6: Alert was closed but prediction still thinks the future contains failure state

5.7). After a while we would be able to see normal threshold to create alert as well.

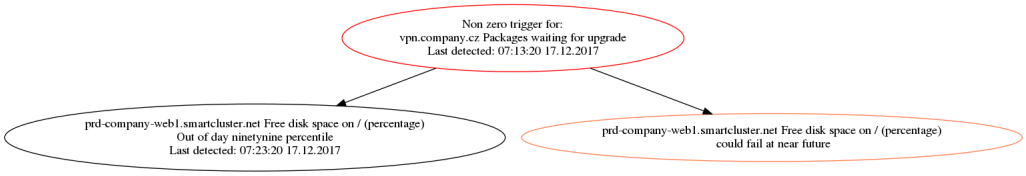


Figure 5.7: Prediction created alert and after a while, threshold created alert too

After removing data from the storage, both alerts disappeared. That is the correct result of using thresholds as classifiers. We have to mention again, that with insufficient data the issue would happen again.

Other methods are quite narrow. Sorting alarms to perform basic time based analysis did not leave any blind spot to observe any drawbacks. Thresholds proved to be a very reliable method to classify fault states. There are some common aspects which could always lead to failure of the used method. Those are:

- Not enough data
- Data marked incorrectly in time as fault data

- Naming results of analysis in the a way human expert can not understand

Those three aspects were the case in the whole testing of the application. On the other hand, the administrators welcomed the resulting application as something which can help them resolve the issues faster. In some cases the administrators were already using they own scripts to see thresholds and their were content that they can see their results in graphs with other analysis.

Conclusion

We have defined what should be the result of this thesis in the introduction with the help of three questions. Discovering possibilities in attempt to answer them was the main task of this thesis. The whole process began with exploring the domain of monitoring. That was done by constructing the case study (1.1) which examined small company dedicated to back end support for the information systems. By constructing the requirements that should help such a company, the application design was proposed. Resulting application should explore possibilities which are available for such a small company. The main functional requirements were stated as follows:

- Application has to provide the administrator with basic prediction of the future
- Application has to create root cause analysis

It seemed that these requirements should help administrators with their daily work and problem solving. Through exploration of other possibilities (1.2), it is more than obvious that there are already some functional solutions. We have explored *Vitrage* and *Dynatrace*, where *Dynatrace* exceeded all the expectations and proved to be a very useful tool for root cause analysis. There were obviously some reasons why the thesis hasn't stopped there and provided space for the exploration of all the features of *Dynatrace*. The main problem with *Dynatrace* was that it was considerably expensive to use. Also, it is patented and in the hands of the *Dynatrace* company. That is, of course, understandable and reasonable. Another problem was lack of functionality for predictions and the need to install new agent to the monitored information systems.

Our solution does not have any new data agent in the information system's servers. It strongly relies on data from monitoring, that was already deployed. The benefit of this approach is an easy installation and major drawback is no ability of estimating topology of systems. Both *Vitrage* and *Dynatrace* have

methods to analyze the whole topology and through that create an entity diagram later used in the root causes analysis. For example, failing router is the root cause of collapse, yet our solution would notice it only in form of aggregations or from well constructed trigger in the monitoring system.

Something which is not so much used in analyzed systems is prediction. There are some obvious reasons – to predict only short near future can lead to large computational overhead. We have explained that there is a solution in estimation of `LSTM` predictive model (3.3.2.4). Training such a model takes a long time, but predictions for short near future are reasonably fast with reasonably small error rate. Training had to be conducted on graphic card (`GeForce GTX 1060`) which is not a basic equipment of the monitoring server. Using `LSTM` networks proved to be very successful in terms of functionality which has been used in the thesis application. There are also some drawbacks. We have always assumed that there can not be any human expert based selection of data from monitoring. We used some features which helped with selecting the data (3.3.1). Standard deviation proved to be very useful. If data showed no variety over time, there was no need to estimate predictions.

The root cause analysis proved to be very difficult to do just from the data from the monitoring systems. We strongly relied on a variety of statistical features (4). Very important proved to be the estimation of threshold (4.2) which is also used as baselining in *Dynatrace*. Estimating threshold based on data gave a very simple classifier of fail states. There were a lot of difficulties in another feature – because of the use of a database storage for all the data collected from the monitoring servers, counting correlation proved to be very difficult and for large datasets even inapplicable. Data denormalization has to be considered. Very promising was the use of clustering which – in case of collapses – created large bubbles of alerts and the administrator could review them as a whole.

Now we have enough information to create answers to the questions from the introduction:

- What are the current options in monitoring?

  As the case study shows, company usually uses only direct monitoring without any further analysis. There are some more robust tools but they are expensive or not open source. For some customers, this is unacceptable.

- Can any system predict fail state of another system before it occurs?

  From the analysis of major tools, this is not a frequently used function. There are machine learning tools which can help with the task, but it is hard to implement them and they take a lot of computational power. If the monitoring system has that power and estimate models, prediction of short time in the future proved to be no problem.

- What are the relationships between fail states of a system?

  Estimating relationships between fail states is done by root cause analysis. This is covered very well by tools such as *Dynatrace* and partially covered by *Vitrage* (based on explicit templates). Estimating relationships only from data given by the monitoring systems proved to be very difficult. Using such a solution gives the administrator another responsibility to analyze resulting analysis well based on the inner knowledge of the system.

Root cause analysis proved to be very difficult to estimate based only on the given data. A solution would be to use a better agent and collect more of the system features (e.g. `strace` or code injections). This is already used by *Dynatrace* which can estimate root cause very well. If we consider only open source monitoring systems, we have none of these tools. That creates another large space for improvement in the available monitoring. To have such tools in open source domain can help improve small projects with small budgets.

We can observe in chapter *Testing of application* 5 what were the major problems with the implementation. There is a large space for improvement. We can notice that aggregation of alerts in large clusters proved to be a working concept, yet we have to add some heuristic which will divide alerts in clusters which are based on hours or days. Because we have used kernel density estimation 4.4.2 we need to keep in mind, that clusters could be distributed through long time periods. Using heuristic in form of sampling clusters into days can help the administrators to assess problems better. The whole solution works much better in case the cluster occurs in a short period of time. In that case it has much bigger informational value.

Another problem which was mentioned multiple times in this thesis are insufficient data. The solution is to download all data which can be relevant for the analysis. This is not an easy task and it could result in an overhead during communication with the monitoring server. A better solution would be to use heuristics or even ask user what he thinks is relevant for the analysis.

There is a large space for improvement in the currently available monitoring systems. If we omit paid or cloud solutions, there is no application which could estimate root cause or create prediction. This thesis proved that prediction is possible even from the data from current monitoring servers. It would be unwise not to implement such a solution.

We can clearly say the whole application has large space for improvement. There are a lot of methods which could help create better analysis. One of the methods could be the creation of aggregation rules based on computed clusters. Another direction which needs to be explored is the human expert feedback which can help with the overall accuracy of the application. Interaction with the application is mainly done through the monitoring system which can be filled with data of no informational value. Collecting feedback from the administrators can improve the whole solution.

Bibliography

[1] Stats, I. L. Internet Users. 2017. Available from: http://www.internetlivestats.com/internet-users/

[2] Ligus, S. *Effective Monitoring and Alerting: For Web Operations*. O'Reilly Media, 2012, ISBN 9781449333485. Available from: https://books.google.cz/books?id=cTqbdNzH7UUC

[3] Timescale Community. TimescaleDB: SQL made scalable for time-series data. 2016. Available from: http://www.timescale.com/papers/timescaledb.pdf

[4] Berners-Lee, T. The first website. 1990. Available from: http://info.cern.ch/hypertext/WWW/TheProject.html

[5] Vidim, J. Co prozradil Seznam.cz na SEO restartu 2016. 2016. Available from: https://www.evisions.cz/blog-2016-06-13-co-prozradil-seznam-cz-na-seo-restartu-2016/

[6] Kasík, P. Výpadek Googlu šokoval Česko. Nebyl to útok hackerů, ale chyba při údržbě. 2016. Available from: http://technet.idnes.cz/google-vypadek-priciny-dopady-dkv-/sw_internet.aspx?c=A161123_114509_sw_internet_vse

[7] Zabbix LLC. Zabbix – The Enterprise-class Monitoring Solution for Everyone. Available from: http://www.zabbix.com/

[8] The Icinga Project. Icinga2. Available from: https://www.icinga.com/

[9] Nagios. The Industry Standard In IT Infrastructure Monitoring. Available from: https://www.nagios.org/

[10] Olups, R. *Zabbix Network Monitoring*. Packt Publishing, 2016, ISBN 9781782161295. Available from: https://books.google.cz/books?id=xgjVDQAAQBAJ

[11] Davis, J.; Daniels, K. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. O'Reilly Media, 2016, ISBN 9781491926437. Available from: https://books.google.cz/books?id=6e5FDAAAQBAJ

[12] Gibson, D. *Managing Risk in Information Systems*. Jones & Bartlett Learning information systems security & assurance series, Jones & Bartlett Learning, LLC, 2014, ISBN 9781284055962. Available from: https://books.google.cz/books?id=5lktBAAAQBAJ

[13] OpsGenie. OpsGenie – Don't let incidents bring you down. Available from: https://www.opsgenie.com/

[14] Pingdom Solar Winds. Pingdom. Available from: https://www.pingdom.com/

[15] Statusdroid s.r.o. Statusdroid. Available from: https://www.statusdroid.com/

[16] Pivotal. RabbitMQ. Available from: https://www.rabbitmq.com/

[17] *Root Cause Analysis: The Core of Problem Solving and Corrective Action*. ASQ Quality Press, 2009, ISBN 9780873897648. Available from: https://books.google.cz/books?id=gcqWSkrEFasC

[18] OpenStack Vitrage Project. 2017. Available from: https://wiki.openstack.org/wiki/Vitrage

[19] LLC, D. 2017. Available from: https://www.dynatrace.com/capabilities/root-cause-analysis/

[20] LLC, D. 2017. Available from: https://www.dynatrace.com/platform/artificial-intelligence/secret-sauce/

[21] OpenStack Foundation. Available from: https://www.youtube.com/watch?v=XL62g1WHH8g&t=1963s

[22] OpenStack Vitrage Project. 2017. Available from: https://docs.openstack.org/vitrage/latest/contributor/alarm-severity-config.html

[23] Grönholm, A. APScheduler. 2017. Available from: http://apscheduler.readthedocs.io/en/latest/userguide.html

[24] Jaynes, E. T. Information Theory and Statistical Mechanics. *Phys. Rev.*, volume 106, May 1957: pp. 620–630, doi:10.1103/PhysRev.106.620. Available from: https://link.aps.org/doi/10.1103/PhysRev.106.620

[25] Box, G.; Jenkins, G. *Time Series Analysis: Forecasting and Control, Revised Ed.* Holden-Day series in time series analysis, Holden-Day, 1976, 158-179 pp. Available from: https://www.jstor.org/stable/2346997

[26] Dickey, D. A.; Fuller, W. A. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica: Journal of the Econometric Society*, 1981: pp. 1057–1072.

[27] Ford, C. Understanding Q-Q Plots. 2017. Available from: http://data.library.virginia.edu/understanding-q-q-plots/

[28] Parzen, E.; Tanabe, K.; et al. *Selected Papers of Hirotugu Akaike.* Springer Series in Statistics, Springer New York, 2012, ISBN 9781461216940. Available from: https://books.google.cz/books?id=9cXgBwAAQBAJ

[29] Ludovici, B. *A Mean-square Error Analysis of Several Demodulation Systems.* Department of Electrical Engineering, Stanford University., 1956. Available from: https://books.google.cz/books?id=9sAUAAAAIAAJ

[30] Bengio, Y.; Simard, P.; et al. Learning Long-term Dependencies with Gradient Descent is Difficult. *Trans. Neur. Netw.*, volume 5, no. 2, Mar. 1994: pp. 157–166, ISSN 1045-9227, doi:10.1109/72.279181. Available from: http://dx.doi.org/10.1109/72.279181

[31] Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.*, volume 9, no. 8, Nov. 1997: pp. 1735–1780, ISSN 0899-7667, doi:10.1162/neco.1997.9.8.1735. Available from: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[32] TensorFlow. Recurrent Neural Networks. 2017. Available from: https://www.tensorflow.org/tutorials/recurrent#loss_function

[33] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2017, ISBN 9781491962268. Available from: https://books.google.cz/books?id=khpYDgAAQBAJ

[34] Commons, W. Unix Timestamp. Available from: https://en.wikipedia.org/wiki/Unix_time

[35] Boslaugh, D. *When Computers Went to Sea: The Digitization of the United States Navy.* Perspectives Series, Wiley, 2003, ISBN 9780471472209. Available from: https://books.google.cz/books?id=Mi8MhzheOokC

[36] Mattern, F. Logical time. Available from: http://www.vs.inf.ethz.ch/publ/papers/logtime.pdf

[37] Silverman, B. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability, Taylor & Francis, 1986, ISBN 9780412246203. Available from: https://books.google.cz/books?id=e-xsrjsL7WkC

[38] Engelhardt, H.; Kohler, H.; et al. *Causal Analysis in Population Studies: Concepts, Methods, Applications*. The Springer Series on Demographic Methods and Population Analysis, Springer Netherlands, 2009, ISBN 9781402099670. Available from: https://books.google.cz/books?id=ArfD-y8lvaQC

[39] Härdle, W.; Müller, M.; et al. *Nonparametric and Semiparametric Models*. Springer Series in Statistics, Springer Berlin Heidelberg, 2012, ISBN 9783642171468. Available from: https://books.google.cz/books?id=wqX7CAAAQBAJ

[40] Group, T. P. G. D. Aggregate Functions. 2017. Available from: https://www.postgresql.org/docs/9.4/static/functions-aggregate.html

[41] Postgresql, W. Bitmap Indexes. 2017. Available from: https://wiki.postgresql.org/wiki/Bitmap_Indexes

[42] Fawcett, T. Avoiding Common Mistakes with Time Series. January 2015. Available from: https://svds.com/avoiding-common-mistakes-with-time-series/

[43] Vigen, T. *Spurious Correlations*. Hachette Books, 2015, ISBN 9780316339452. Available from: https://books.google.cz/books?id=bmsVBgAAQBAJ

[44] community, T. S. Signal Detrend. 2017. Available from: https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.detrend.html

[45] Michalove, A. Amdahl's Law. 2017. Available from: https://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html

Appendix A

Acronyms

SLA Service Level Agreement

LSTM Long Short Term Memory

AIC Akaike Information Criterion

ARIMA Autoregressive Integrated Moving Average

PCA Principal Component Analysis

RCA Root cause Analysis

MTBF Mean Time Between Failures

MISE Mean Integrated Square Error

AMISE Asymptotic Mean Integrated Squared Error

AI Artificial Intelligence

VPN Virtual Private Network

Appendix B

Repository description

Diploma thesis repository

This repository servers as storage space for the whole thesis.

Basic topics covered in this thesis:

- collecting data from different monitoring servers (only Zabbix currently stable)
- creating simple root cause analysis based on:
  - time
  - correlation
  - clusters
  - dynamic thresholds
- estimating future of monitored items based on the following algorithm:
  - `LSTM` networks on `CUDA` (you need to use `CUDA` device or take immense amounts of time with `CPU` TensorFlow version)
  - Prophet from Facebook incubator (using `SARIMA` and `ARIMA`)

Folders list:

- `./text/` – contains all of the publications from the thesis
- `./text/main` – contains `DP_Beranek_Martin_2017.pdf` which is the main text of the work
- `./apps/` – contains all the applications for the resulting application
- `./task/` – contains only the written task from school's information system
- `./resources/` – contains all the examples and programming articles I needed to deal with the task

Application use

If you found yourself in the main directory of this repository, use:

```
cd apps;
```

Testing

The whole application stack is quite complicated. If you need the app use for testing, use docker-compose:

Starting the app:

```
docker-compose -p predmon up -d;
```

Application will probably fail due to some configuration issues, that's why you need to change some small facts.

To just **test** the app in:

```
config.sh
```

paste somewhere:

```
tests="True";
```

Results will show up at: http://localhost:8080/

Collector settings

At `./credentials/zabbix.yml` you can set up the configuration to access zabbix server. Do not forget to add the correct user with proper permissions. Zabbix `API` permissions can be tricky.

At `./collectors/zabbix_collect/configure/hostnames.yml` you can specify with `Unix` filenames patterns showing what machines should be monitored and included in the analysis. Keep in mind that to monitor every machine (expression '`*`') in large topology could take forever to analyse. It's not recommended to do this. Application was succesfully tested for ten machines at small laptop (`Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz` with `4 GB` of `RAM`).

Credentials

All other credentials related settings you can find in the `credentials` folder.

Configuration

At `configuration/basics.yml` you can find the following settings:

- `maximum_prediction_graphs`: how many `png` files of predictions should be kept in `./static/classify/graphs/`
- `maximum_hows_graphs`: how many png files of cause analysis should be kept in `./static/hows/graphs/`
- `prediction_alg`: which algorithm for prediction should be used, currently available `lstm`, `arima` and `Prophet`
- `predict_items_delay`: states how much collection times per item will the process wait until it starts predicting, for example if sampling of an item is half a minute and `predict_items_delay` is set to 4, prediction will happen on every second minute
- `collect_delay`: states the interval of how often is monitoring server being asked about data items, it does not mean it will collect the items in this interval, because data items have their own interval from the monitoring system
- `disabled_features`: is a list of features which can be disabled, you can put there: correlations, out_of_border_items

Application services

Each service states what inputs does it need and what are the ouputs. Also, readiness is provided from 1 (not ready) to 10 (would put it in production and trust it with my life). For further reading open `./apps/docker-compose.yml`.

collect

Collects data from monitoring servers into local database. Outputs data into *rabbitmq* and *postgres*.

Ready 6/10

predict

Get data from `collect` and create prediction based on the algorithm from `configuration/basics.yml`. Result doesn't provide any information on whether anything in the future will fail, that's why the result is sent to `classify`.

Ready 4/10

classify

Gets data from `predict` sent from predictions and classify data for bad states. Also the graph is constructed into `static/classify/graphs`.

Ready 5/10

hows

Create root cause analysis from data saved in *postgres* and *rabbitmq*. The result is provided in `static/hows/graphs`.

Ready 5/10

frontend

Creates frontend application on http://localhost:8080/. The app simply reads what is in static folders for predictions and root cause analysis and displays it in browser.

Ready 7/10

API

Creates API on 5000 port. It just makes a lists of predictions and cause analysis. Access to pictures have to be done via frontend, `API` is not ready at all.

Ready 2/10

Cleaner

Reads directories:

- "static/classify/graphs"
- "static/hows/graphs"

and removes old data based on counters given in `configuration/basics.yml`.

Ready 9/10

Git Blame & contact and affiliation

To spare you from writting command `git blame` all the errors can be sent to martin.beranek112@gmail.com. You can also reach me at the company *Lukapo*.

Whole thesis was tested on live data from company *Lukapo* and their customers.

You can also find this thesis at *CTU* library in Prague.

Appendix C

Folder list on attached USB drive

- `text` ..... contains all of the publications from the thesis
  - `main` ..... contains main text of the work
    - `DP_Beranek_Martin_2017.pdf` .... main written text of this thesis
- `apps` ..... contains all the applications for the resulting application
- `task` .... contains only the written task from school's information system
- `resources` .. contains all the examples and programming articles I needed to deal with the task