České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Udržal Vojtěch

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Dohledové nástroje telekomunikačního operátora

Pokyny pro vypracování:

Cílem diplomové práce je vytvořit monitorovací systém pro doménu telekomunikačního operátora. Aplikace by měla sledovat a analyzovat provoz datových toků na mediation systému a detekovat anomality. Dále bude aplikace monitorovat chod a funkčnost big data nástrojů, zobrazovat jejich stav a ve vhodných případech upozorňovat obsluhu.

Pokyny:
1) Seznamte se s nástroji pro monitorování síťového provozu a big data nástroji používanými telekomunikačním operátorem i obecně.
2) Navrhněte a implementujte nový open source framework pro monitorování síťového provozu.
3) Navrhněte a implementujte monitorovací moduly pro vybrané big data nástroje (např. Zookeeper, Spark, Cassandra).
4) Vytvořte webovou aplikaci pro testováni a demonstraci implementovaného frameworku.
5) Vyhodnoťte funkčnost aplikace a implementovaných modulů a porovnejte je se současným řešením.

Seznam odborné literatury:

[1] Cassandra: The Definitive Guide: Distributed Data at Web Scale Jeff Carpenter, Eben Hewitt ISBN-10: 1491933666
[2] Learning Spark: Lightning-Fast Big Data Analysis Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia ISBN-10: 1449358624

Vedoucí: Ing. Ondřej Macháček

Platnost zadání do konce letního semestru 2017/2018

[redacted]

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

[redacted]

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 6.2.2017

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Master's thesis

Telecommunication provider's monitoring tools

Bc. Vojtěch Udržal

Supervisor: Ing. Ondřej Macháček

24th May 2017

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Macháček and his colleagues Ing. Václav Surovec and Peter Macháčik for giving me the opportunity to work on this thesis and for their support during the development. I would also like to express many thanks to my partner for her great support and understanding. Last but not least, I would like to thank my parents for overall support during my studies.

Declaration

Citation of this thesis

Udržal, Vojtěch. *Telecommunication provider's monitoring tools.* Master's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2017.

Abstrakt

Cílem této diplomové práce je návrh a implementace monitorovací aplikace pro telekomunikačního operátora. Práce nejdříve představí infrastrukturu, systém pro mediaci a nástroje pro monitorování. Poté se zabývá analýzou systému pro mediaci a rozebírá nové možnosti pro jeho monitorování. V další části je popsán návrh, implementace a vyhodnocení nové monitorovací aplikace. Dále je v práci popsáno několik big data nástroju, které operátor používá, a pro jeden z nich je vytvořen monitorovací modul.

Klíčová slova mobilní operátor, monitorování, mediace, telekomunikace, Python, JavaScript, MongoDB, ZooKeeper

Abstract

The goal of the thesis is to design and implement a new monitoring application for telecommunication provider. At first, the thesis briefly introduces provider's infrastructure, mediation system and monitoring tools. Next, it analyzes mediation system and discusses new ways of its monitoring. At last, a design and implementation of a new monitoring application is described and evaluated. Additionally, several big data tools used by the provider are discussed and a monitoring module for one big data tool is implemented.

Keywords monitoring, mediation, telecommunication, provider, Python, JavaScript, MongoDB, ZooKeeper

Contents

List of Figures

List of Tables

Introduction

In today's world of information technologies, systems are becoming more complex and ubiquitous and our every day lives depend on them. Nowadays, people are expecting online service to be available immediately, everywhere and always. Any outage of service poses a threat as it can have significant impact on company's revenue and reputation. No software can be guaranteed to contain zero bugs and have uptime of 100% and even if it could, IT systems are still influenced by real world events as they must be at least physically hosted in some data center. It is even more the case of IT companies and enterprises whose businesses are closer to real world, and telecommunication service providers (TSP) are one of them.

Telecommunication providers can be considered as core elements or backbone of modern society. Except for traditional postal service, communication between 2 entities on distance longer than few meters depends on their services. Outage of their systems is therefore even more serious and sensitive issue as people take phone and internet services for granted. To prevent and minimize failures, every system and its metrics must be carefully monitored.

First of all, this thesis will introduce mediation system, describe its role in TSP's infrastructure, analyze its metrics and currently used monitoring techniques.

Afterwards, the thesis will focus on development of better mediation system's monitoring algorithm and application and the whole development process will be discussed.

This thesis was written and the final application was developed for Competence and Delivery Center of one of the 3 largest telecommunication providers in the Czech Republic. This application was developed in accordance with their needs and they will be referred to as stakeholders.

Several chapters are also dedicated to big data tools used by the provider and their monitoring. However, this part of thesis and application is not meant to be used by the stakeholders and is considered to be a proof-of-concept.

Chapter 1

Mediation system monitoring analysis

In this chapter I will introduce the problem domain in detail, describe mediation system, currently used monitoring tools and its flaws. Moreover, I will specify requirements that were considered as important for the new monitoring application.

1.1 Mediation system

In general, mediation system's task is to connect heterogeneous systems by formatting and modifying transported data to meet each system's expectations. If I were to find a similar system from software engineering field, which is my main area of expertise, I would compare it to Apache Camel [1].

Simply put, an administrator of mediation system defines a set of routes, extractions and transformations between heterogeneous systems so that it complies with the company's needs and business requirements and the mediation enables systems to communicate with each other.

To understand a function of mediation system in telecommunication provider's field, one first needs to understand the base principle of their network. The network consists of thousands or tens of thousands base radio stations, telephone exchanges and other network elements. Each user (provider's client) must be connected to one of these elements if they want to make a call, send a text message or use any other service. The element is keeping metadata information of all service usages that happened on it. The metadata is called *call detail record* (CDR) and contains information about calling and called numbers, duration of a call and so on. The term CDR is also used for text messages and other services usage and similar information is tracked [2]. It is important to note that it does not contain an actual content of a call or a text message. Each element has its own limited memory where this metadata information is stored. The metadata are submitted to the mediation system in batches once accumulated data's size exceeds specified threshold or a timeout is reached. Then, the mediation system decodes the batch and sends relevant parts to subsequent systems in proper format for further processing. The subsequent systems are typically billing, fraud detection, data warehousing, data analysis, legal investigation and many more.

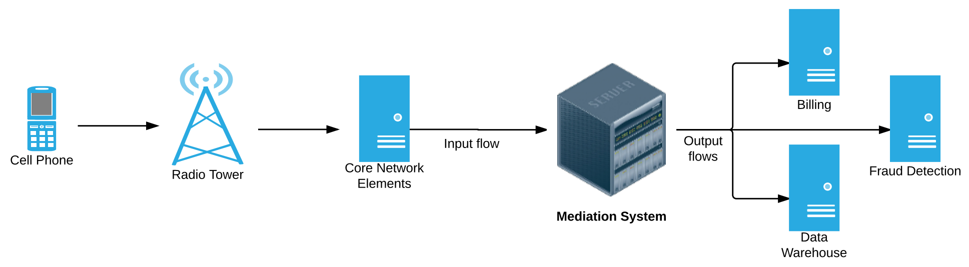


Figure 1.1: Architecture Overview

Each of these systems is important, yet the billing system is one of the really business critical ones. Its task is to record provided services to each client based on the received metadata. If the metadata are not delivered to this system, the information about a call or a text message cannot be properly accounted and billed and the telecommunication provider loses money.

1.2 Mediation system in provider's context

The mediation system at provider's network is called iBMD (international Billing Mediation Device) and is an international platform that is responsible for collecting and processing telecommunication data from four European countries: the Czech Republic, Germany, Austria, the Netherlands. The data include information about every phone call, sent SMS or MMS, internet usage or any other service usage offered to public or private clients and businesses by the provider. iBMD is managed by Competence and delivery center Prague (CDCP). It deals with a lot of interesting, but also very sensitive and valuable data, where ten billion of CDRs are processed every day. Therefore, data security, correctness, availability and completeness is a must in this area.

Essentially, the mediation system has input and output flows of data. The input and output data flows are hierarchically split into several categories. On the highest level it is divided by a country name. Then, in each country there are about 40 categories of services, also called LoBs (Line of Business). They represent type of service in the network: for example there is SMS, GSM or MMS LoB. Then, each LoB has various number of incoming data flows coming from network elements (also called inputs) and outgoing data flows (also called forwards) [2]. The incoming data flows are the metadata collected from the network elements and the outgoing data flows are decoded and transformed data going to subsequent IT systems. An output flow can also be called as a forward flow since it represents a connection between input and output.

To best illustrate the hierarchy of flows, I will describe the Czech Republic's SMS LoB in more detail. The provider has the Czech Republic divided into four regions and each region has one main city. The cities are Prague, České Budějovice, Hradec Králové and Brno. Each city then has four Mobile Switching Servers where metadata from other network elements are collected and from where the mediation system collects them. In Prague, names of Mobile Switching Servers and thus names of input flows are MSSPRC1A, MSSPRC1B, MSSPRC1C and MSSPRC1D. Likewise, there are input flows named MSSBRN1A from Brno, MSSHRK1A from Hradec Králové and MSS-CEB1A from České Budějovice. There are sixteen input flows in the Czech SMS LoB and each one is transformed and forwarded to twelve subsequent systems. That means there are 192 forward flows in the SMS LoB (sixteen inputs times twelve outputs). In total, including all countries, LoBs, inputs and forwards, there are currently over 4500 flows in the mediation system.

1.3 Monitoring

As was described in previous sections, the mediation system is critical platform for every telecommunication provider. Even partial outage of input flow or the platform itself can have impact on the provider's revenue and credibility. Therefore, it is in the provider's best interest to have a reliable and effective monitoring tool detecting any anomaly in the system. The outages can be caused by various events. The common causes are upgrades of network elements to some new or unsupported protocol or format, which typically result in problems on input flow. Likewise, changes of mediation configuration or transformation rules are common causes of outage on forward flows.

A prerequisite for monitoring of any system is to have access to metrics of the system. In this case, it means to have information about how much data is passing through each flow in the mediation system. Extracting this kind of information out of the mediation was already implemented by the provider and I was granted access to it, including historical data for past months. The data do not consist of CDRs or other network activity records, but contains only sizes of collected or sent batches to or from the mediation. For input flow, it contains name of country, name of LoB, name of network element, time the data was received and size of a batch. For forward flow it also contains name of target system. In a way, the source data for a mediation monitoring can be understood as metadata of metadata. While metadata collected from one network element in one day can easily exceed hundreds of megabytes, storing the size takes only few bytes per batch and still gives great insight into traffic level on the mediation system. Nevertheless, the monitoring data of forwards for one country and one day still takes about 30MB and contains over six hundred thousand of records (transmitted batches). The input data per country and day are about 3MB large and contains seventy-four thousand of records. That means seventy-four thousands of batches collected by the mediation system.

1.4 Currently used monitoring

The mediation system is storing size of each processed file to Oracle database where it is accessed by other systems, including currently used monitoring tool. However, that tools is quite simple and therefore insufficient. Each input or forward flow has an attribute which specifies maximum interval, in which at least one byte of data must be transmitted via that flow. If no data is transmitted in the specified interval, alarm is triggered. The disadvantage of this is obvious. Even if the traffic on some flow is only 1% of what it is usually, the alarm will not be triggered as long as at least one byte is transmitted in the time limit. Every transmission of a byte resets the limit and the countdown starts from the beginning. There were already several cases when a customer was not billed for some used services because the mediation system was misconfigured and did not forward some data as it should have. The application is written in PHP and only contains an input field for each flow to set the maximum interval attribute. It does not give any insight of how much data is passing through the mediation system, nor gives the possibility to look at historical data. The new monitoring tool should be designed and implemented in a way that the mentioned problems will be eliminated and will provide better insight into the mediation system traffic.

1.5 Analysis of flows

In order to come up with a better algorithm for monitoring the mediation system and detecting outages, it is necessary to first look at, examine and analyze the flows that are going to be monitored. As mentioned earlier, there is about 4500 flows in the mediation system. To simplify the text, I will choose few flows with different characteristics that will be used as examples. Usually flows of one LoB have similar features, but there are few exceptions.

The first thing one notices when looking at the data is, that different flows have different period in which they are downloaded to mediation system or sent to subsequent systems. It is caused by buffers at the network elements, where the data is stored until a certain size is reached or time period elapsed and only after that it is sent to the mediation system. In figure 1.2 is depicted MSSBRN1A flow of GSM LoB with five minute ticks period on the x axis.

Such settings is too fine-grained for the mentioned data flow, causing the flow level to be very unstable and therefore making any further analysis impossible.

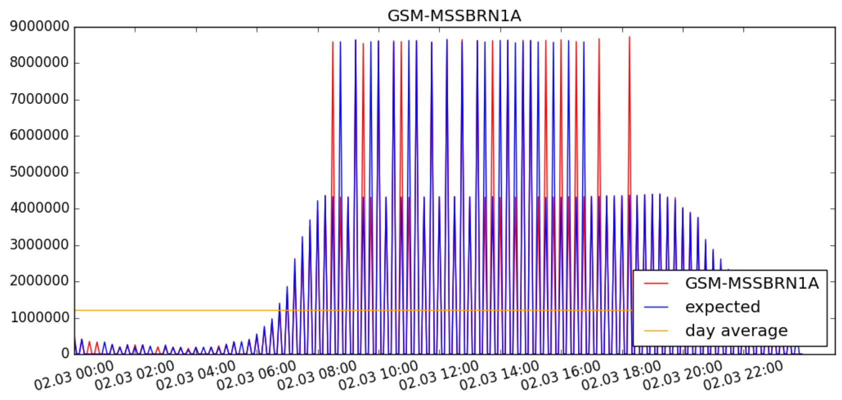


Figure 1.2: GSM MSSBRN1A - Thu Mar 3, 5 minutes granularity

1.5.1 Data smoothing

Several preprocessing techniques can be applied on the data to enable further analysis. First one I tried was to smooth the data. There are numerous ways how to smooth a data and in this section I would like to introduce few techniques that I tried.

1.5.1.1 Running mean

First of them was running mean. It is called running mean because it uses window of fixed size n and uses data points inside of the window to calculate mean. As the computation advances to compute mean of a following data point, the window moves as well. When calculating simple running mean, the data point x_i being smoothed is in the middle of the window and the mean is calculated from points between $x_{i-n/2}$ and $x_{i+n/2}$. The problem with this approach is that it does not meet the requirements of the use case. The application will need to analyze the data as it comes in a stream and only the previous data points will be available at the time the analysis is ran. This method would be feasible for showing historical data, but I thought that it is better to use one algorithm for both, showing historical data and actual real time analysis to keep things simple, consistent and predictable.

Therefore, I looked into cumulative mean next. The cumulative mean is using window which has only points preceding the data point being smoothed. In other words, when smoothing point x_i, data points in the window are between x_{i-n} and x_i. The outcome of this method is highly depending on a configuration of how many previous data points should be considered. If the window is small, the filter is more sensitive to local anomalies, as seen in

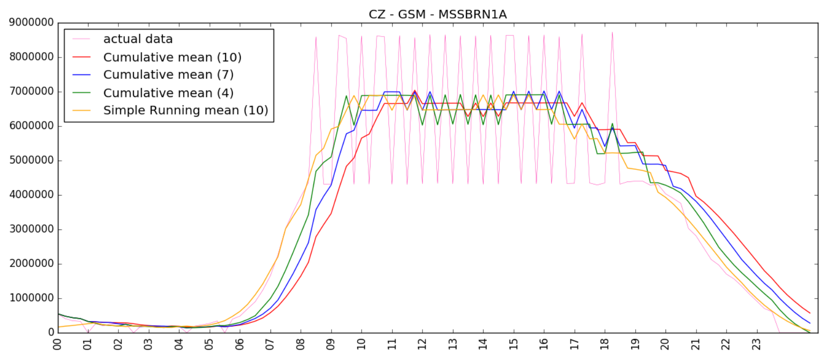


Figure 1.3: Running mean

figure 1.3 around 13:00. The more the size of a window is increased, the more it is resistant to local anomalies. However, with increased window size the filter gets also more resistant to trend changes. That can be seen when the traffic increases in the forenoon or decreases in the afternoon. The smoothed data by simple running mean (10) is delayed by about one hour, if not more.

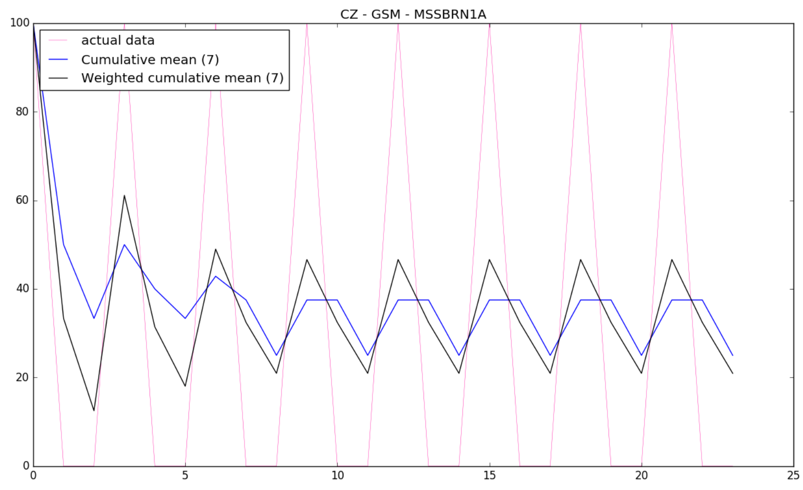


Figure 1.4: Weighted cumulative running mean

To remove the lag from cumulative running mean and make it reflect a trend faster, I tried weighted cumulative running mean. The idea behind this is to assign more importance to values closer to a point that is being smoothed. The results improved in desired way, but still were still not sufficient. Also, for a data set with longer period the weighted mean is less stable than cumulative mean, as shown in figure 1.4

I also tried calculating running median, but the results turned out to be even worse than running mean.

1.5.1.2 Savitzky–Golay filter

Furthermore, I tried another smoothing method called Savitzky–Golay filter. It was first published in 1964 in [3]. Even though this filter's window contains half of the preceding data points and half of succeeding data points - and therefore is not feasible for the use case - I was very surprised by the smoothed results it produces that I have decided to mention it in this text as well. The Savitzky–Golay filter expects two parameters, a polynom's order d and window's size n. When calculating the smoothed value of point x_i, the filter uses linear least square method to find optimal coefficients of a polynom with order d to most fit data points between $x_{i-n/2}$ and $x_{i+n/2}$. The smoothed value of x_i (middle data point of the window) is found by using the polynom with calculated coefficients. The Savitzky-Golay filter is widely used smoothing method and is one of the top ten most cited articles ever published in Analytic Chemistry journal [4]. Readers looking for more detail explanation of the filter I would recommend article [5].

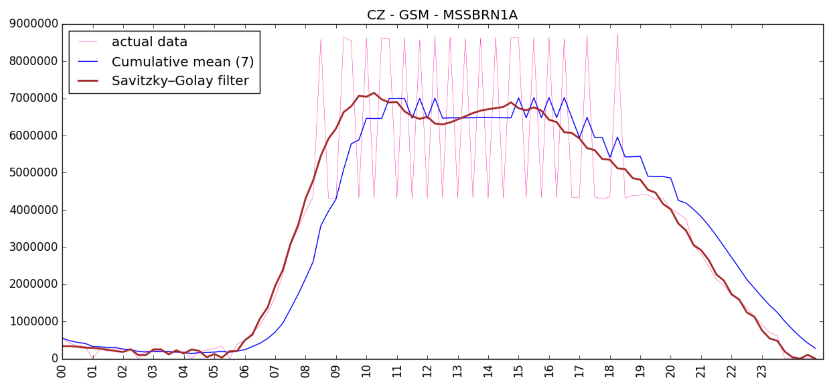


Figure 1.5: Savitzky–Golay filter

1.5.2 Increased tick size

Out of the previously mentioned methods of smoothing dataset into a more stable flow, none would completely satisfy my needs. Furthermore, any smoothing of data removes some information or anomaly which might turn out important later. Therefore, I have decided that the best I could do is to increase the size of ticks of the dataset and thus remove any high frequencies caused by buffers and periodical file submissions. The length of a tick will be further named as a granularity. The MSSBRN1A flow of GSM LoB is shown in figure with granularity of 60 minutes 1.6. This level of granularity means that in case of outage on this flow, the monitoring system will detect it no later than in 60 minutes. While this may sound still like a too long time, the currently used monitoring tool has for this flow a timeout interval of 360 minutes.

While examining flows, it quickly turned out that the minimal but acceptable granularity can range from 15 minutes to 1440 minutes (one day). Also, the granularity can change over time due to changes on preceding network elements to even less than 15 minutes. For these reasons, the new monitoring application will need to have an interface which will allow users to examine, prototype and set the right granularity for each flow.

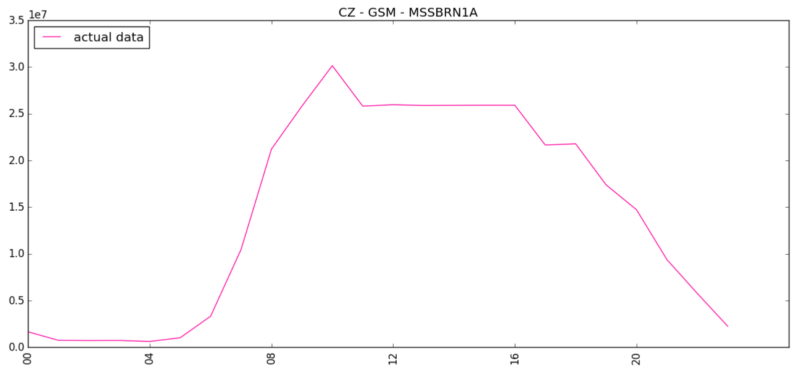


Figure 1.6: Increased granularity to 60 minutes

1.5.3 Traffic changes and trends

From the beginning it was made apparent by the stakeholders that there are hours or days when the level of traffic is exceptionally different from its usual level. The new monitoring tool should be aware of such a possibility and try to reflect it as much as possible. In this section I would like to analyze and describe unusual changes in traffic so I can later design appropriate algorithm. It comes as a no surprise that the root cause for any irregularities of a traffic are national holidays, weekends and special events with nation-wide impact.

First of all I looked into a regular week with no national holidays or known events. It was found that most of the flows have smaller traffic on weekends than during work day. For few flows, such as MMS LoB, the traffic can be just slightly lower - by about 8%. I assume it is caused by the fact that sending MMS is not strictly a work day activity and therefore is equally popular on weekends when people, for example, share their activities in free time. On the other hand, the weekend traffic of ICG - CENTREX01 is smaller by 97% than the traffic during work days. A whole week traffic is shown in figure 1.7. ICG is LoB for collecting data from VOIP land lines of business customers

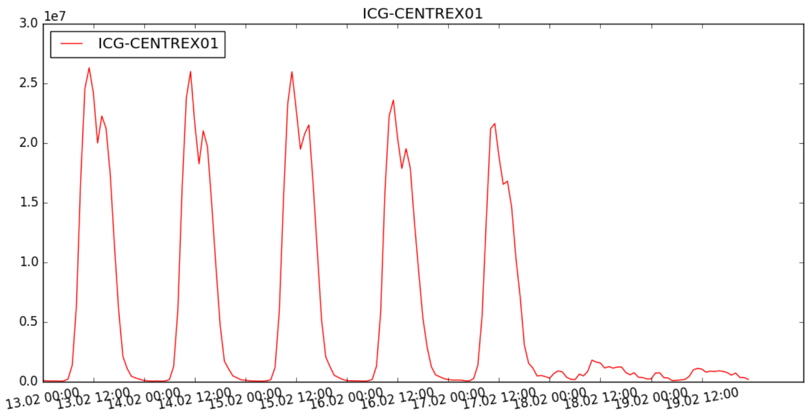


Figure 1.7: ICG - CENTREX01 - week traffic

of a company acquired by the provider several years ago, so the dramatic decrease on weekends is understandable. The GSM - MSSBRN1A has 50% smaller traffic on weekend than on week days. Based on these findings it was concluded, that weekdays and weekends will need to be handled independently.

I focused on national holidays next. The way how traffic behaves on national holidays quite depends on other factors, such as type of the holiday and day of a week. If the holiday day was close to a weekend, such as Friday 28th of October 2016, the traffic seemed to be lower than when the holiday was on Wednesday 28th of September 2016. However, there is not enough of data to make any reliable conclusion about trends of traffic on national holidays. It is safe to say though, that the traffic on national holidays is very similar to the traffic on weekends. There also appeared an interesting occurrence around national holiday on 17th of November 2016. That day is national holiday, however, it has also noticeable impact on surrounding days, 16th and 18th of November. The traffic on these days was about 35% lower than usual and following weekend was also influenced 1.8. The reason behind this might be prolonged holiday, when many people took vacation from 16th to 20th of November. Yet, the significance is surprising and worth mentioning.

As one could expect, the traffic around Christmas Holidays and New Years is also very unstable. As shown in figure 1.9, the traffic (or number of sent text messages in this particular case) on 24th of December is about twice as high as on other weekends or national holidays. On the other hand, the whole following week after Christmas is noticeably less busy due to the fact that most people take vacation and communicate less. The Christmas Holidays are ended by New Years Eve, where the traffic sky rockets around midnight as people send New Year wishes. Surprisingly, the traffic does not reach the same level as on Christmas Day and quickly diminishes.

Another request from the stakeholders was that the monitoring tool follows

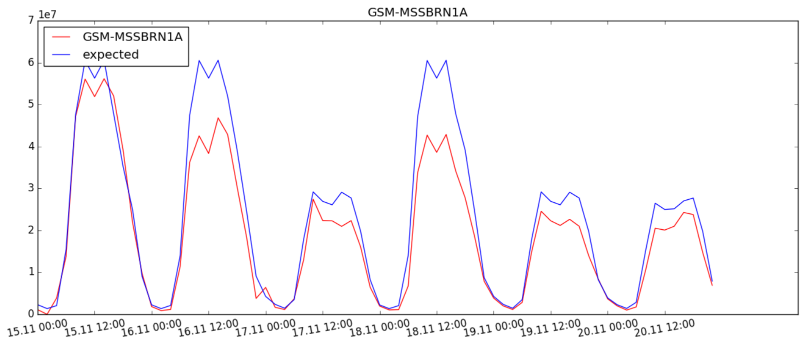


Figure 1.8: ICG - MSSBRN1A - traffic around 17th November

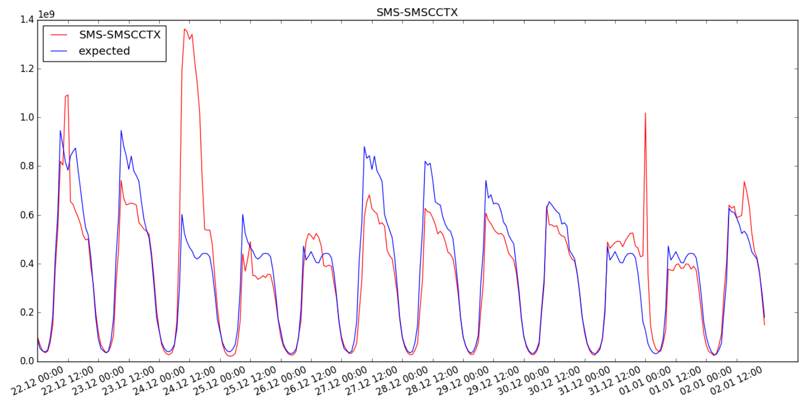


Figure 1.9: SMS - MSSBRN1A - traffic around Christmas Holiday

long-term trends. When traffic is slowly increasing or decreasing day by day, the algorithm should adapt accordingly.

These are briefly explained challenges that the new monitoring application is expected to handle. In a design part I will describe how each one of them was tackled.

1.6 Other tools for monitoring

Before I started to design and implement the new monitoring application, I also had a look on already existing tools, which might have similar functionality or at least be a good source of inspiration. Apart from the already mentioned custom made mediation monitoring tool in section 1.4, I asked the stakeholders what other tools they use for monitoring of all their systems, not just mediation.

For years the main tool used for monitoring of distributed systems in provider's network was IBM Tivoli [6]. Tivoli Systems Inc. was found in Texas in 1989 by former employees of IBM and was acquired by IBM in 1996 [7]. Nowadays it is offered as part of IBM's product called Cloud & Smarter Infrastructure.

In 2012 the provider's management decided to replace Tivoli by TrueSight [8] from BMC. The reasons for this replacement are not publicly available, yet personally I had the feeling that Tivoli is rather a system with quite a lot of legacy technologies and technical debt, while TrueSight seems more progressive and modern. Truesight is a complex product offering four main modules: TrueSight Intelligence, TrueSight Pulse, TrueSight Capacity Optimization and TrueSight Operations Management. Out of these four, TrueSight Intelligence is most similar to what mediation monitoring application should achieve. As their website says, "TrueSight Intelligence is a digital analytics platform that scales to discover, organize, and analyze high volumes of volatile IT operations, service, and business data..". Apart from being able to get data from REST APIs and other TrueSight products, it also integrates with high-volume data tools like Apache Spark, Kafka and Cassandra. However, most importantly, the documentation says that TrueSight Intelligence "automatically learns behavior of any metric: machine data, service desk, business, or external (e.g. social media, traffic, and environmental) and visualizes behavior and detects abnormalities" [9]. Unfortunately, the provider does not have this module purchased, so I could not examine it in more detail. Moreover, purchase of this or any other module is a complex process and is out of control of Competence and Delivery Center, so it is not expected to be available in at least the next year or two.

The provider has numerous divisions and has acquired several companies in its life time. Each acquisition is usually followed by a decision, whether different tools with the same purpose in newly acquired company should be replaced or coexist. This results in numerous often obsolete monitoring tools being used and it is difficult to cover them all in the thesis. Nevertheless, I would like to describe one more monitoring system of operating systems, which is built with modern technologies and therefore interesting.

Monitoring of operating systems consists of several independent tools. First tool is *collectd* [10], which is running on each host and in periodical intervals sends various statistics of the OS to Logstash [11]. Logstash is an open source tool for collecting, parsing and transmitting logs and contains plugins for all kinds of inputs and outputs, including collectd. In other words, it might be called mediation system focused on logs. Logstash's task is to collect metrics from collectd and store them in Elasticsearch [11]. Elasticsearch is a very popular and extremely fast search engine, which in this use case stores metrics from all operating systems. Last step in the pipeline is Kibana [11], which is data visualization plugin for Elasticsearch and allows to build UI and custom graphs over the stored metrics data. The metrics of each operating systems are then easily available to support engineers via web browser. The pipeline from Logstash to Kibana is also called as ELK stack and the products are open source and developed by Elastic Inc. [11].

1.7 Functional Requirements

Based on the analysis of problem domain and discussions with the stakeholders, a high-level requirements were agreed on. Most of them were decided in the beginning and are essential for the main goal of the system, but some of them were added later, when it turned out as a useful feature. Following list is an overview of the high-level requirements.

F1 Visualization of flows Users will be able to visualize data flows, including historical data.

F2 Adding and removing flows Users will be able to add or remove data flows. As a nice-to-have feature the application could detect and add new data flows automatically.

F3 Countries support The application will support flows from 4 countries - the Czech Republic, Germany, Austria and the Netherlands.

F4 Monitoring and anomaly detection The application will be monitoring data flows and trigger alarms based on negative anomalies (outages).

F5 Flows settings The application will allow users to change settings of flows to tune the monitoring analysis.

F6 User login The application will support two user roles: root and read-only. Available actions for each role are shown in figure 1.10.

F7 System integration The system will integrate with other systems to receive mediation traffic data and to send detected outages.

F8 Provide API The application will provide API over the received mediation traffic data so other applications can use it and get access to the data.

1.8 Non-functional Requirements

NF1 Performance The application will be reasonably performing to allow trouble-free monitoring of all data flows and also to allow users to get quick insight on the data.

NF2 Monitoring The application must provide a way that allows other tools to monitor it.

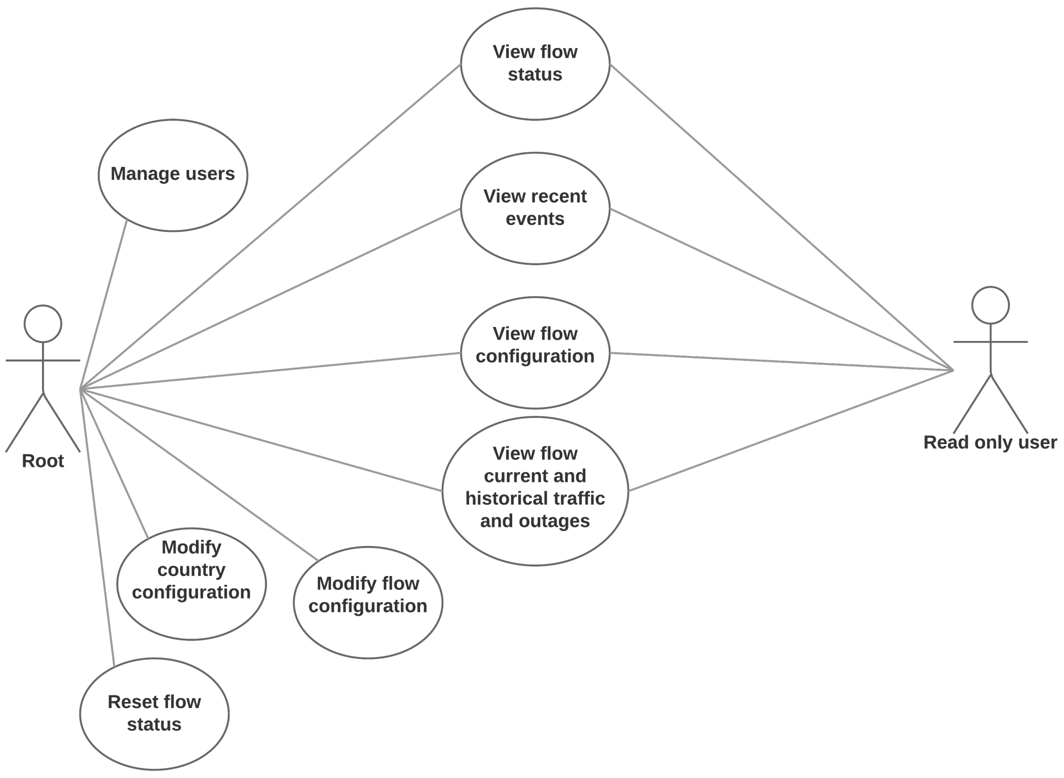

View flow status
Manage users
View recent events
View configuration
Root
View flow current and historical traffic and outages
Read only user
Modify country configuration
Modify flow configuration
Reset flow status

Figure 1.10: Use case diagram

NF3 Design The application's design should be clear and easy to use.

Chapter 2

Big data tools analysis

In this chapter I would like to describe what big data tools are used by the provider and how they are monitored.

In today's market, effective processing and analysis of data gives huge competitive advantage over other competitors. While one of the first companies to do massive data collection and analysis were IT companies like Google or Facebook, it is safe to say that nowadays data mining of all kinds is performed by all key players in any type of industry, and telecommunications are not different. The provider is using big data tools for analysis of their customers to offer them better services and experience. Unfortunately, most if not all companies are keeping detailed information about their data mining efforts in secret and thus they cannot be specified here. Therefore, I will focus strictly on technical part of their big data stack.

2.1 Data processing tools

It can be said that one of the founding stones of modern big data processing is article published by Google employees about MapReduce. "MapReduce is a programing model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/-value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key." [12]. This article was used as an inspiration for development of Hadoop MapReduce module, which works on the same principle and is part of large big data software framework called *Hadoop* [13]. Another significant module of this framework worth mentioning is its distributed file-system *Hadoop Distributed File System* (HDFS).

Hadoop was first released on September 4, 2007 and has experienced tremendous hype soon after. However, in recent years as other big data processing tools appeared, the fame of Hadoop MapReduce has started to decrease. Hadoop MapReduce has several flaws which make it not attractive for modern big data processing requirements. First of all, Hadoop MapReduce is limited to only batch processing, making it unusable in cases where stream processing is required. Next, Hadoop MapReduce has linear one directional data flow and writes results of reducers immediately back to HDFS, which make it slow due to many I/O operations. On the other hand, it does not require much RAM, which might be beneficial in some cases. To close the list of the biggest disadvantages, I would mention limited possibilities of a map-reduce model. It is very difficult if not impossible to create machine learning algorithms or even simple iteration algorithms, such as KMeans, on a map-reduce model.

Because of mentioned disadvantages, Hadoop MapReduce is being replaced by more progressive tool called Apache Spark [14]. It was initially created by UC Berkeley and now is managed and developed by Apache Foundation. It was developed specifically as a response to limitations of MapReduce model. The key stone of Apache Spark are resilient distributed datasets (RDD). Simply put, it is a distributed collection of elements and all operations (transformations or computations) in Spark are executed on this collection. "Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them." [15]. Because of that, the programer can execute several subsequent operations on data and Spark distributes and executes the operations in the most effective way. In Hadoop MapReduce, this is not possible since data would have to be written back to HDFS after every reduce function and therefore decrease the performance heavily. Even though this approach brings more requirements on large RAM capacity, Spark jobs can still be ran on commodity hardware as the number of processing nodes can be easily scaled. Furthermore, increasing RAM is much easier task compared to upgrading CPU or even increasing disk space. Last but not least, since running operations on same data is much easier in Apache Spark, there are already many machine learning libraries and techniques that programers can take advantage of for their needs. Therefore, it is no surprise that the telecommunication provider runs processing jobs only on Apache Spark.

2.2 Data infrastructure tools

In most organizations, processing of data is only one part of their big data stack. Usually there needs to be whole infrastructure built around the processing tools to create the most effective setup. In this section I would like to focus on a system called ZooKeeper [16], which is a true backbone of many big data systems. ZooKeeper might be even less known than tools that are built on top of it, such as Apache Kafka (distributed streaming), Hadoop YARN (resource management) and many other distributed services.

Apache ZooKeeper is a project managed by Apache Foundation and their aim is to "develop and maintain an open-source server which enables highly reliable distributed coordination." [16]. It is a service which should ease development of almost any distributed software and allows developer to implement code for synchronization, communication and coordination of distributed nodes, such as locks, barriers, queues and leader election. One way how to look at ZooKeeper is as at distributed highly available file system. ZooKeeper keeps information in a hierarchical tree, much like a file system does, and each node of the tree is called z-node. Each z-node has a path, such as `/service1/node1` and stored data.

ZooKeeper is always ran as a cluster consisting of several nodes. Each node must have configuration file with IP and port of all other nodes. When all nodes have started, usually the node with the lowest id (first node online) is selected as a ZooKeeper leader. At this point, other external applications (such as Apache Kafka or Hadoop YARN) can connect to some node of the ZooKeeper cluster and take advantage of its functionality. An overview of such architecture is shown in figure 2.1. In order for ZooKeeper to be functional, at least half of its nodes must be online and connected with each other (forming a quorum). ZooKeeper cluster usually consits of odd number of nodes, typically 3 (1 can fail) or 5 (2 can fail). Running with even number of nodes does not bring any benefit: with 4 nodes, still only 1 may fail to not cause failure of the whole cluster. All write requests from external application nodes must go through ZooKeeper leader which then distributes the requests to other ZooKeeper nodes (so called followers). When more than half of ZooKeeper nodes accepts the write request, it is successfully stored and client initiating the request is notified. In case there was less than half of ZooKeeper nodes in the quorum, the write will not succeed. This ensures that the data stored in ZooKeeper are always either consistent, or not stored at all. A read request is similar to write and will only succeed if more than half of ZooKeeper nodes is online. The clients usually have list of all zookeeper nodes and their hostnames, so if the connection to one of them fails, they will try to connect to other node, which might still be in the quorum and work. Apart from basic writing and reading of data, ZooKeeper also allows clients to watch a specific znode for a change or acquire locks on z-nodes, which can be used to implement coordination between distributed nodes.

The whole ZooKeeper software is far more complex and would require several chapters if described in more detail. To readers interested in ZooKeeper and willing to read more I would recommend the official and well maintained documentation, which also contains tutorials and examples. Furthermore, to developers considering usage of ZooKeeper in their own distributed application I strongly recommend looking at Curator library [17]. It provides high-level API of ZooKeeper, making its usage more simple and developer friendly, and also already implemented recipes, such as leader election, queues or barriers ready to be used out of the box.

The provider uses ZooKeeper as backbone for other tools that require it,

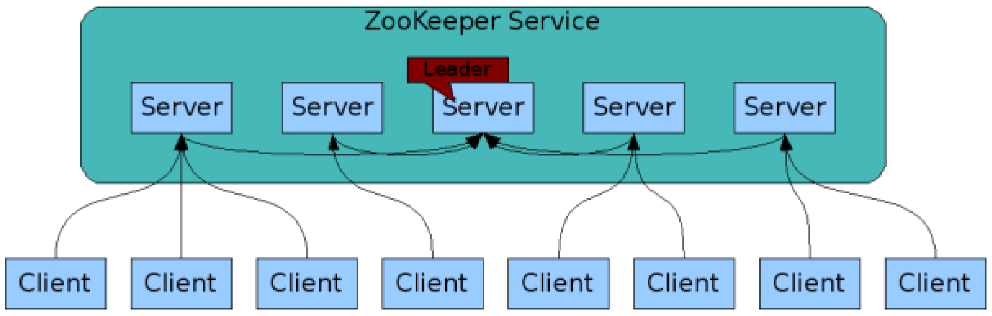


Figure 2.1: ZooKeeper architecture setup

such as Apache Kafka or HBase, which is ”a distributed, scalable, big data store” [18].

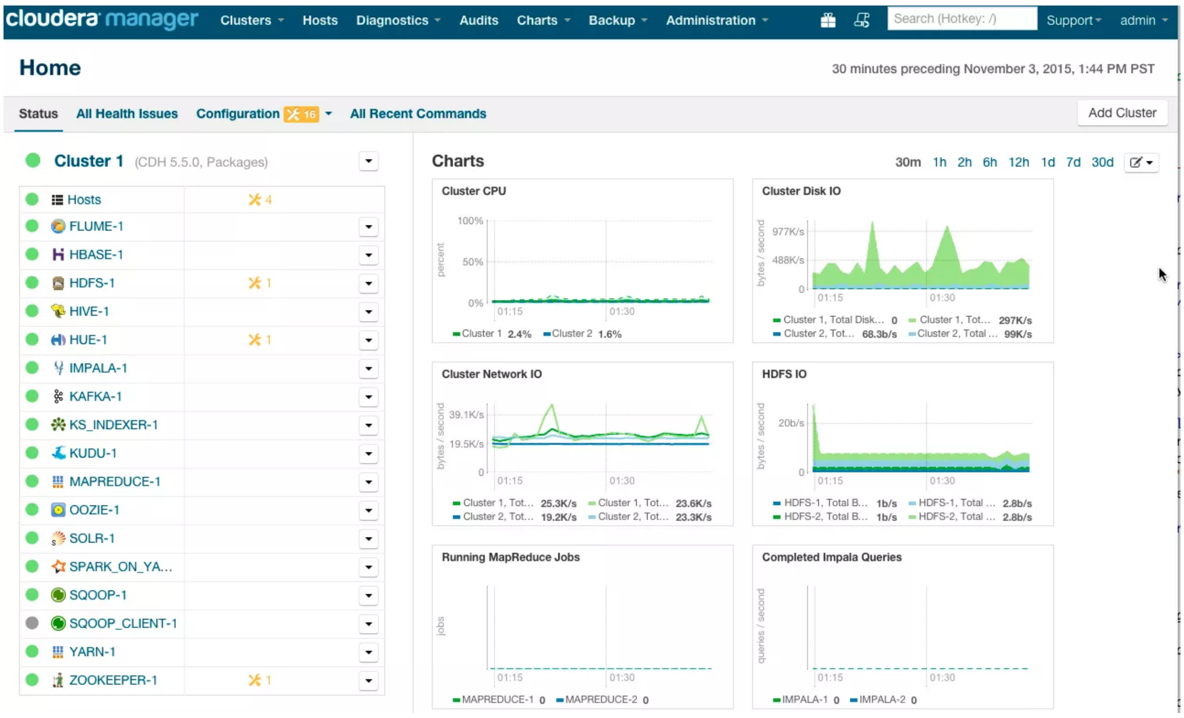
2.3 Monitoring of big data tools

All previously mentioned tools are open-source. On the first sight, that sounds great and certainly is for an experimental, testing or first time usage. However, for a reliable setup in a large company, many other steps need to be done.

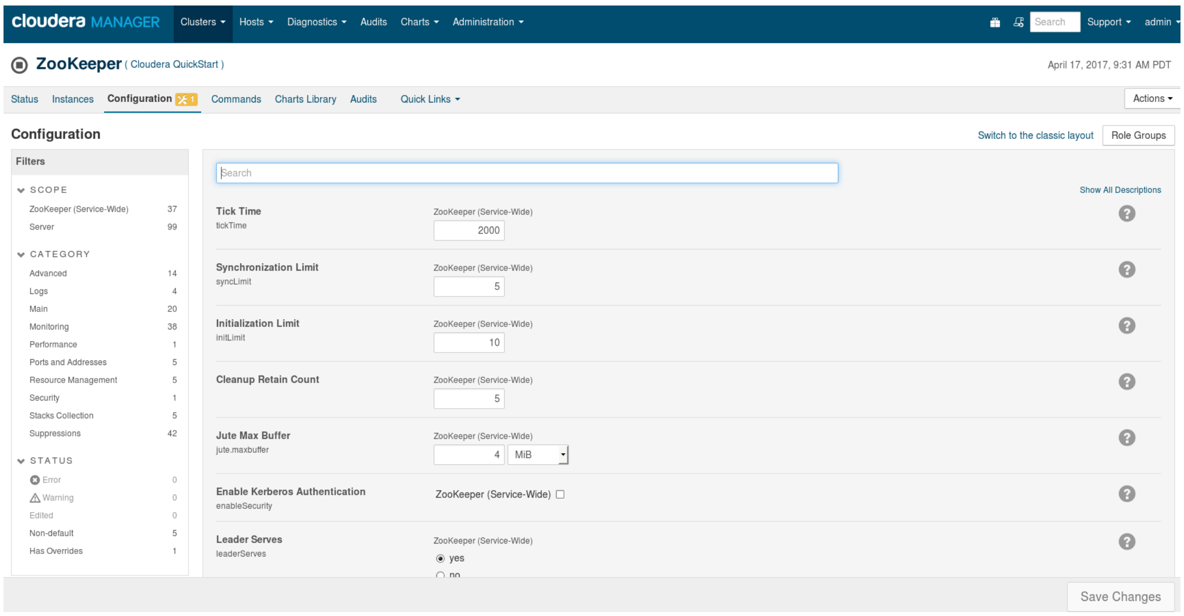
The problems start coming when several of these tools should be combined together, such as streaming data via Kafka to Spark and storing the results to HDFS and HBase. Each of the tools have different release cycles, so issues during upgrade can occur, and does not even have to be backward compatible with itself, much less with other tools used. Furthermore, most of the tools have ways how to get their health status or performance statistics, but none have a way how to systematically and continuously monitor the tool and notify IT department in case of problems that will surely occur.

To solve mentioned problems, there are companies like Cloudera or Hortonworks, which offer big data tools packages. The packages contain most of big data tools one might need and all are tuned and tested to work flawlessly together. Furthermore, it has a fixed release cycle with extra features and bug fixes, which might not be available in open-source releases yet. All this can dramatically speed up setup of company’s big data cluster and ease maintenance. However, most importantly in regard to the topic of this thesis, these vendors’ distributions contain applications for monitoring the tools, nodes and whole cluster. Since the provider is using Cloudera distribution, I examined it further. Interestingly, some parts of Cloudera are for free and even open-source, but more advanced and enterprise-facing features such as rolling upgrades or auditing are only available in Enterprise paid versions. The Cloudera’s monitoring tool is called *Cloudera Manager* and as I found out while running Enterprise trial version, it is far more than just monitoring. It works as UI interface to whole big data Cloudera cluster. It offers special tailored interface for each tool, allowing it to be configured, managed and monitored from one place 2.2.

Due to time and thesis's size constraints, I have decided to implement monitoring of only one tool. I have chosen ZooKeeper because it is core tool and backbone of many big data systems.



(a) Health status overview (from Cloudera Documentation)



ZooKeeper (Cloudera QuickStart)

April 17, 2017, 9:31 AM PDT

Configuration

Filters

SCOPE

ZooKeeper (Service-Wide)	37
Server	99

CATEGORY

Advanced	14
Logs	4
Main	20
Monitoring	38
Performance	1
Ports and Addresses	5
Resource Management	5
Security	1
Stacks Collection	5
Suppressions	42

STATUS

Error	0
Warning	0
Edited	0
Non-default	5
Has Overrides	1

Tick Time tickTime	ZooKeeper (Service-Wide)	2000
Synchronization Limit syncLimit	ZooKeeper (Service-Wide)	5
Initialization Limit initLimit	ZooKeeper (Service-Wide)	10
Cleanup Retain Count	ZooKeeper (Service-Wide)	5
Jute Max Buffer jute.maxbuffer	ZooKeeper (Service-Wide)	4 MiB
Enable Kerberos Authentication enableSecurity	ZooKeeper (Service-Wide)	☐
Leader Serves leaderServes	ZooKeeper (Service-Wide)	[✓ yes] no

(b) ZooKeeper Configuration

Figure 2.2: Cloudera Manager

Chapter 3

Design and Implementation

In this chapter I will focus on selection of technologies, design of monitoring algorithms and the whole application's architecture. Moreover, I will also describe user interface and interesting implementation details.

3.1 Selecting technologies

Selecting the right technologies is one of key factors that shape a software for its whole life. A wrong decision can have devastating consequences. Every lead software developer usually stands in front of this decision multiple times in a software project. Although this case was more challenging than any previous experience I have had before. As I was the one and only developer on this project, I had to select the right technologies for the whole stack, from database and backend technologies to frontend. On one hand it is very nice for a developer, since he can choose technologies that he is most comfortable with or wants to learn more about without being forced to work in a technology due to management or legacy reason. On the other hand it requires him to carry out extensive research of technologies to select the most suitable ones, especially in areas he is not experienced in - which in my case was frontend.

3.1.1 Database

At first, I have decided to select technology for database, then for backend and at last for frontend. It seemed to me that the database layer influences the future architecture the most and can also be the biggest performance bottleneck when chosen or designed poorly. Therefore, I did not want to be restricted by any previously selected technology at this crucial point.

In the past years until around 2008, there was basically only one type of database in use named relational. As the demand for storing and working with big data increased, relation databases turned out to be insufficient and new database types started getting popularity to satisfy new needs. Nowadays, there are many (maybe too many) options to choose from with different pros and cons. Out of all requirements for the new application I choose the following points as most relevant and sensitive to database technology. In the following sections I will introduce several database types and analyze their feasibility with regard to the selected requirements.

To remind readers how mediation monitoring data looks like, it is rows where each input flow row contains name of country, name of LoB, name of source network element, time the data was processed and size of a file containing CDRs. For forward flow it also contains name of the target system.

D1 Aggregate flows with different granularity As mentioned in 1.5.2, the flows can have different granularity, ranging from 15 to 1440 minutes, can change in time and is also expected to decrease in future to even less than one minute. The database should be able to aggregate data or allow backend to do that effectively in reasonable time (NF1), so the flow can be visualized upon user's request (F1) or analyzed by the application (F4).

D2 Add or remove flows The flows can often be added or removed by user or automatically by the application (F2).

D3 Store data for large number of flows As mentioned in 1.2, there are currently over 4500 flows that should be monitored. The database should be able to flawlessly scale as the number of flows increases or decreases.

D4 Store other application data The database should be capable of storing information about users (F6), flows (F5) and countries (F3).

3.1.1.1 Relational databases

Relational databases are probably the most used type of database and so far I have worked with this type of database only. It is based on tables with columns and rows, where each row represents one stored record. Relational databases favor consistency and integrity of data by strictly given schema, transactions and foreign keys. That is beneficial for storage of sensitive crucial data, where consistency and integrity must be maintained at all times. To ensure that data are consistent and integrity is achieved, correct relational database data model should follow Third normal form (3NF) [19], which removes transitive dependency, data redundancy (2NF) and keeps one value per column (1NF). However, while this optimization helps us achieve mentioned goals, it brings problems down the road, when data should be read or queried. Since business records are normalized and stored across several tables, the database engine or programer must first combine this data in order to get desired information (typically by using JOIN query). In case of students and their enrolled subjects, a database query will span across at least three tables: students, students_subjects and subjects. Relational databases, such as MySQL, can handle easily from about a million of rows to ten million of rows if tuned correctly and that is in most cases more than enough. It is the machine generated data, massive number of customers and other entities at big companies where relational databases' performance starts to be insufficient.

To evaluate feasibility of relation databases for my use case, I tried several data models to store data. First and obvious one I started from is to have one table, which would have column for each flow and row as a time entry. One problem with this solution is how to decide the row granularity. The granularity of flows can be smaller than one minute in future and storing entries by five seconds, for example, can be very inefficient in terms of space or query time, since there would be over three million rows created and indexed in 180 days. In case of fixed-lenght table, such as default settings of MyISAM storage engine in MySQL, a row occupies same size even if it contains only NULL values [20]. To prevent this, *ROW_FORMAT=DYNAMIC* can be used when creating the table and NULL values will no longer take space, but "a row can become fragmented (stored in noncontiguous pieces) when it is made longer as a result of an update."[20].

More serious problem is the row size limit though, which is 4096 columns per MySQL table [20] and 1000 columns per Oracle 11g table [21]. To overcome this limitation, data flows could be split into smaller tables, grouped by LoB for example, or even each flow put in its own table. Nevertheless, there would still be a problem with what granularity of the time entries should be used. If each flow would have its own table, the table's row granularity could be adjusted independently, but would have to be decided at the time the data is stored, not later.

The last data model would already satisfy my needs as specified in the database requirements, but would still perform inefficiently in a case that data of n flows should be retrieved at one time. In such a situation, not one, but n queries must be executed on the database, because each flow is in different table now. That means n times finding necessary keys of desired time range. If the flows were all in one table, only one query with a *SUM* function and an optimal GROUP clause is necessary. Moreover, new flow or LoB can appear at any time and thus each insertion of flow data would have to verify that its column or table exists and create it if it did not.

Last but not least, relational databases are difficult to scale horizontally. Usually, the performance can be improved by enhancing the machine (scaling vertically) rather than adding other nodes. However, I and the skateholders do not expect to need horizontal scaling, so this is not crucial for my application.

Overall, I think that relational databases would work, but would certainly bring some limitations, such as worse performance when querying multiple flows at one time, checking existance of table or column when inserting flow data or the need to decide granularity at the time data is stored, not when it is queried.

3.1.1.2 Document databases

Document databases are one type of so called NoSQL databases, which gained on popularity especially in the last five years with development of cloud technologies and big data tools. Document databases do not have a fixed schema and instead of tables use collections of documents. There can be unlimited numbers of documents in one collection and they do not have to have same structure. The only restriction is that each document must have some kind of id attribute, which is used as a unique identifier and index key (analogy of primary key from relational databases). It is up to an application if other attributes of document should follow some structure and how strictly it should be enforced. In general it can be said that document databases are far more relaxed and less strict that relational databases. They do not offer transactions (they must be implemented on application level) or foreign keys and thus do not guarantee consistency. They usually offer eventual consistency between nodes, which means that changes are not propagated to other nodes immediately, but after some time. Furthermore, document database do not have any mechanism of combining data from two collections, such as *JOIN* in relational databases, which means that it also must be implemented on application level. Therefore, it is recommended to denormalize data so the need for joining two collections or documents is minimized.

For instance, to model students and their subjects (for which join across three tables in relational database is needed), I can create only one collection *students*, where each student will have attribute *enrolledSubjects*, which will be a list of subjects. Each subject in this list can then be either just id, or small to medium size object with other information such as subject's description or teacher's name. There is no clear rule how much the data should be denormalized. The more data are denormalized, the more effective the application will be for read queries, but also there will be more redundancy, higher possibility of data inconsistency and worse performance in case of data update. It is task for developers to find the right amount of denormalization for their expected queries and use cases.

3.1.1.2.1 Comparison of different document databases There are many document databases available. In this paragraph I would like to focus on the most significant ones: MarkLogic, MongoDB [22] and Couchbase. They are all quite similar in terms of usability and it is very difficult to choose one over another. According to db-engines.com, all of them place in the top five in document stores category, together with AmazonDB, which can be ran only as part of Amazon's AWS platform, and CouchDB (predecessor of Couchbase). When picking between these three databases it comes down to personal opinion, preference and colleagues' recommendations. Even though I

Name	Popularity Score
MongoDB	325
Couchbase	30
MarkLogic	11

Table 3.1: Popularity Score according to db-engines.com

had no prior experience with document databases, I have already heard quite a lot about MongoDB, which is clearly the most popular database available 3.1. What I appreciated on MongoDB and Couchbase is that they are open-source and for free and while MarkLogic offers decent free licenses, it needs to be manually renewed every 6 months. One thing that I especially liked about Couchbase was their query language N1QL, which was introduced in 2015 [23]. Their aim is to make it as much similar to SQL as possible and ease it for people used to relation databases - they even offer alternative to JOIN query. Querying in MongoDB is done via composing JSON objects and submitting them to the database. It is slightly difficult to get used to at first, but becomes very powerful later on - especially when combined with MongoDB aggregation framework 3.1.1.2.3.

After some considerations I have decided to perform further analysis on MongoDB mainly due to its popularity and stick to it if it proves to work well. It is very well proven, widely used and has large community online to ask for help. On the other hand, Couchbase looks like a strong competitor with its N1QL language and I am very curious if it is going to take more market share in near future. Quite a lot of posts can be found from people considering move from MongoDB to Couchbase. I ruled out MarkLogic as open-source alternatives turned out very well and I am not a fan of closed software.

3.1.1.2.2 Data model For storing monitoring data of mediation flows I have decided to create one collection called *flows_data*. Each document in this collection will represent one minute, which means, that in one year there will be maximum of 525600 documents in this collection. Each document will have *id* attribute, which will be date object, and *data* attribute, which will be object storing data of all flows for that minute. The data object will have hierarchical structure, with countries at top level, then LoB name, flow category (input or forward), flow name and second. Structure of the document is shown in figure 3.1. The fact that the documents will be divided by minute will not cause excessive number of records (and thus will not degrade the performance) and because each flow will have up to 60 attributes corresponding to each second of that particular minute, this structure allows us to store data with only 1 second granularity. This document structure is therefore very efficient and flexible and because monitoring data are not complex and are without any relations, disadvantages from previous paragraph about denormalization and combining entities are not applicable here.

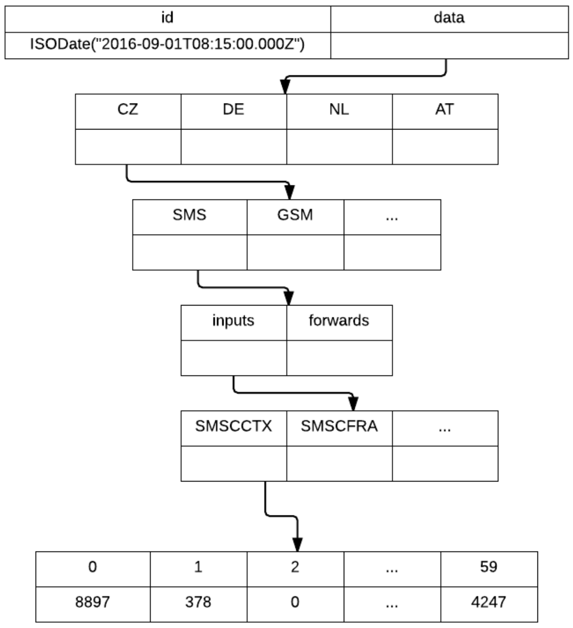


Figure 3.1: One minute document structure

Adding and removing flows (D2) is achieved naturally, I don't have to add any columns or modify existing documents - as data for new flow comes, they are just stored to respective position in a hierarchy of the minute document. Storing of many flows (D3) could become problematic in case all flows would have record for each second. In such a case, for 4500 flows there would be 270000 numbers (4500 * 60 seconds) to store in one minute document. MongoDB has a limit of 16MB per document (Couchbase 20MB) and this theoretical document would still not exceed 2.2MB when stored. Nevertheless, it is possible that performance would somewhat decrease. On the other hand though, having data for each second of all flows is not going to happen in foreseeable future and this theoretical limitation is acceptable by the stakeholders.

Storing other application data (D4) is not going to be problematic since the data model will not be complex and could be denormalized if needed. In the worst case, relational database could be used to store entities for which document database won't be acceptable, although using just one database will be preferred for the sake of simplicity.

3.1.1.2.3 Data aggregation Data aggregation (D1) is probably the most necessary and at the same time complex requirement. It comes in play when data needs to be aggregated by granularity for monitoring analysis (more in

1.5.2) or when user wants to see data for large time range, such as five days. In that situation data needs to be aggregated into reasonable chunks to be still meaningful (e.g. fifteen minutes intervals) but not too large for transmission.

Fortunately, MongoDB has *aggregation framework* which is used to achieve same results as with SQL's GROUP BY. In MongoDB, client composes array of JSON objects which specify pipeline of one aggregation query. Each object in the array represents a step (or stage in MongoDB's terminology) of the pipeline and transforms data in a certain way. To achieve desired results, following stages had to used.

- *$match* Filters relevant documents only.
- *$group* Groups documents by specified expression or field. Requires accumulator operator to aggregate the actual results.
- *$sort* Sorts documents.

In the first *$match* stage I only select relevant documents, let's say from 1.1.2017 to 5.1.2017. Then in *$group* stage I have to transform attribute of date type into individual numerical attributes year, month, day, hour and minute. Based on how the conversion from date to numerical attributes is done I control the granularity of resulting data. In *$group* stage I also have to specify accumulator operator and attribute it will be executed on - I must use *$sum* over *data.CZ.GSM.input.MSSBRN1A.sum* attribute of the documents. After this stage, the documents are grouped and contain summed value of *data.CZ.GSM.input.MSSBRN1A.sum*. To make subsequent sorting easier, I also keep any date object as representative of each aggregated group. Final sorting stage is straight forward and can be followed by *$projection* stage, which can be used to remove no longer needed attributes. Example of group stage in aggregation query is shown on 3.1.

```
db.collection("flows_data").aggregate(
[
  {"$match": {...}},
  {
    "$group": {
      "_id": {
        "year": {"$year": "$_id"},
        "month": {"$month": "$_id"},
        "dayOfMonth": {"$dayOfMonth": "$_id"},
        "hour": {"$hour": "$_id"},
        "interval": {
          "$subtract": [
            {"$minute": "$_id"},
            {
              "$mod": [{"$minute": "$_id"}, 15]
            }
```

```
                ]
            }
        },
        "anyDate": {"$first": "$_id"},
        "MSSBRN1A": {"$sum": "$data.CZ.GSM.input.MSSBRN1A.
            sum"}
    }
  },
  {"$sort": {...}},
  {"$project": {...}}
]
)
```

Listing 3.1: MongoDB aggregation query

3.1.1.3 Other database types

I have done quick analysis of other NoSQL database types like key-value stores, graph based and column based. First two were ruled out immediately since they clearly do not fit this use case. What seemed to be more promising were column databases like HBase or Cassandra [24]. The have some constraints when it comes to querying data but on the other hand are capable to store huge amounts of data and scale horizontally very well. In Cassandra I would use similar data model as in MongoDB: minute as a row index and flow name with second as a column name. It would be a bit challenging to store hierarchical structure LoB, input type and flow name, but still doable. Cassandra database has been evolving and changing ever since it was created and even though it is stated that ".. the data tables are sparse, so you can just start adding data to it, using the columns that you want; there's no need to define your columns ahead of time." in [25], the later version of Cassandra requires the column schema to be defined up front, which can make inserting new flows' data more difficult as in relation databases. Nevertheless, I believe that column database like Cassandra would work well for this use case, but it is not as friendly and easy to use as MongoDB and a bit overkill. In case the data were multiple times larger (for example if I dealt with actual CDRs' content and not only their sizes), I believe Cassandra would be well worth the time and effort.

3.1.1.4 Summary

Based on the previous analysis, document databases (and MongoDB) are the most suitable for specified use case. Flows can be added and removed easily and only one aggregation query is necessary to retrieve data of several flows at one time. Therefore, the application will use MongoDB for persisting data.

3.1.2 Backend

Selecting backend technology and programing language can turn into never ending discussion, where each debater can have different opinion and still be right. From the very beginning of this project I was seriously considering three languages for the backend layer and in this section I will describe their pros and cons and which on was chosen.

3.1.2.1 Java

First of all I was considering Java. It is still the most popular language, but currently experiencing decline in popularity according to TIOBE Index 3.2. Nevertheless, in enterprise applications field Java is expected to keep large market share. There is large number of frameworks and libraries available for Java and together with my vast experience with it from professional and university projects I knew that Java would not be a wrong choice. Furthermore, it is also very strong in the field of big data and at the start of this project there was possibility, that Spark or Hadoop will have to be used for analyzing the data flows. Ironically, at the same time my past experience was discouraging me from using Java as I had already known that technology quite well and I would learn nothing new. Last but not least, development in Java can be slower compared to dynamic programing languages.

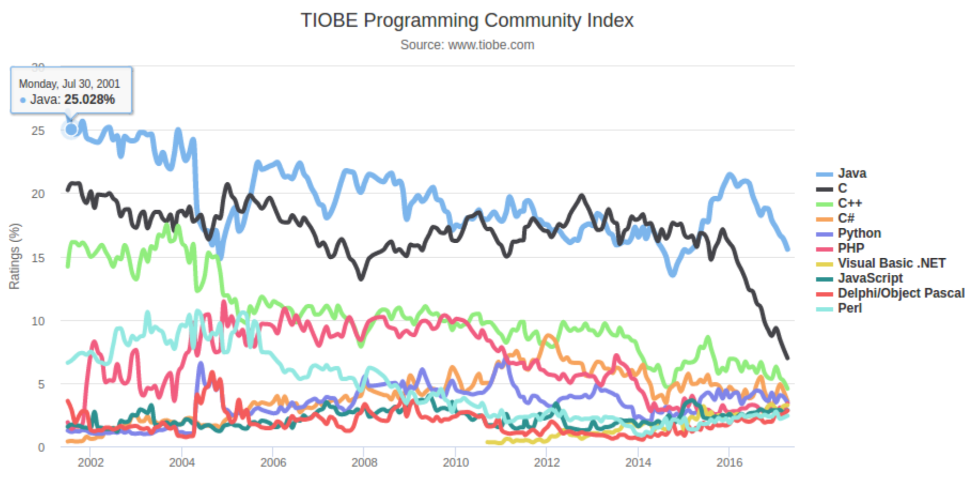


Figure 3.2: Programing language popularity

3.1.2.2 JavaScript

Next I was considering JavaScript. It has monopoly over frontend and with release of Node.js it started quickly getting traction in backend, too. One aspect of Node.js (and JavaScript in fact) is its asynchronicity. JavaScript is ran only in one thread, which means that the programing code is not always executed sequentially, but asynchronously. Simply said, the programing code just specifies small functions (also called tasks or messages), that should be ran and the thread (also called event loop) executes them whenever possible. This brings big advantage when the programs's task is to mainly do non-blocking I/O operations, such as sending data from database to the application or from the application to a user. In that case, JavaScript is not doing the main work, it only sends query to database and moves onto some other tasks. Once the data from database are ready, JavaScript receives callback, passes data to network layer and again moves onto other tasks. Because of that, it is possible to serve many requests with just one thread, compared to typical worker-thread pool architecture, where each request is assigned to one thread. On the other hand, if there should be some heavier CPU processing done at each request, the whole server suddenly becomes far less performing. One of other advantages is, that a developer does not have to change language when switching between frontend and backend and with combination of MongoDB, not even when changing to database layer, since MongoDB uses JSON for queries.

3.1.2.3 Python

Python is general purpose programing language, which maintained stable user base for past two decades. It is efficient, easy to write and is used especially for complex scripts, backend applications, all sorts of science programs and also data mining. It has a huge ecosystem built around with many libraries for data mining, machine learning and numerical computation. When I was evaluating Python, its versatility intrigued me the most, since I could use one language for quick researching and analyzing the data and as well use it to implement the backend layer. I dived deeper into Python's frameworks and found simple lightweight HTTP server called Flask [26]. I was also considering Django, which is famous large framework for web applications, but it seemed like a bit of overkill for this project.

3.1.2.4 Selected backend technology

After all I have decided to go with Python, since it can be used for everything I might need to implement in this project. JavaScript was ruled out because it is not suitable for heavy CPU processing and does not have many scientific libraries. I did not want to use Java since I had already known that technology quite well and I wanted to try and learn something new.

3.1.3 Frontend

JavaScript and its ecosystem have dramatically changed in the past five years. The JavaScript world is constantly evolving with new frameworks appearing and disappearing basically on monthly basis. For years, it was a standard and sufficient to use jQuery library for all frontend development, but as the websites started to be more like applications than just static pages, the need for better frameworks than jQuery emerged. One of the first frameworks were Backbone and AngularJS (both released in October 2010), which were quite popular and widely used until around 2014. At that time, a Facebook's framework called React [27] started getting large traction and made AngularJS, Backbone and all others look obsolete. Google, developer of AngularJS, learned from mistakes and disadvantages in AngularJS and in September 2016 released brand new rewrite of AngularJS under the name Angular (also called Angular2) [28]. Nowadays, React and Angular2 are the biggest competitors, known also by others outside of frontend developers community. Vue.js is becoming popular very recently, but is still not as known as the first two.

As I had no experience with either React or Angular2, I had to rely on other resources when comparing these two. The opinions on React and Angular2 vary and it is impossible to find a clear answer to which is a better framework. From technical point of view I especially liked how React is composed of small components and has a one direction binding with virtual DOM (Document Object Model). On the other hand, Angular2 utilizes TypeScript (JavaScript with static typing and class-based OOP) and provides all functionality out of the box, whereas React is more like a library. Each framework has its advantages and disadvantages and for unbiased person it is difficult to choose one over another. After all, I have decided to use React mainly because of its popularity. React has been around for over four years and has a proven track record of production usage, larger community, many learning resources and other support libraries. Angular2, which was released in September 2016 and at the time of this decision was just few months old, did not have any of that. Therefore, the following section will focus on React only and will briefly describe its fundamentals.

3.1.3.1 React

As previously stated, React is developed by Facebook and was initially released in 2013. It is something between a small unopinionated framework and a large library, even though its official description says: "A declarative, efficient, and flexible JavaScript library for building user interfaces." [27]. In my opinion, since there are many libraries specificaly made for and strongly connected to React, all together it can be considered as one framework.

3.1.3.1.1 Virtual DOM React popularized several interesting principles, which were not common in frontend development before; one of them being Virtual DOM. Every website is made of HTML DOM (Document Object Model) and uses nested HTML elements to specify how it looks. Browsers provide JavaScript API functions like `getElemebtById` or `setAttribute`, which is used by developers directly or by other JavaScript frameworks to modify the HTML DOM and thus appearance of a website. If the API is invoked only with necessary changes, it performs well, however, it puts constraints on framework and developer to always invoke only necessary calls to achieve desired changes. React and its Virtual DOM made it possible, that developer does not have to bother about what changed on each HTML element, but only set the right state on a component (React's class, which is then rendered as one or more HTML elements). The Virtual DOM is a parallel DOM in JavaScript memory and whenever some component's state changes, React invokes `render` method on that component, which returns simple JavaScript objects (Called React elements) defining how the component itself and all nested components should look like. This is then compared to Virtual DOM and any difference between element in Virtual DOM and corresponding element in just returned React elements is reflected by invocation of API methods to modify and synchronize actual DOM with Virtual DOM. This is one of the core concepts of React and is described in more detail at official documentation [29] or by Facebook's Software Engineer at [30].

3.1.3.1.2 Components For someone coming from MVC framework world, starting with React can be confusing. In terms of MVC, React itself is only the 'V'; it is just view library. The key building stone in React are components, which should represent small set of HTML elements needed to create one meaningful but small part of the whole page, such as in figure 3.3. React components can and should be nested and if designed correctly, easily reusable even across different projects. React component has two important attributes, props and state. Props holds data passed by parent component and state keeps current state of the component. Component has no more information about the application's state than what is passed to it in props. Whenever state changes, `render` method is invoked. In `render` method developer writes code which specifies how this component should look like based on state, desired styles, values to display, other nested components and so on. The render method typically uses JSX - a language syntactically similar to HTML. Returned React element is then compared to Virtual DOM as described in previous paragraph, and changes are applied on real DOM via browser's API.

3.1.3.1.3 Redux React component can easily communicate with its parent or children via props and communication between sibling components can be done via their parent. However, this makes it difficult when components are in different sub trees of the hierarchy with the first common ancestor way up in the hierarchy, at worst case at root. To illustrate the problem let's consider root component of an application called `App` which contains information in its state whether user is logged in or not. Then, at deeper level of some sub

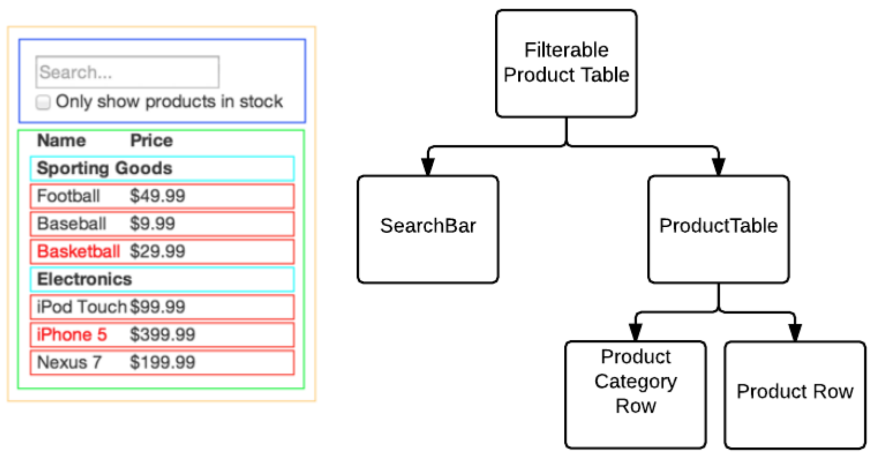


Figure 3.3: React components hierarchy - from risingstack.com

tree, there is a component `UserSettingsButton`, which should be shown only if the user is logged in. In order for `UserSettingsButton` to know whether the user is logged in, each component between `App` and `UserSettingsButton` would have to pass this information to its child, which is obviously bothersome 3.4. There are several ways how to solve this problem such as using global variables or passing large state objects (contexts) to each child component. However, the most elegant and recommended way is to use Redux.

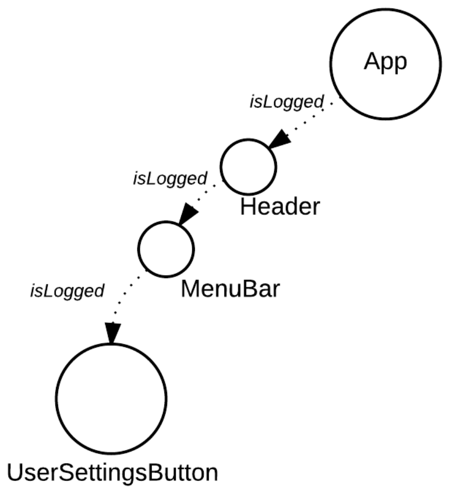


Figure 3.4: React components passing state via props

Simply put, Redux is a JavaScript library for managing application state. It holds state of an application in one place called store and provides ways how each component can modify and read the state. Reading the state is straightforward: component subscribes for specific part of the state it needs and whenever this part of the state changes, component is notified and rerendered. Modifying the state is a bit more tricky. It is not done by components directly, but by emitting Actions, which are then processed by functions called Reducers. Components dispatch action with some optional payload information, which are received by reducer function. This function then based on current state and the action returns new modified state. For example, if `UserLogoutButton` emits *LOGOUT* action, reducer will modify the state by setting `isLoggedIn` to `false`, which will immediately trigger rerender of components subscribed to this part of state, such as `UserSettingsButton`. Same principle can be used for any state modification including HTTP requests, which typically utilizes action *TRIGGER_REQUEST* with request information such as url and HTTP method in payload. One of the advantages of this approach is also the way how an application can be debugged. Since every change of state happens as a reaction to actions, it is easy to log and later simulate any change of the application's state; in Redux terminology this method is called time traveling.

There are other interesting concepts used by Redux, such as immutability of state, however, I believe that the necessary fundamentals have been described. While using Redux I have also realized, that it requires a developer to make many steps to implement even basic functionality; subscribe component, define action, dispatch action and implement reducer behavior. Therefore, I used it only for state information which is more of global character and shared between several components in different subtrees of the application hierarchy, such as already mentioned logged in user information.

3.2 Outage detection

Detecting outage is the absolute core part of the application, so I started developing it first. In analysis section 1.5 I have described that flows in mediation system are not streams but data sent in batches in more or less periodic intervals. It was also decided, that the best way to smooth this data is by increasing the granularity. In this section I will describe development of the outage detection algorithm, problems that appeared and how they were tackled.

3.2.1 Algorithm's fundamental concept

In cases I just want to monitor if a system is running or not, detecting outage can be implemented quite easily by some periodical checks. After all, similar approach have been used in old monitoring tool until now. However, this method is not reliable enough, because the system may look functional even though it does not work as expected, such as when it is processing only small fraction of data than it usually does.

To make the detection smarter, I have decided to create an algorithm which will in its basis firstly calculate expected traffic level, then compare it to the actual traffic level and potentially, if other conditions are met, trigger alarm.

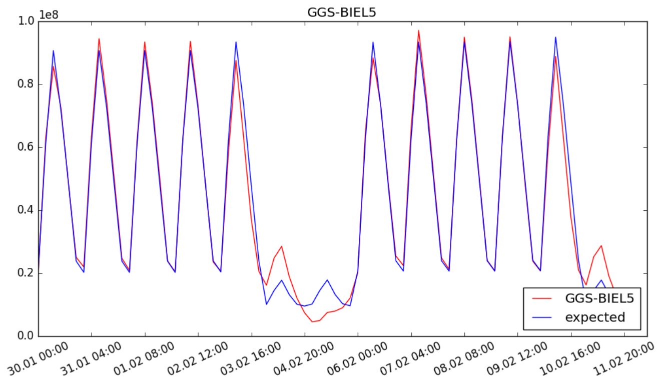
3.2.2 Calculating expected traffic

At the first sight it is noticeable that most flows have a traffic cycle of one day. The prediction algorithm will therefore calculate expected traffic level from traffic levels of the same time but in previous days. I tried several functions for computing the prediction: median, mean, weighted mean and trimmed mean. The results were mostly similar to already mentioned comparison of those function in analysis chapter 1.5. Mean is not suitable because exceptional days such as New Year's day can increase the expected value for many following days. Weighted mean with larger weight for most recent days brings advantage of faster following trends, but still has the same problem as mean (for shorter period though). It turned out that the most suitable function to choose here is median; it is resilient to outliers and no unusual traffic can significantly distort the predictions. This also brings the benefit that the prediction computation does not have to ignore days which experienced outage, which is considered important since data from the provider do not contain information about historical outages.

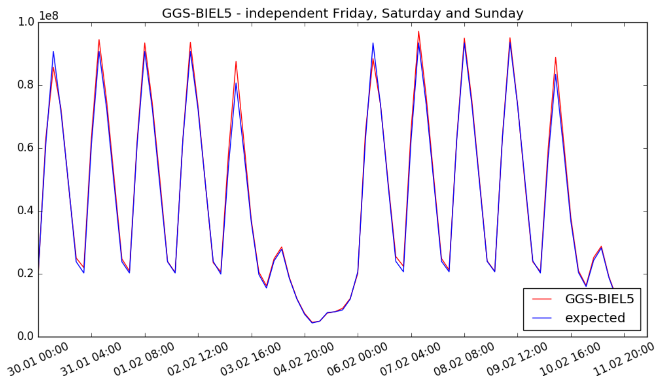
3.2.3 Periodical traffic changes and trends

In 1.5.3 I analyzed and found out unexpected traffic changes and trends that can occur. It was discovered, that the traffic mostly depends on whether it is weekend, national holiday or business day. First step was to tune the algorithm for business days and weekends as it is the most common change. I started with the simplest solution which is to divide days on work days (Monday to Friday) and weekend days (Saturday and Sunday). Calculating expected traffic would then only use days of the same type: either work days or weekend days. The results were surprisingly successful and for most flows sufficient.

However, few flows cannot be divided into work and weekend days. For instance, German flow GGS - BIEL5 has traffic on Friday about 20% smaller than during other work days and traffic on Sunday is 95% lower compared to Saturday. Without any changes, the prediction for Sunday would always be higher than it is usually, since Saturday would be used for prediction too 3.5a. I came up with two possible solutions to this problem. First one was to allow mediation engineers to set adjustment factor of the expected value for each day. For BVSRZT it could be 80% of the expected value for Friday and 15% of the expected value for Sunday. However, this solution would require engineers to change the adjustment factor whenever the traffic on Friday or Sunday changes compared to other days. The other option, which was later chosen as



(a) Regular prediction



(b) Prediction with independent Friday, Saturday and Sunday

Figure 3.5: Independent days prediction

preferred, was to allow mediation engineers to set independent days. Those days would then only be compared to past but same days of a week. This solution is superior to the first one because it does not require any modification in cases that traffic changes. It can be considered as a small disadvantage that it will not follow very recent trend changes, but while examining the flows, I have not found any single one where it would cause troubles. Illustration of regular prediction and prediction with independent Friday, Saturday and Sunday can be seen in figure 3.5. Another flow where independent days were applied is BVS - BVSRZT, where Fridays are 10% less busy and Mondays 10% more busy than other work days.

3.2.4 Sudden traffic changes

Using algorithm with floating window works well for long lasting and slowly changing trends, where the prediction can easily and automatically adjust because of floating median window. Quick but weekly periodical trends can be also overcome by mechanisms described in previous section. What is left to tackle and turned out to be the most difficult are sudden quick changes in traffic. Sudden changes occur during nationwide events such as Christmas, New Year's Eve or national holiday and were already partially described in analysis chapter 1.5.3.

3.2.4.1 National holidays

At first I looked into national holidays and tried to find some pattern about how the traffic behaves. From analysis phase I already knew, that national holiday also influences traffic of surrounding days, which makes it even more difficult and complex. Furthermore, some national holidays have bigger influence on traffic than others. While developing the algorithm I had data available only for three typical national holidays: September 28, October 28 and November 17, and I realized that there will never be enough of recent national holidays data to base predictions strictly on past national holidays. Fortunately, it was discovered that weekends have similar traffic pattern as national holidays and can be used for calculating the expected traffic level. As seen in figure 3.6, on 28th of September the traffic was greater than expected and on 17th of November the traffic was smaller than expected by 17%, even though weekend predictions are typically withing 5% percent deviation for this flow. Sadly, I could not find any reason why some national holidays are more busy than others and since only one year data are available, making any conclusions is not possible here. Nevertheless, approximating national holidays to weekends is the best I could do and 17% difference is still within acceptable range.

Christmas Day and New Year's Eve have the opposite effect on some flows, such as SMS LoB where the traffic skyrockets at peak hours 1.9. Since this is a positive change, there is no risk that false alarm could be triggered and fine-tuning the algorithm to increase the expected traffic level for few hours of a year is not worth the time and extra complexity, which would have to be introduced.

3.2.4.2 Days around national holidays

The traffic on days around national holidays is even more tricky to estimate than the national holidays itself. It was already shown in analysis chapter, that national holiday can influence and significantly decrease traffic on surrounding days as it happened a day after 17th of November 2016. First idea I had was to try to come up with a way how to find flows with correlating traffic and use data of other correlated flows when deciding about an outage. Obviously, this approach would be tricky in the sense that if all correlated flows suffered a real outage, the algorithm might not detect it. Another problem with this solution

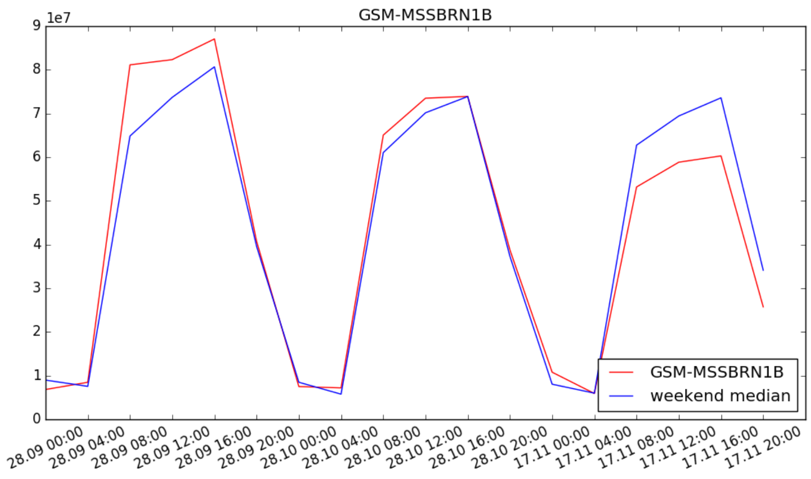


Figure 3.6: National holidays with weekend median

would be an introduction of certain complexity that would turn the monitoring algorithm into a black box.

After consultation with the stakeholders I decided to use much simpler and understandable approach. Work days anticipated to be less busy than usual will be called *lazy days* and each flow will have a configuration attribute specifying how much the expected traffic should be decreased on lazy days called `lazyDayDecrease` (default value was set to 0.7, meaning a traffic on lazy days is expected to be 70% of its usual level). The last step is to find the lazy days. For now, I have decided to let administrators manually specify lazy days in global settings for each country similarly to national holidays, but a basic algorithm could be implemented in the future to find them automatically.

3.2.5 Severity of outage

With described algorithm and techniques, the application is able to set expected level of traffic for each tick by using floating mean and then compare it to current traffic level and compute the difference; I called it *tick difference*. While evaluating the results it was found that traffic over night is more unstable and prone to changes than during a day. Night traffic is dramatically smaller and around 3am is usually not even 1% of peak time traffic. Therefore, naturally, any change in traffic during a night can have large impact on tick difference while the severity of such a change is for most flows absolutely negligible when put in context of daily average traffic levels. To understand why this is a problem, it is important to note that each alarm causes support engineers to take an action and that costs money, thus in some cases it might not be worth to trigger alarm.

It was suggested by stakeholders to create peak and off-peak hours, which

$$tickDifference = \frac{actual}{expected}$$

$$dayDifference = \frac{actual - expected}{dayAverage}$$

$$trafficDifference = max(dayDifference, ticDifference)$$

Figure 3.7: Calculating traffic difference

would each have different levels of alarm threshold, such as 0.8 for peak and 0.4 for off-peak hours. However, there are several disadvantages of this solution. Firstly, it requires specific configuration for each flow of peak hours and thresholds and they might even change over time. Furthermore, the change in traffic is typically gradual and setting strictly peak and off-peak hours does not reflect that.

I proposed different solution which was later accepted by the stakeholders. In addition to tick difference, a *day difference* is computed. The difference between expected and actual traffic is divided by day average so that it expresses a significance of the decrease in context of average traffic level. This approach suppresses alarms for traffic drops which are not significant and does not require any special settings like rush hours. However, in cases that day difference would be greater than tick difference (typically in peak hours), the tick difference is used to prevent exaggeration of traffic drops. The whole formula is shown in figure 3.7 and comparison of day difference to tick difference in figure 3.8. It can be seen that around 3am the traffic is almost same as was expected and tick difference still shows decrease of 30%. The day difference performs more reasonably and shows drop of about 2%. On the other hand, to not completely ignore very significant outages during less busy hours, the dayDifference is not applied if traffic is less than 10% of expected tick level.

Last but not least, some flows such as paid SMSs should be monitored strictly even during off-peak hours, because not forwarding paid text messages can cause severe losses even then. The application will therefore allow monitoring engineers to choose between using day difference and tick difference.

3.2.6 Detecting outage

Previous sections have shown how to calculate expected traffic value, deal with predictable and less predictable changes and compute traffic difference. In this section I will describe rules which are applied when deciding whether outage alarm should be triggered or not.

I divided the detecting outage process into two phases: detecting outlier

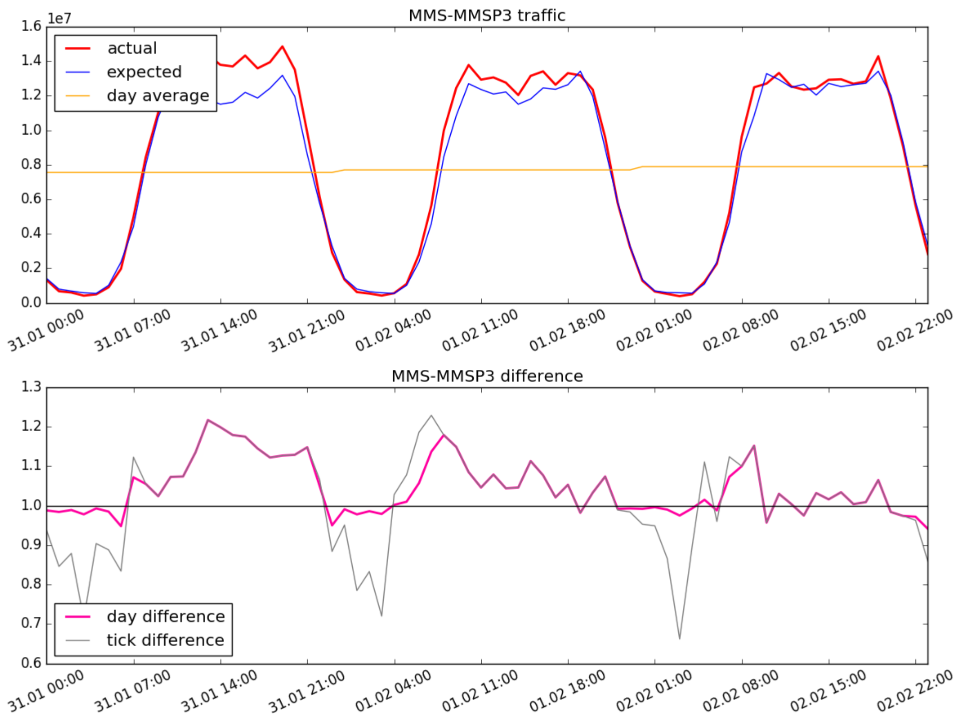


Figure 3.8: Tick and day difference

and detecting outage. Detecting outlier is executed every time analysis is run. The goal of this phase is to quickly find whether a flow is behaving correctly or not and to achieve that, traffic difference value is computed by using mentioned techniques. Then it is compared to *softAlarmLevel*, which can be different for each flow. Typically, softAlarmLevel is around 0.8 and means that traffic difference above 0.8 is considered as inlier and healthy. If traffic difference is below softAlarmLevel, the flow is handed over to outage detector for further evaluation.

At first, outage detector compares the traffic difference to *hardAlarmLevel*, which is usually around 0.5. If traffic difference is lower than hardAlarmLevel, flow is marked immediately as failing. If it is between soft and hard alarm levels, traffic difference is calculated also for one previous tick. If chronologically first traffic difference is above soft alarm, flow is marked with warning label only, but if both time differences are below soft alarm level, flow is announced as failing. Implementing this soft and hard levels logic was necessary, because some flows are not stable enough and sometimes time difference of stand-alone ticks can be as low as 0.7 and still not signify any traffic problems. However, two consecutive ticks with time difference below 0.8 are usually already a sign of some problems.

This two phase design also allows to implement not so accurate but quick outlier filter in first phase and in second phase execute CPU-intensive but not so performant algorithm only on flows which are more likely to be failing. After all, the logic implemented in second phase turned out to be not as complicated as was initially expected, however, this two phase design might prove to be beneficial in the future when second phase is extended.

To illustrate outage detection, I picked input flow SMSCCTX of SMS LoB. On 20th of January 2017 the provider started slowly moving traffic from the flow to SMSCFRA and thus it can be very well seen, how soon the algorithm detects this. I did not manage to find a way how to indicate outages in Matplotlib (Python graph library) nicely, therefore, I am using screenshot taken from the application itself 3.9. The gray areas bordered with red lines depict detected outages. Soft alarm (yellow horizontal line) is set to 0.8 and hard alarm (red horizontal line) to 0.5. Needless to say, that SMSCCTX is quite stable flow and much more strict alarm levels could be used.

3.3 Architecture design

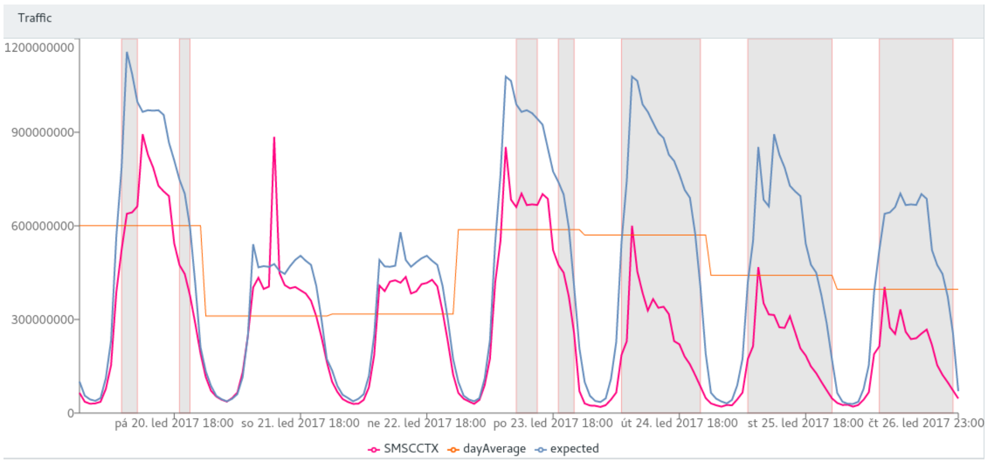
The application's architecture is divided into four stand-alone parts. First one is MongoDB database, which is persisting all data. Next, there is API service which provides API over the database for frontend application and allows it to access and modify configurations, flows or get flow traffic data. Then there is monitoring daemon, which is a service running in background and monitoring each flow in periodic intervals. Furthermore, it also receives and stores mediation data to database via message broker and sends outage alarms to other systems. Last but not least, there is a frontend service which serves only JavaScript and static files to users and the frontend application on users's machine communicates with API service directly. The whole overview of the architecture is depicted on 3.10 and interesting parts will be described in following sections.

3.4 Monitoring daemon

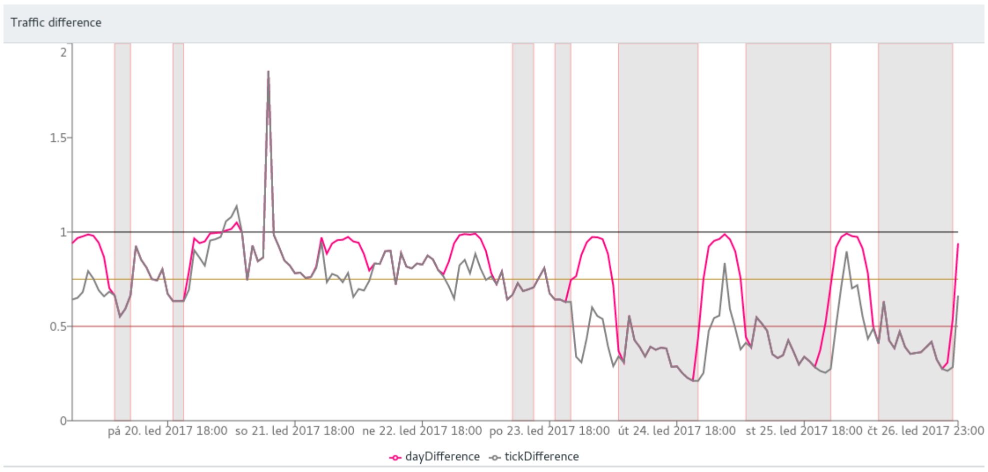
Monitoring daemon is with database the most crucial service and apart of monitoring the flows it also communicates with other systems to receive mediation traffic data and send outage alarms. Daemon is composed of several components which are started immediately upon daemon starts and are described in following subsections.

3.4.1 Consuming mediation flow data

Receiving mediation traffic data is essential for the monitoring analysis. Data are transported via Kafka messaging system to achieve smooth and real-time delivery. Data receiver is implemented in `integration` package in class



(a) Traffic



(b) Traffic difference

Figure 3.9: Outage detection of SMS - SMSCCTX from 20th of January 2017

`MediationDataConsumer`. Right after daemon starts, this class depending on configuration, connects to specified Kafka server and topic. The incoming messages are expected to be in JSON format and follow structure specified in figure 3.2. The data are then stored into appropriate collection and minute document in the database.

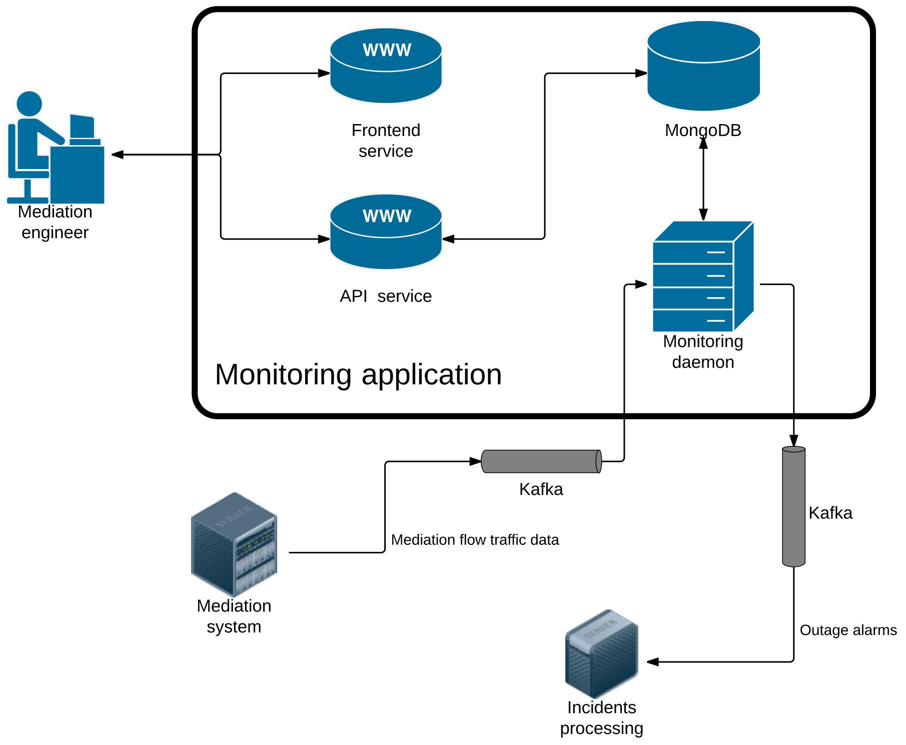


Figure 3.10: Application's architecture

```
{
"country": <String: "CZ"|"DE"|"NL"|"AT">,
"lobName": <String: name of LoB>,
"type": <String: "inputs" | "forwards">,
"flowName": <String: name of flow>,
"dataSize": <Integer: number of bytes transferred>,
"time": <String: time in format %d.%m.%Y %H:%M:%S>
}
```

Listing 3.2: Incoming mediation traffic message schema

3.4.2 Analyzing flows

Another component is called `MediationScheduler`. This component periodically executes `MediationAnalyzerExecutor` every 15 seconds and `DiscoverFlowsExecutor` every hour. `MediationAnalyzerExecutor` checks each flow and sends the flow in a queue if last completed tick has not been analyzed yet. Then there are workers waiting to analyze any flow that appears in the queue. The actual number of workers can be specified in application's configuration file. Every flow in the queue is processed by exactly one worker instance and analyzed using the outage detection algorithm. The resulting

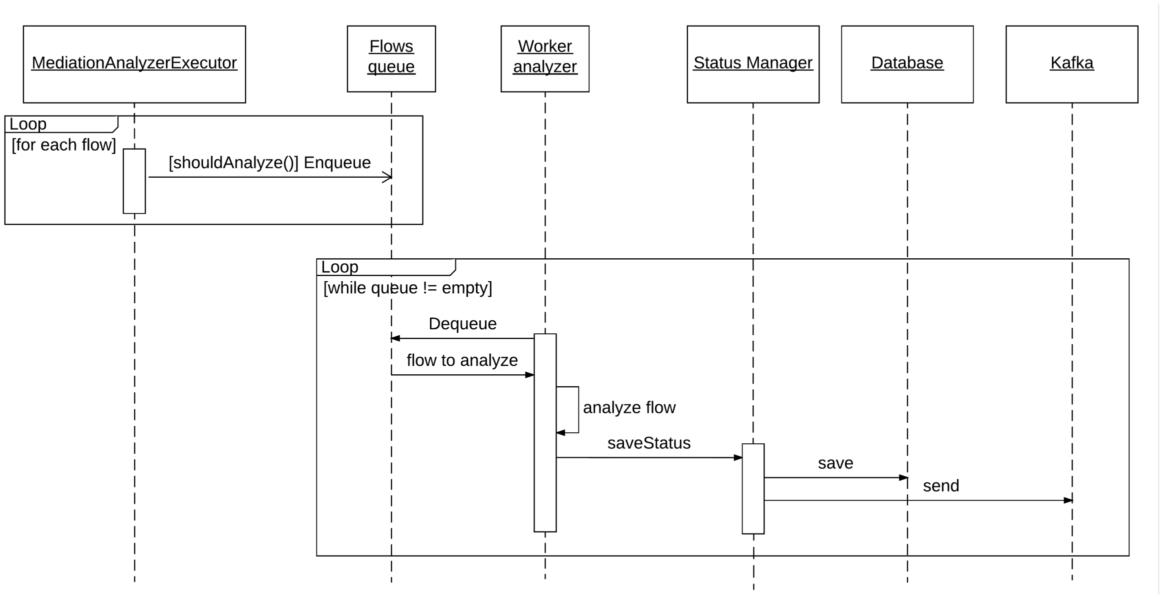


Figure 3.11: Flow analysis process

flow's state is saved into database and if the state is different from previous one, other systems are notified via Kafka and a record of the change is created in `events` collection. Sequence diagram of this process is sketched in figure 3.11. The processing model used here is called producer-consumer and takes advantage of multiple execution threads. Performance tests have shown that analyzing all 4500 flows with only one worker takes about 10 minutes but using 5 workers reduces the running time by half to 5 minutes. There is also a possibility to extract workers from this daemon and make them as a stand-alone services, which would then allow for easy horizontally scaling across multiple nodes and thus increasing the performance linearly. However, the running time of five minutes does not require such a solution since the lowest granularity of a flow is fifteen minutes now and I did not want to complicate the deployment at this moment. Furthermore, scanning all flows is necessary only when the applications starts for the first time or at midnight, when even flows with 1 day granularity need to be analyzed. Nevertheless, changing into stand-alone worker services will not require much of additional work once it is required.

3.4.3 Triggering alarms

A flow can be in three states: OK, WARNING and OUTAGE. Any change between these states is emitted via messaging system to other systems and especially a change to OUTAGE state is important. Sending out notifications is managed by `MediationStateProducer`. It is a singleton class instantiated at start of the monitoring daemon. At first, it tries to connect to Kafka server specified in the configuration and if it fails, retry will follow in two minutes intervals. In order to not lose any emitted messages even when the producer is not yet connected to Kafka server or when the connection breaks for a while, `MediationStateProducer` keeps all messages in its own queue until they are successfully sent. There is a separate thread which reads the queue and submits them to Kafka. Moreover, first message that fails to be sent to Kafka causes `EmailSender` to send an informative email regarding the issue to an email address specified in configuration file. The schema of flow state change message is shown in figure 3.3

```
{
"system": "mediation",
"country": <String: "CZ"|"DE"|"NL"|"AT">,
"lobName": <String: name of LoB>,
"flowName": <String: name of flow>,
"tickTime": <String: time of tick in ISO 8601 format>,
"time": <String: time of creation in ISO 8601 format>,
"newStatus": <String: "OK"|"WARNING"|"OUTAGE"|"N_A">,
"previousStatus": <String: "OK"|"WARNING"|"OUTAGE"|"N_A">,
}
```

Listing 3.3: Outgoing flow state change message schema

3.4.4 Discovering new flows

`DiscoverFlowsExecutor`'s goal is to accommodate requirement for adding and removing flows automatically (F2). This component is executed every hour and looks at flow's which were active in last day. When a new flow not yet in `config` collection is discovered, this component creates a record for it with default configuration settings and thus immediately schedules it for analysis in next `MediationAnalyzerExecutor` round.

3.4.5 Monitoring of daemon's components

Even though the application's task is to monitor mediation flows, the application itself must also be monitored to ensure everything works correctly. For monitoring of Kafka consumer and producer, `ComponentMonitoring` is executed every 10 seconds and stores current state of consumer and producer into database. It can be then accessed by API service when a request for application's status is received. Moreover, each time `MediationAnalyzerExecutor` is executed, a current timestamp is stored in the database so other system can be sure that it is up and running.

3.5 API service

Monitoring daemon and database would be enough just for monitoring the flows. In order for the application to be fully usable, it also needs to provide ways for mediation engineers to examine and visualize traffic on mediation flows, determine and set the right settings for each flow and allow other systems to consume traffic data or application's status in an easy way. API service exists to satisfy these needs.

While many APIs are called RESTful, not many of them actually meet all six requirements of RESTful architecture. It can be said, that this API service is stateless, follows HTTP methods usage and resources naming conventions and uses JSON format.

3.5.1 Authorization

The API service implements authorization to meet the functional requirement F6. Any request to the API must contain valid API key in `X-API-KEY` header, except for `/login` endpoint, which returns API key in response to valid username and password. Wrong or missing API key results in 401 response status code and request with insufficient permissions gets 403.

3.5.2 Endpoints

There is over fourty endpoints and all are divided into four categories: app, config, data, flows.

App /app Endpoints under this path are related to general application's functionality. There are endpoints for getting status of the monitoring application, getting recent status changes of flows (called events), getting current time of the application, authenticating users into the system as well as creating and modifying them.

Config /mediation/config Endpoints under this path allow modification of metadata of countries such as national holidays or lazy days.

Flows /mediation/flows These endpoints allow mediation engineers to configure each LoB or flow. LoB or flow can be enabled or set with new configuration parameters, such as granularity, alarm levels or type of difference to be used. If a GET request for country, LoB or specific flow contains query parameter `includeStatus=true`, in addition to configuration the response will also contain their status. New flows or LoBs can be also added or even deleted. The endpoints follow the hierarchical structure of flows (country, LoB, flow) and REST API resource naming conventions, so for instance, resource of a single flow is located at `GET /<country>/<lobName>/<flowName>`.

Data /mediation/data There are two endpoints in this path. `POST /query` servers traffic data for flow based on granularity and other parameters in the requests's body. This endpoint is used by frontend application when rendering traffic, difference and outage graph. The response contains a list of ticks with traffic, differences, status and expected traffic. Furthermore, the response also contains an object with metadata. An example of request and response can be seen on 3.4. Next data endpoint is `POST /insert`, which can be used to insert mediation traffic data from csv file into database as an alternative to Apache Kafka. Additionally, data can also be stored to database via command-line interface tool, which has far better performance and is not limited to HTTP server's maximum uploaded file size limit.

```
POST /query
{
  "from":"01.02.2017",
  "to":"15.02.2017",
  "country":"CZ",
  "lobName":"GSM",
  "flowName":"MSSBRN1A",
  "granularity":0
}

Response:
{
  "data": [
    {
      "MSSBRN1A": 188385532,
      "_id": "2017-02-01T00:00:00+01:00",
      "dayAverage": 1162595297.6666667,
      "dayDifference": 1.023,
      "expected": 161627916,
      "status": "OK",
      "tickDifference": 1.166
    },
    ...//object for each tick withing the range
  ],
   "metadata": {
    "flowName": "MSSBRN1A",
    "granularity": 480,
    "metrics": {
      "GSM": {
        "type": "traffic"
      },
      "dayAverage": {
        "type": "traffic"
      },
      "dayDifference": {
        "type": "difference"
      },
      "expected": {
        "type": "traffic"
      },
      "status": {
        "type": "other"
      },
      "tickDifference": {
        "type": "difference"
      }
    }
  }
```

Listing 3.4: Illustration of a request for getting flow's traffic data and its response

3.6 Data model

Since document databases allow to store hierarchical data easily, I took full advantage of it and the model of structures stored in database is very similar and usually identical to structures used on application layer. As a result, little to none mapping between application and database entities need to be done.

3.6.1 Traffic data

Flow mediation traffic data are stored in database named `mediation_data` in a collection `traffic`. The model used for storing traffic data is identical to the one already described in document databases section 3.1.1.2.2 where different types of databases were evaluated and document database with used model was chosen as the best solution.

3.6.2 Flows configuration

Next, the application uses `mediation` database, which keeps all other collections. `Config` collection has one object with key *mediation* and contains configuration of countries (national holidays and lazy days) and all flows. A single configuration of flow is simple object with following properties:

granularity Integer: Length of tick in minutes to be used for analysis.

hardAlarmLevel Double: Level of hard alarm. Value between 0 and 1.

softAlarmLevel Double: Level of soft alarm. Value between 0 and 1.

difference String: Type of difference to be used for analysis. Value either `day` or `tick`.

enabled Boolean: Whether the flow should be monitored.

independentDays List of integers: Days which should be handled independently when computing expected value. Monday to Sundays equals to 0 to 6 respectively.

minimalExpectation Integer: Minimal amount of bytes transmitted on this flow in one tick. Default value is 1 and it mimics the old monitoring system's behavior 1.4.

As was discussed earlier, there are two types of flows: input flow and forward flow, and both must be assigned to LoB. It was discovered that flows in one LoB usually have similar characteristics, such as granularity or suitable alarm levels. Moreover, forward flows are depending on exactly one input flow (they forward data from input flow to other system) and are somewhat subordinates of inputs flows. To ease and reduce necessary configurations, I have decided to model configuration data model to respect this hierarchy by implementing configuration inheritance. In case forward flow does not have configuration specified, it will inherit configuration of connected input flow. If input flow does not have configuration specified, it will use configuration of LoB and if LoB has no configuration, it will use default configuration specified in the application's code.

3.6.3 Analysis results

The monitoring daemon performs analysis of flows in interval specified by flow's granularity. For each flow, the results of analysis are stored into database `mediation` collection `statuses` and document `flows`. Any change in status is also recorded in `events` collection. This data is then access by frontend application via API service when it displays dashboard or list of flows. The status object contains name of flow, tick time which was analyzed, time of analysis, traffic level and status. `Events` collection have one document per event with automatically generated id. Each event document contains time of event, country, message, tick time, previous and new status and flow name.

3.6.4 Daemon's components monitoring

As was mentioned, it is necessary to assure that all monitoring daemon's components are running. For this purpose, their status is periodically or at the time of execution written into `system` document in `statuses` collection together with time. Similarly to analysis results, this document is accessed by API service whenever other systems or scripts monitoring this application request it.

3.6.5 Users

Last but not the least collection is `users`. Each document represents one user and contains its login name, password hash, account type (user or app), API key and permissions (root or readOnly).

3.7 Backend Implementation

In this section I would like to describe parts of backend implementation that were not mentioned in API service or monitoring daemon sections. The whole application is contained in one git repository and monitoring daemon and API service are located in backend directory. Since many parts of the API service and daemon are common (especially queries to database and flow analyzing algorithm), they share most of the codebase. Monitoring daemon is started by running `daemon.py` and it starts scheduler from monitoring module `mediation`, Kafka producer and consumer. API service is started by `api_service.py` which starts Flask HTTP server on port 5000 and exposes the API endpoints.

3.7.1 Database queries

Both, API service and monitoring daemon must access traffic data in the database. Queries from API service are typically much larger since frontend displays data of certain date range; maximum allowed range is 28 days. Data queries are located in module `mediation.data_query` with most general queries communicating with database in `mediation.data_query.engine`. The following list provides overview of queries used by the API service and monitoring daemon and their relationships are depicted on class diagram 3.12. All queries used by API service are used for getting data for frontend applications' charts.

DatesQuery For given dates and flow it retrieves traffic data from database.

DateRangeGroupQuery Accepts 2 dates in constructor and with the help of DateQuery retrieves traffic data for all days between them.

ExpectedTrafficQuery This query calculates expected traffic for the given flow and a date. To get traffic data of past days it uses `DatesQuery`.

FlowLevelDateRangeQuery For provided dates range this query first calculates expected traffic for each date by using `ExpectedTrafficQuery` and then computes differences between expected and actual traffic. This query is called from API data query endpoint only and to boost the performance, traffic data for each day are loaded once and provided in constructor.

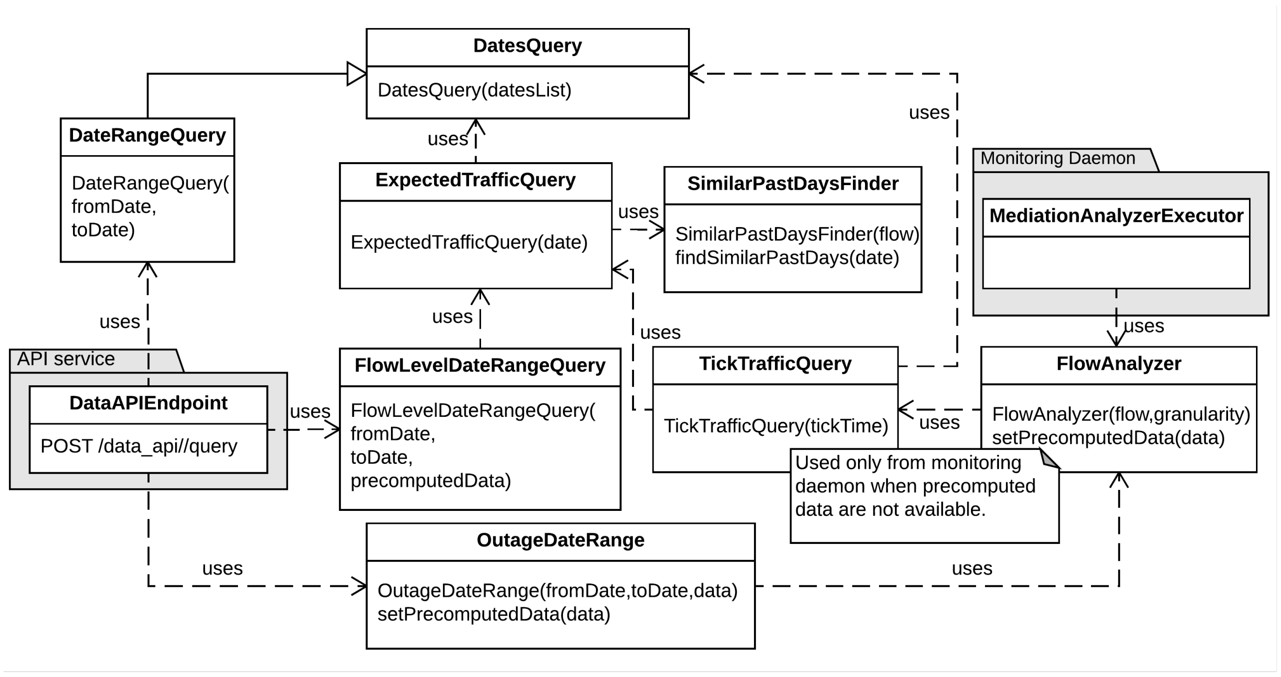


Figure 3.12: Class diagram of traffic queries

OutageDateRangeQuery Another query used by API data query endpoint only. For each tick in date range, this query calls `FlowAnalyzer` and retrieves its status. Actual and expected traffic data retrieved earlier in the endpoint are passed down to `FlowAnalyzer` and `OutageDetector` to improve performance by avoiding repetitive queries for the same data.

TickTrafficQuery This query is used by `OutageDetector` in cases it does not have cached data for the tick - it is executed only from monitoring daemon, where only current tick is evaluated. The query retrieves current and expected traffic for tick provided in constructor.

SimilarPastDaysFinder This class is used by `ExpectedTrafficQuery` to find similar days which should be used for calculating the expected traffic.

3.7.1.1 Daylight saving time

It can be seen that all queries are using `DatesQuery` in some way to retrieve data. The principle on which `DatesQuery` works was explained earlier in 3.1.1.2.3. It uses MongoDB aggregation framework, but unfortunately it turned out to be problematic for retrieving traffic data across summer and winter time changes. All times in MongoDB are saved in UTC and the aggregation framework groups records based on UTC times. To align ticks with

Prague time	CET	CEST	UTC
0:00	0:00	1:00	23:00
1:00	1:00	2:00	0:00
3:00	2:00	3:00	1:00
4:00	3:00	4:00	2:00
5:00	4:00	5:00	3:00

Table 3.2: Daylight saving time change on 26th of March 2017

start of a day, I was subtracting 60 minutes from each datetime before grouping stage. That works as long as the times are only in CET timezone, which is only one hour ahead of UTC. Once the time changed to CEST timezone, this no longer worked for queries with dates spanning across CET and CEST timezones and tuning MongoDB aggregate query to subtract 60 or 120 minutes depending on the specific date and time is too difficult if not impossible. Instead, I have decided to use the aggregate framework to group data by maximum of 60 minutes and all further processing (grouping and setting the right timezone) is done in Python. It still significantly reduces the amount of data returned from MongoDB and at the same time allows easier handling of timezones.

Nevertheless, change to summer time can still be troublesome for ticks of two hours. The 2:00 hour is simply left out 3.2 and thus the ticks for 26th of March 2017 are 0:00, 3:00, 4:00, 6:00. Naturally, the 3:00 (or could be also called 2:00) tick consists of just one hour and thus is going to have less (half) traffic than expected. Since most of the flows are using day difference it does not cause any outage alarms, but in the future it might be needed to separately handle this very special corner case that occurs once in a year on flows with 2 or 3 hours granularity settings. Functions for handling ticks time are located in `util.py` file and are using 'Europe/Prague' timezone which stands for UTC+1 or UTC+2 depending on specific date and time. Change from summer to winter time is not so bothersome, because one hour is added so the ticks are 0:00, 2:00, 4:00 and the 2:00 tick consists of three hours in total - that certainly will not cause any outage. Interestingly, the stakeholders confirmed that there are always troubles with at least few systems or scripts that stop working as a result of daylight saving time change. Handling time changes correctly is surely a big challenge for all developers.

3.7.2 Time shift feature

Live stream of mediation traffic does not always have to be available, especially in development and testing phase. For this reason I have decided to implement time shift feature so that the whole application can be simulated as if it was working on real-time data without actually having access to it. The whole application is using `AppConfig.getCurrentTime()` function to get current time, which can return time shifted backwards by defined number of days in configuration file. Historical data can be inserted into database via csv files and special scripts.

3.7.3 Application configuration file

Aside of configuration of flows which is stored in database, the more technical parts of application are configured via config.json file. It is located in root directory and uses JSON format. It contains information about number of threads used by mediation analysis, credentials to MongoBD database, Kafka servers and names of topics to connect to and also necessary information for sending emails in cases that sending messages over Kafka is not working. Example of full configuration file is shown at listing C.2.

3.8 Frontend application

In this section I will describe application's user interface and its implementation details.

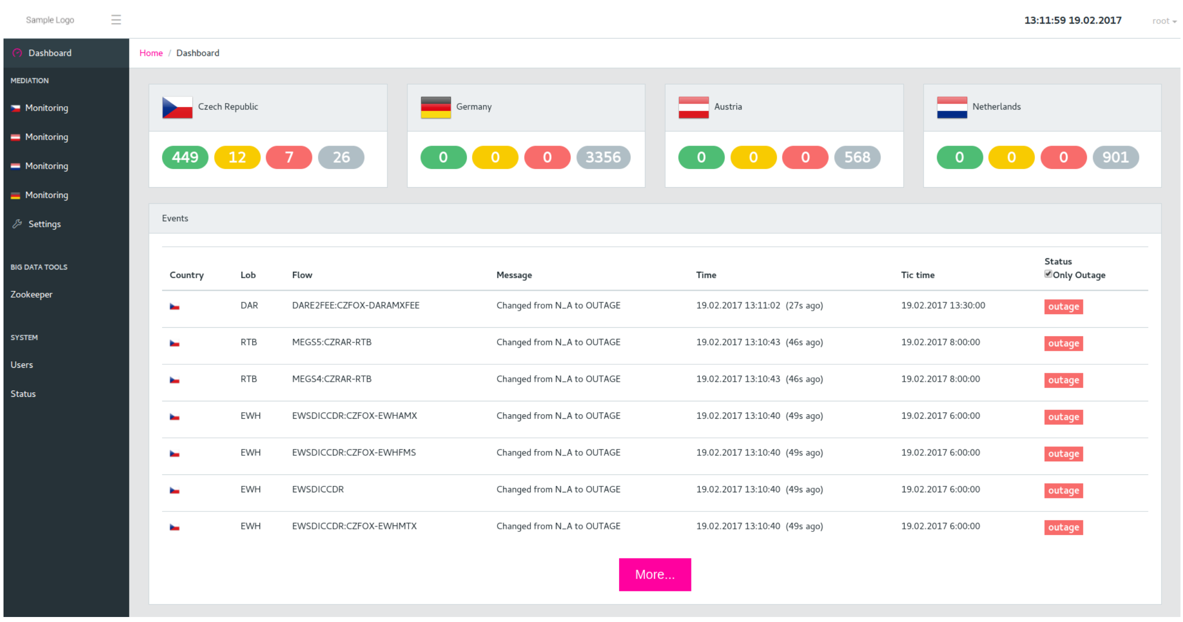
3.8.1 User interface

The main purpose of the application is to reliably run on a background and monitor mediation system. The user interface is going to be used by about ten operation engineers mainly to set flows' configuration and occasionaly inspect traffic when problems occur. Due to low number of final users and frontend application being less crucial than backend, less effort was devoted to the frontend's development process. As a result, some phases such as a design of mockups and a creation of prototypes were omitted. The application's appearance was initially created based on expected use cases and then continuously developed and adjusted according to target user group's feedback.

From the beginning it was apparent that the application should give a quick overview of mediation system's status upon login. As a solution, dashboard is shown at home page 3.13. On first sight it gives overview of how many flows are healthy or failing in each country. Furthermore, the dashboard includes event log and allows user to show only recent severe events.

3.8.1.1 Flows

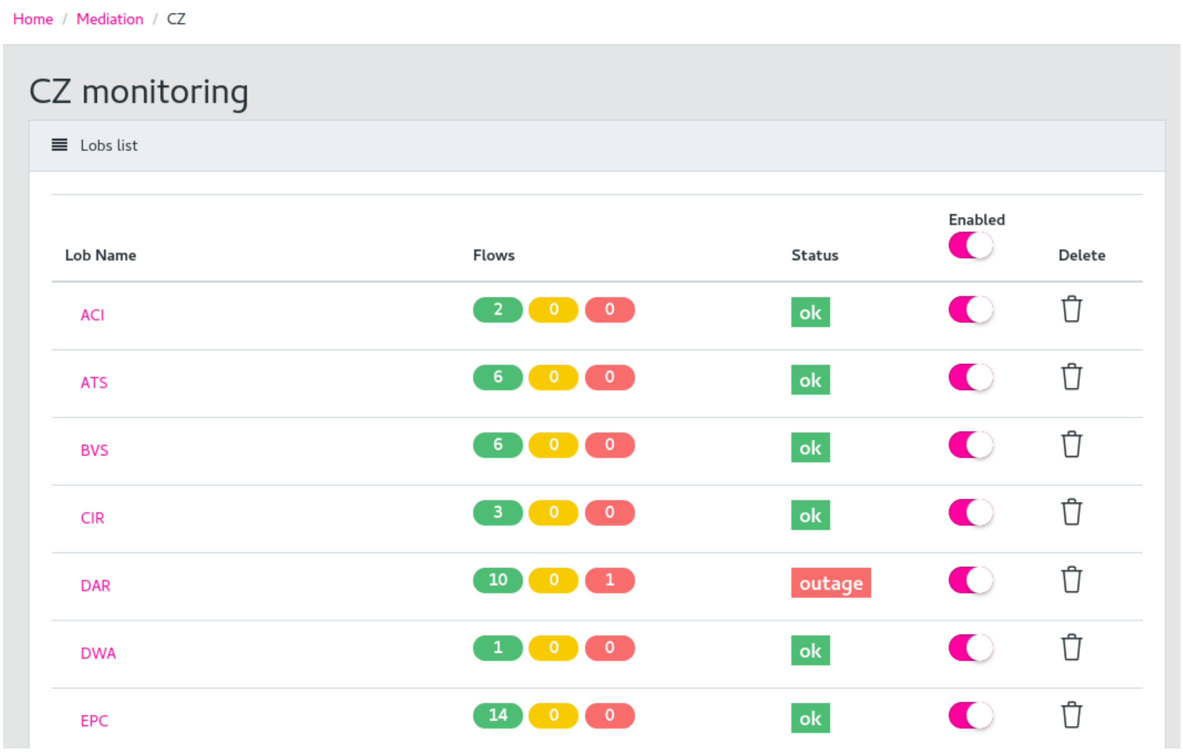
Left menu bar provides main navigation and user can go to country detail, settings, user management or status page of the monitoring application. Country detail shows list of LoBs and summary of flows statuses 3.14. LoBs can be added, disabled or removed completely. Disabling LoB makes all flows in the respective LoB disabled, too.



Events

Country	Lob	Flow	Message	Time	Tic time	Status ☑Only Outage
	DAR	DARE2FEE:CZFOX-DARAMXFEE	Changed from N_A to OUTAGE	19.02.2017 13:11:02 (27s ago)	19.02.2017 13:30:00	outage
	RTB	MEGS5:CZRAR-RTB	Changed from N_A to OUTAGE	19.02.2017 13:10:43 (46s ago)	19.02.2017 8:00:00	outage
	RTB	MEGS4:CZRAR-RTB	Changed from N_A to OUTAGE	19.02.2017 13:10:43 (46s ago)	19.02.2017 8:00:00	outage
	EWH	EWSDICCDR:CZFOX-EWHAMX	Changed from N_A to OUTAGE	19.02.2017 13:10:40 (49s ago)	19.02.2017 6:00:00	outage
	EWH	EWSDICCDR:CZFOX-EWHFMS	Changed from N_A to OUTAGE	19.02.2017 13:10:40 (49s ago)	19.02.2017 6:00:00	outage
	EWH	EWSDICCDR	Changed from N_A to OUTAGE	19.02.2017 13:10:40 (49s ago)	19.02.2017 6:00:00	outage
	EWH	EWSDICCDR:CZFOX-EWHMTX	Changed from N_A to OUTAGE	19.02.2017 13:10:40 (49s ago)	19.02.2017 6:00:00	outage

Figure 3.13: Application's dashboard

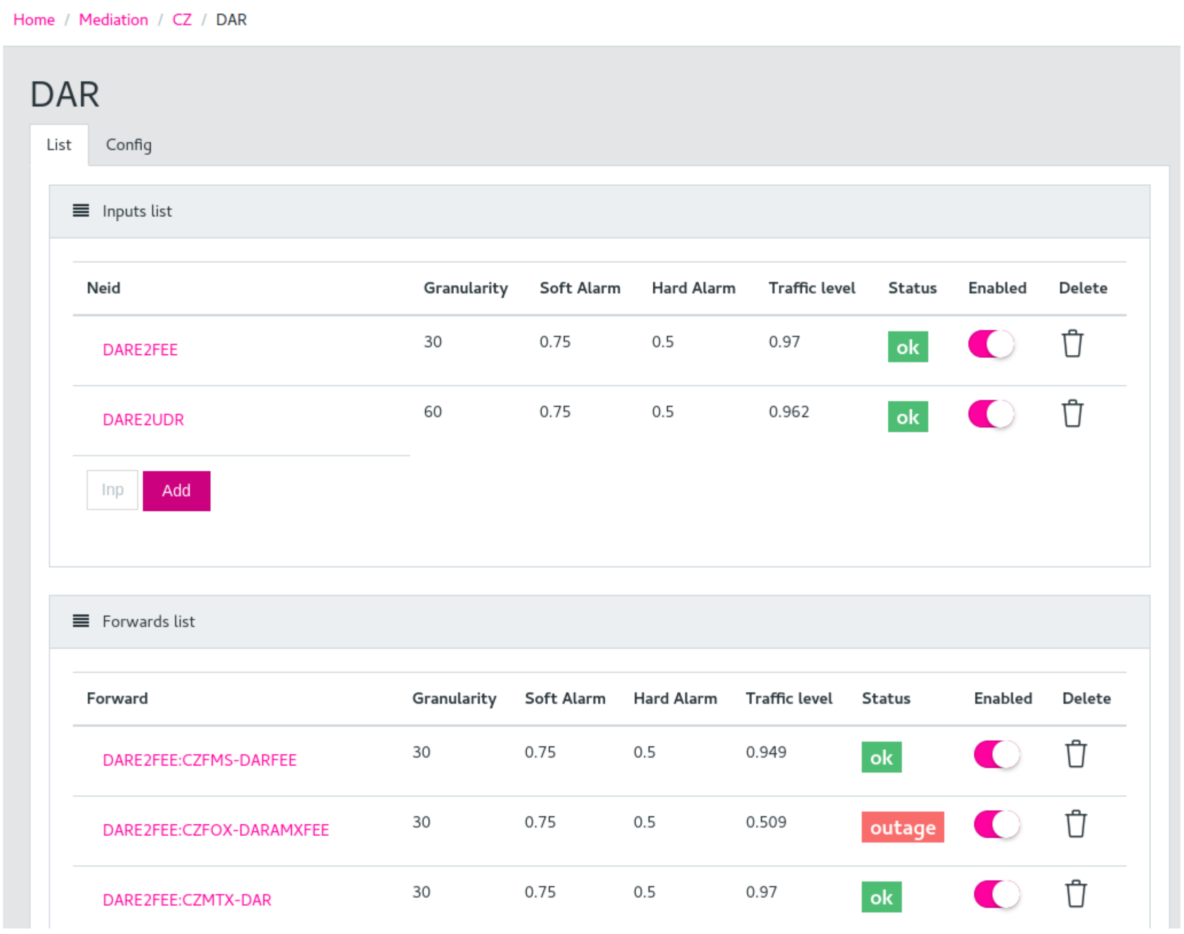


CZ monitoring

Lobs list

Lob Name	Flows	Status	Enabled	Delete
ACI	2 0 0	ok		
ATS	6 0 0	ok		
BVS	6 0 0	ok		
CIR	3 0 0	ok		
DAR	10 0 1	outage		
DWA	1 0 0	ok		
EPC	14 0 0	ok		

Figure 3.14: Country detail


Home / Mediation / CZ / DAR

DAR

List | Config

Inputs list

Neid	Granularity	Soft Alarm	Hard Alarm	Traffic level	Status	Enabled	Delete
DARE2FEE	30	0.75	0.5	0.97	ok		
DARE2UDR	60	0.75	0.5	0.962	ok		

Inp Add

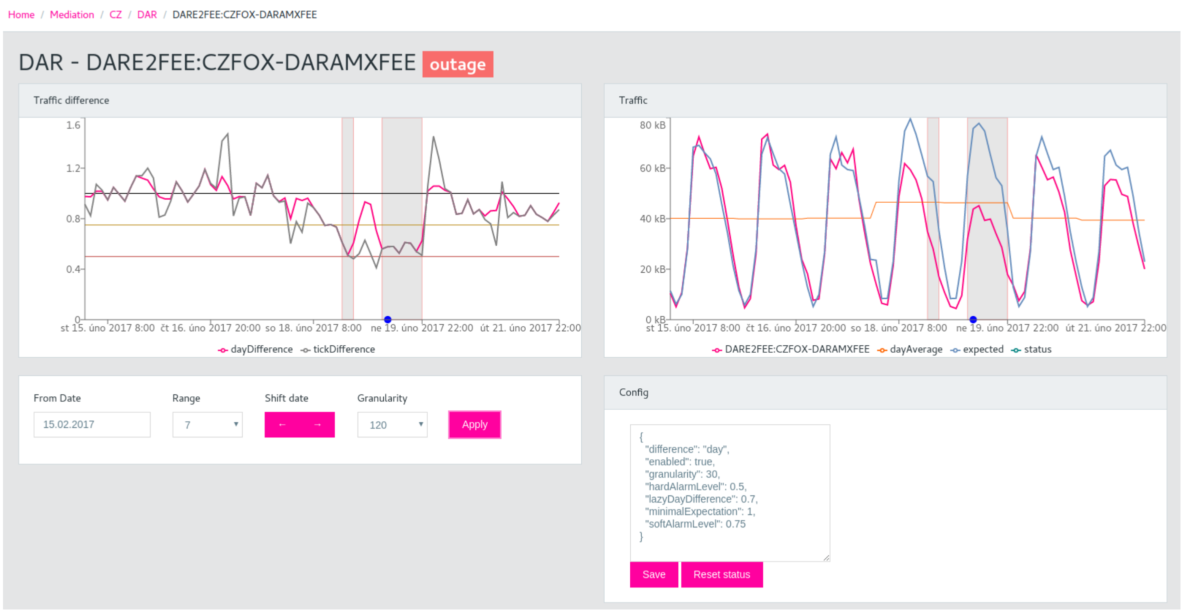
Forwards list

Forward	Granularity	Soft Alarm	Hard Alarm	Traffic level	Status	Enabled	Delete
DARE2FEE:CZFMS-DARFEE	30	0.75	0.5	0.949	ok		
DARE2FEE:CZFOX-DARAMXFEE	30	0.75	0.5	0.509	outage		
DARE2FEE:CZMTX-DAR	30	0.75	0.5	0.97	ok		

Figure 3.15: LoB detail

Similarly to country details, LoB detail presents flows divided into input and forward lists 3.15. Each flow's basic settings and status is shown and they can be added, disabled or removed. LoBs default settings can be configured in Config tab.

Most important page is probably detail of each flow. This page shows specific information about flow's configuration and traffic level including historical traffic data 3.16. Left chart shows both types of difference and right chart renders the actual traffic. Below differences chart is a control panel for charts, where user can select granularity and days to display. Special thought was given to a way how flows are configured. At first, I created HTML components for each configuration property, such as dropdown for difference types, input fields for alarm levels and so on. However, as number of configuration properties was varying it turned out that best way is to let users modify configuration directly via JSON format. Not only it saves time when adding new configuration properties, but mainly it is very transparent, understand-



From Date	Range	Granularity
15.02.2017	7	120

```
{
  "difference": "day",
  "enabled": true,
  "granularity": 30,
  "hardAlarmLevel": 0.5,
  "lazyDayDifference": 0.7,
  "minimalExpectation": 1,
  "softAlarmLevel": 0.75
}
```

Figure 3.16: Flow detail - traffic graphs and configuration

able and easier approach for users, who are technically competent. Moreover, it implicitly allows very easy copying of configurations, which would not be possible with HTML components. The frontend only checks that specified configuration is in a valid JSON format.

When charts are rendered for some time period using defined configuration, detected outages are indicated with gray squares with red border lines. Even though the data comes from API service, the outage detection algorithm is shared between API service and monitoring daemon and thus allows precise simulation of how the actual monitoring analysis will work. This is considered as a very important and useful feature since the user can fine-tune the configuration on past data and get better estimate about how it is going to perform in the future.

3.8.1.2 Country settings

Mediation settings page implements configuration of countries. Since each country has mostly different holidays and expected lazy days, each country is configured independently in tab. For a while I was considering to use online service like holidayapi.com to fetch national holidays data, however, the application is expected to run without connection to public internet, which makes this option unfeasible. Instead, each country's national holidays and lazy days are configured manually on this page. I consider this as the biggest weakness of the application's user experience and in the future it could be improved by an algorithm computing lazy days automatically.

3.8.2 Implementation

Since React is mostly a basic library and many other libraries are necessary for a production ready application, I decided to use a React template project to overcome difficulties in plumbing all React libraries to work together. After evaluation of several templates I have decided to use CoreUI [31], which says to be "Bootstrap Template built as framework" and already includes necessary libraries such as reactstrap, react-router and many others.

The frontend application's view classes are divided into 2 main packages: components and views. Components package contains individual reusable components such as `StatusBadge`, `ChartControl`, `FlowChart` and so on. These components are then used together in classes from views package, which represent individual pages. For instance, there are views like `Dashboard`, `FlowSettings`, `LoBDetail` or `FlowDetail`. Navigation and routing between view pages is managed by react-router library and routing rules are specified in `router.js`.

Redux library is used only for storing user details since it needs to by accessed by many components, especially the permissions attribute. Currently, each view like `LoBDetail` or `FlowDetail` executes own API request to get status of particular LoB or flow. This could be further optimized so the statuses are downloaded once, stored globally with Redux and access by all components in the application. On the other hand, the navigation speed between pages is still smooth enough and this inefficiency is negligible.

For accessing API service there is a class `Api.js`, which adds necessary API key header to all requests. Other helper methods are implemented in `Util.js`.

3.8.2.1 Authorization

When user visits the application for the very first time, a login screen is presented to him. Upon submitting valid credentials, API service returns API key which is saved in browser's local storage. At next visit (or page reload), API key is read from local storage and its validity is verified against API service.

When users authorization and management was implemented, a request for anonymous access with read only permission was raised by the stakeholders. The motivation behind this is that mediation engineers, who only want to inspect flow's traffic and not to modify any configuration, does not have to remember any login credentials. As a solution, the application creates user account named *visitor* with readOnly permissions at the first run together with root user. Login screen then contains button *Login as visitor* which triggers request to `/visitorLogin` endpoint and returns API key of the visitor user. The rest of application then works as expected. The visitor user can be deleted as any other user from users management page.

Chapter 4

Design and Implementation of ZooKeeper Monitoring

Due to the fact that main focus of the application is put on monitoring of mediation system, ZooKeeper monitoring is considered to be a proof of concept. The resulting codebase should be modularizable and allow simple removal of ZooKeeper monitoring module since it is not planned to be used in production by the stakeholders. In this section I will focus on ZooKeeper monitoring specifics and describe how it was implemented into the application.

4.1 Outage detection

Monitoring of ZooKeeper turned out to be far easier than monitoring of mediation flows. The main difference is that it is not necessary to create algorithm for calculating expected traffic or value. The status of ZooKeeper cluster strictly depends on number of nodes and their individual statuses. For ZooKeeper, it is crucial to have more than half of nodes working and connected with each other to form a working quorum. Once a majority of nodes is online ZooKeeper is guaranteed to work properly. Naturally, there might always be bugs in the system which may cause ZooKeeper to not work as expected, however, that is not responsibility of the monitoring tool detect it. The monitoring algorithm thus only needs to periodically check all defined nodes, fetch their statuses and based on that determine status of the whole cluster. If all nodes are OK, the cluster status is set to OK. If not all but still more than half of nodes are OK, cluster's status is set to WARNING and less than half of nodes in OK results in cluster's status to be OUTAGE.

4.1.1 Fetching status

ZooKeeper is distributed system with no central node which would aggregate the cluster's information. Each node listens on standard port 2181 for incoming TCP connections and understands several commands. One of the commands is named `stat` and its response contains various information about the node 4.1. If the node is not connected to quorum, no response at all or "This ZooKeeper instance is not currently serving requests" is returned.

```
ZooKeeper version: 3.4.8, built on 02/06/2016 03:18 GMT
Clients:
 /127.0.0.1:40638[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 2
Sent: 1
Connections: 1
Outstanding: 0
Zxid: 0x3b00000000
Mode: follower
Node count: 10
```

Listing 4.1: ZooKeeper stat command response

4.2 Monitoring daemon

The ZooKeeper monitoring module follows architecture pattern and principles introduced by mediation monitoring. Monitoring daemon is fetching nodes' statuses, writes them into database together with the cluster's status and sends message over Kafka if necessary.

At the start of monitoring daemon, `ZookeeperAnalyzerExecutor` is scheduled to run every five seconds. It sends `stat` command to each node instance specified in `config` collection in document `zookeeper`. Various stats are parsed from the response using regular expressions and node's status is determined. Analogously to flow statuses, it is then stored to collection `statuses` in document `zookeeper` together with whole cluster's status.

4.3 API service

All ZooKeeper related endpoints are under the path /zookeeper/.

/node/{string:socketAddress} POST request adds new ZooKeeper node to configuration and DELETE request deletes it.

/cluster GET request returns configuration of the cluster: all nodes and whether monitoring is enabled.

/cluster/enabled POST request enables monitoring.

/cluster/disable POST request disables monitoring.

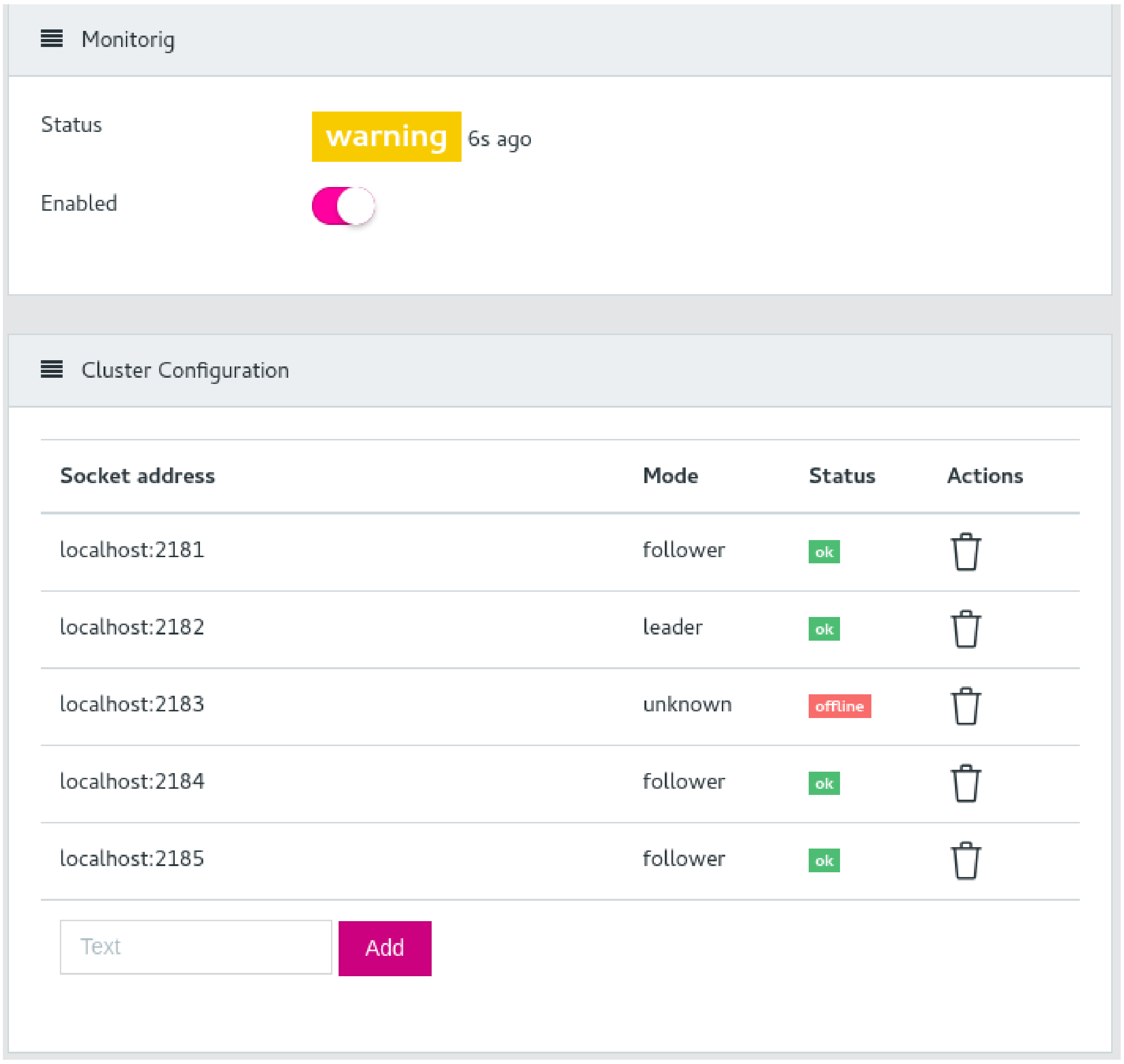
/status GET request returns statuses of whole cluster and all nodes.

4.4 Data model

The configuration document contains ip address and port of all node instances in ZooKeeper cluster and boolean attribute if the monitoring is enabled. Document in `statuses` collection contains time of analysis, cluster's status and status of each configured node.

4.5 Frontend application

Frontend application is extended by one page, which works as both, a configuration and statuses overview page. It contains a list of nodes together with their status and mode. Each node can also be removed or new node added. Cluster's status is displayed above the nodes list 4.1. Since ZooKeeper's status can change in few seconds and cluster analysis is executed every five seconds, frontend application should show the most recent data. As a solution, basic HTTP polling is used, which means a request is sent to server every few seconds. Nicer and more efficient solution would be to use push technology: the server would send message to frontend application whenever status of some node changes. On the other hand, as a proof of concept solution and given the expected number of users, HTTP polling is sufficient and works well.



Status warning 6s ago

Enabled

Socket address	Mode	Status	Actions
localhost:2181	follower	ok	
localhost:2182	leader	ok	
localhost:2183	unknown	offline	
localhost:2184	follower	ok	
localhost:2185	follower	ok	

Figure 4.1: ZooKeeper monitoring

Chapter 5

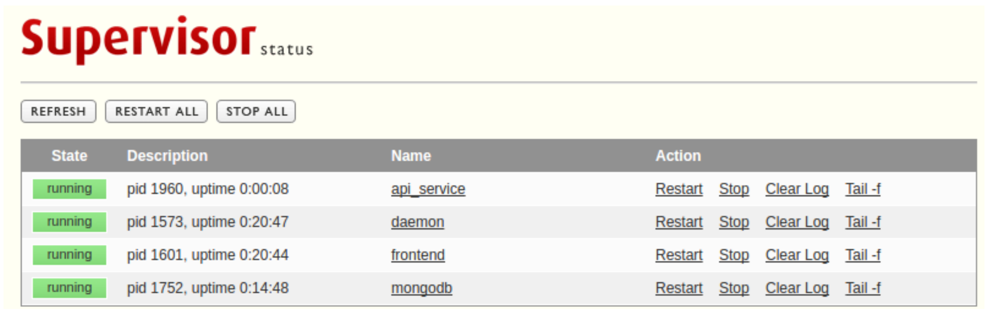
Deployment

Deployment is one of the last essential steps of software development. In this chapter I will describe how the application can be ran and what types of deployments are possible.

5.1 Deploying the application

There are two options how this application can be deployed and ran. For quick and easy deployment I have prepared a Docker image. Docker is a platform similar to virtual machine, but instead of simulating the whole operating system it uses the host OS's Linux kernel and thus has significantly better performance and smaller image size. Typically, one Docker image corresponds to one service or process, which in this case would mean total of four Docker images and might require some Docker container orchestration tool such as Kubernetes or Rancher. I did not want to overcomplicate the Docker deployment and decided to put all four processes into a single image, which for the application that is not expected to scale horizontally at this moment works fine. Dockerfile used to create application's image is shown at listing C.3. It exposes ports for database (27017), API service (5000) and frontend service (9000). On port 9001 listens Supervisor, which is described in the next section.

Another option is to deploy the application directly to a host OS. The big disadvantage here is that over fifteen dependencies must be installed on the OS, including Python modules and compatible MongoDB version. All dependencies are specified in Appendix A. Once they are installed, the application can be simply checked out from git repository or unpacked from zip file.



State	Description	Name	Action
running	pid 1960, uptime 0:00:08	api_service	Restart Stop Clear Log Tail -f
running	pid 1573, uptime 0:20:47	daemon	Restart Stop Clear Log Tail -f
running	pid 1601, uptime 0:20:44	frontend	Restart Stop Clear Log Tail -f
running	pid 1752, uptime 0:14:48	mongodb	Restart Stop Clear Log Tail -f

Figure 5.1: Supervisor control panel

5.2 Running the application

The application consists of four independent processes (or services) that all must be started in order to work properly. Furthermore, it is important to restart them if they crash, give overview of their status and store logs. One way to do this is by creating custom scripts, but that can be quite demanding task if done properly. Other option is to use some already implemented solution and I have decided to use Supervisor [32]. It is a ”client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems” and fits perfectly for this task. All Supervisor’s configuration is written in `supervisor.conf` file like processes’ start commands or log files paths. For this application I configured four processes to be ran under the Supervisor. I was nicely surprised that Supervisor automatically compresses log files and deletes too old logs. Furthermore, Supervisor runs its own HTTP server, which can be used to monitor and control processes and access most recent log messages via browser 5.1. An example of Supervisor’s config file is shown at listing C.1. After all, it was a great choice to use Supervisor to orchestrate application processes and saved a lot of time. The Supervisor is started by shell command `supervisord -c supervisor.conf`.

5.3 Deployment to production

The provider has several environments in independent networks. Prior to application being used in production, it is first ran and evaluated in test environment. Test environment can be imagined as a smaller copy of production environment which is trying to imitate most of its functionality. Unfortunately, the mediation system is not fully simulated in test environment so deploying the monitoring application there does not bring any benefit as it cannot be properly tested. The only option left is deployment to production, where the mediation system is running and data can be transported to monitoring application and analized in real time. Since the application is not critical, it can be deployed in parallel to already existing monitoring tool without having any impact on live running operations and rest of the environment.

However, deploying the application turned out to be more problematic than was initially expected. Until now, I had no work experience in large corporations and did not realize how complicated and lengthy internal processes, rules and overall bureaucracy can be. The application was ready for first deployment at the end of March and both deployment possibilities were presented to stakeholders: Docker image and direct deployment. The provider has very strict internal rules about what can be used in production environment and any software including all dependencies must be approved by security team and managers first before being installed. Unfortunately, Docker is not approved and thus this option, which is significantly easier, is not realizable. The only choice left is direct deployment to one of servers in production environment, which also showed up to be complicated. What makes the approval process even more complicated is the fact, that security team and managers are located in Germany and upon their approval, it is passed onto system administrators in Košice, Slovakia, who do the actual installation. Request for approval and installation of all dependencies specified in Appendix A has been submitted at the beginning of April and as of today, May 20th 2017, it is still pending for approval.

5.4 Deployment to DigitalOcean

Since deployment to the production environment is not possible yet, I have decided to carry out deployment into some cloud service and verify that it works. There are numerous cloud hosting services and the biggest players in the market are AWS from Amazon, Google Cloud Platform and Azure from Microsoft. All of these platforms offer tens of services for all kinds of cloud applications such as load balancers, distributed storages, message queues and many more. However, the monitoring application is going to be hosted on usual UNIX system and all these platforms are overkill. Therefore, I was looking for more simple platform with shallow learning curve. After some research I have decided to use DigitalOcean [33], which offers many preconfigured Linux systems including one with Docker. That allowed me to try also direct deployment, since DigitalOcean provided me access to virtual private server where Docker was preinstalled. To run Docker image, only `docker run --name monitoring-app -itd -v <mongoDataFilePath>:/data/db -p 9000:9000 -p 9001:9001 -p 5000:5000 -p 27017:27017 <imageName>` command needs to be executed.

Installing all dependencies is certainly more difficult. The specific steps of installation depends on Linux versions and configured software repositories. In case that requried dependency version is not present in configured repositories, it might be even necessary to compile the dependency directly on the system. When all dependencies are installed, the application is started with `supervisord` command as was described in 5.2.

Obviously, DigitalOcean deployment does not have access to the provider's production network, so integration with Kafka and live streaming mediation traffic data to the application could not be tested. On the other hand, thanks to the time shift feature the application is running and behaving as if it was in production environment except for the fact that the application's time is shifted few weeks back.

5.5 Running ZooKeeper

For testing and demonstration purposes it was also necessary to simulate ZooKeeper cluster and its outages. For this purpose I have downloaded ZooKeeper and configured five nodes on local machine. Since all nodes are running on one host they have to use different ports. The client port of node i is set to $2180 + i$ starting from 1. Each nodes's configuration also contains information about all other nodes including their ports for communication with leader and leader election. The cluster configuration is shown on 5.1. Last but not the least configuration step is to write id of each node (i) to corresponding file /data/myid. Once node is configured, it is started by `./zkServer.sh start-foreground`.

```
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
server.4=localhost:2891:3891
server.5=localhost:2892:3892
```

Listing 5.1: ZooKeeper cluster node's configuration

Chapter 6

Evaluation

The algorithm and application were described in detail in previous chapters and here I would like to provide a summary of what was accomplished and what are the results.

6.1 Mediation monitoring evaluation

Due to the fact that the application was not connected to live stream of mediation traffic data yet, it is not possible to fully evaluate its integration functionality with live data in production. However, the application can still be tested by using time shift feature or by exploring historical data. Unfortunately, even though the stakeholders provided historical incidents information to me, it could not be used for quantifying the algorithm's results because it is not apparent if particular incident influenced and was reflected by mediation traffic.

Outage detection algorithm's results highly depend on correctly set expected traffic level. I have identified traffic deviation trends and came up with ways how to use them when estimating the traffic levels. Estimating the traffic for ordinary work day and weekends was just a first step. Next problem that was successfully tackled were national holidays. Another traffic trend that I have identified and had to be addressed were days around national holidays (so-called lazy days). Even though I tried several ways how to correctly calculate the expected value for them, I did not discover any pattern that would be guaranteed to work. What also made it more difficult was the fact, that there is only very few lazy days in the historical dataset I had available. As the best solution, it was decided to let mediation engineers manually set expected lazy days in countries' configuration.

Thanks to the time shift feature and application deployed and permanently running at DigitalOcean, I and stakeholders were able to observe and evaluate the application as if it was running in production. At first, most of detected outages were caused by incorrect flow settings like too low granularity

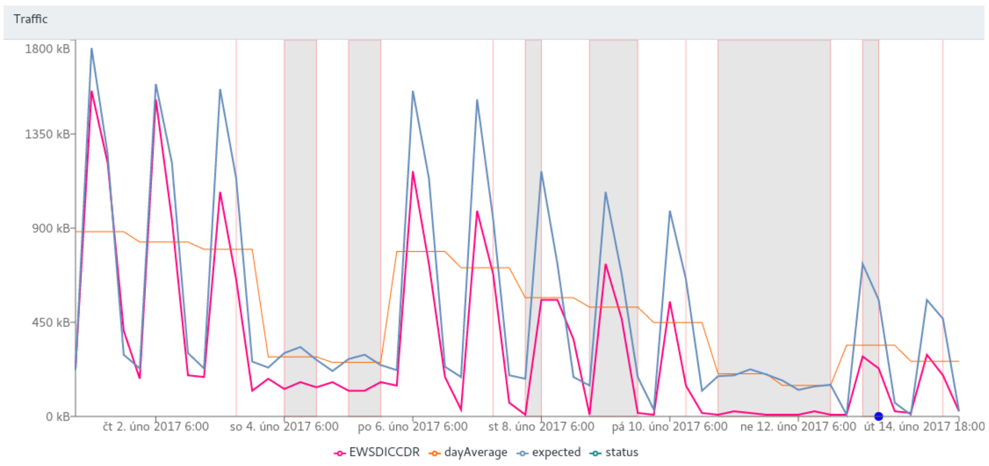


Figure 6.1: EWSDICCDR February decrease of traffic

or too high alarm levels. After a few days of tuning Czech flows' configurations, the application detected minimal or none false positives, but it was still detecting traffic anomalies reliably. For instance, on time shifted day of February 7th 2017, the monitoring daemon has detected outage on eight flows with five being a part of EWH LoB. After examining EWSDICCDR input flow I had found that first outage was already detected few days ago on 3rd of February at 12pm. It can be seen in figure 6.1 that the flow was slowly loosing traffic down to zero on 21st of February. The old monitoring system would detect first outage on 21st of February while this application triggered alarm 18 days earlier. This is obviously considered as a huge improvement and the new monitoring algorithm is superior to the old one. Other flows were part of M2M LoB and had no traffic at all so the alarm was also very well legitimate.

The feedback regarding overall application, such as user experience or other functionality, was very positive. I have received few requests for improvement of small UI related things like to use units with metric prefixes in chart (instead of 9000000 display 9MB) or put direct link from event log record to related flow. There are several unit tests for functionality where it was suitable and worth, including time operations on daylight saving time change and analysis of flow level with mocked traffic data.

F1 Visualization of flows Users are able to select desired date range and granularity and examine the traffic. It also applies outage detection algorithm on the selected date range, so that user knows how it would perform.

F2 Adding and removing flows There is a discovering code running every hour which adds new flows to the configuration and schedules them for analysis automatically.

F3 Countries support The application supports all four countries, but only flows of the Czech Republic were examined in more detail.

F4 Monitoring and anomaly detection The application is triggering alarms in situations and with limitations described earlier.

F5 Flows settings Settings of any flow can be changed by the user. Furthermore, the settings is inherited from LoB by input flow and forward flow, but it can be overridden.

F6 User login The application allows only authenticated users to change settings of the flows. Furthermore, there is a visitor login button for quick read only access.

F7 System integration The application implements consumer and producer for Apache Kafka, but its functionality could not be tested in production yet.

F8 Provide API API used by the frontend application is available for any other application if the correct API key is contained in the request, including traffic data endpoint `POST /mediation/data/query`.

NF1 Performance The analysis of flows is run in parallel on multiple threads and for all 4500 flows takes about five minutes on DigitalOcean server (2GB RAM, 2x virtual CPU@1.8 GHz). Request for data of a flow for 28 days and with sixty minutes granularity takes less than four seconds. Requests with smaller granularities should also use smaller time range.

NF2 Monitoring The application exposes monitoring endpoint providing information about seven components or services of the monitoring application. The frontend application also uses this endpoint.

NF3 Design Based on feedback from the stakeholders, the application's interface is easy to use and understandable. Their comments and requests for improvements have been incorporated.

6.1.1 Further development

The application implements all necessary functionality to serve its purpose, but there are still few future development ideas (or more like nice-to-haves) that were not implemented due to time and scope constraints.

It was already mentioned in frontend application's description, but it would be nice to cache data from API in Redux state instead of sending request every time a page is changed. Furthermore, push messages could be used to provide more elegant way of getting the newest data to the users, but that will also require implementation of messaging queue between monitoring daemon and API service.

In regard to monitoring itself, it would be indeed great to come up with a better way to the problem of lazy days. Next, the backend could also try to guess initial configuration for newly discovered flows, especially the granularity level.

6.2 ZooKeeper monitoring evaluation

ZooKeeper monitoring module was tested only locally since it is not meant to be deployed into provider's environment. Local ZooKeeper cluster is started by following steps described in 5.5. The outage of single node was simulated by killing single node's process and it was reflected in the frontend application within five to ten seconds, together with changing cluster's status to warning or outage depending on number of online nodes. The ZooKeeper monitoring is performing well and there are no changes or improvements that I would know of.

Chapter 7

Conclusion

This thesis has focused on monitoring of various systems and the main goal was to implement monitoring application for core system of telecommunication's provider.

In the beginning this thesis has described mediation system and its key role in provider's infrastructure. Next, several monitoring tools used by the provider were briefly introduced including an application for monitoring of mediation system and its flaws. After that, it has analyzed mediation system's flows, described their characteristics and traffic trends and specified general requirements for new monitoring application.

Later, this thesis has covered development process of a new mediation system's monitoring application. At first, various technologies for database, backend and frontend were considered and most suitable ones were selected. Based on flows' analysis, new outage detection algorithm was designed and implemented with chosen technologies, followed by careful architectural design of the whole application, which consists of total of four services. Interesting services and design and implementation issues were discussed together with application's user interface as well as its implementation.

I have been in close contact with the stakeholders to assure that the application meets their needs and that it can be deployed to production. At first, I had to understand the mediation system well so I could later propose solutions and ways how to solve particular monitoring problems. I have also collected requirements for the user interface and further improved it based on stakeholders' feedback. Unfortunately, due to internal corporate processes, the application did not receive permission for deployment to production yet due to various dependencies, which need to be approved first by security team. However, the monitoring functionality was simulated by using historical data and the results were discussed. It is estimated by the stakeholders that the application will receive a permission for deployment in the following months.

In parallel to mediation system monitoring, this thesis has analyzed several big data tools and for one chosen tool, ZooKeeper, a monitoring module has been implemented to the monitoring application and its functionality has been evaluated.

7.1 Personal evaluation

Thanks to this thesis I had a chance to experience an atmosphere of a large IT corporation and work on a project that is going to be used in a production, which made it more challenging but also exciting at the same time. It was also very interesting to learn about infrastructure of a telecommunication provider and the mediation system. Furthermore, I have learned several new technologies such as document databases, backend Python, React for frontend development and Apache Kafka and immediately got experience by using them on real project.

At the end I was a bit disappointed by the complex lengthy approval process of application dependencies, which delay the application deployment. On the other hand I understand it is necessary, since the provider is serving millions of customers and businesses and after all, I even admire the security level they have. I enjoyed working on this project and I am very grateful I was given this opportunity.

Bibliography

[1] Apache Software Foundation. Apache Camel. [Online, accessed: 2017-04-09]. Available from: `http://camel.apache.org/`

[2] GPP Organizational Partners (ARIB, ATIS, CCSA, ETSI, TTA, TTC). *Technical Specification Group Services and System Aspects; Telecommunication management; Charging management; Charging architecture and principles.* Reference 32.240, Release 11.

[3] Savitzky, A.; Golay, M. J. E. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Analytical Chemistry*, 1964.

[4] Riordon, J.; Zubritsky, E.; Newman, A. Top 10 Articles. *Analytical Chemistry*, volume 72, 2000.

[5] Schafer, R. What Is a Savitzky-Golay Filter? [Online, accessed: 2017-04-09]. Available from: `https://ai.berkeley.edu/~ee123/sp15/docs/SGFilter.pdf`

[6] IBM. IBM Tivoli. [Online, accessed: 2017-04-12]. Available from: `https://www.ibm.com/software/tivoli`

[7] Lohr, S. I.B.M. to Pay $743 Million For Developer Of Software. [Online, accessed: 2017-04-12]. Available from: `http://www.nytimes.com/1996/02/01/business/ibm-to-pay-743-million-for-developer-of-software.html`

[8] BMC Software Inc. Truesight. [Online, accessed: 2017-04-12]. Available from: `http://www.bmc.com/it-solutions/truesight.html`

[9] BMC Software Inc. TrueSight Intelligence. [Online, accessed: 2017-04-12]. Available from: `http://documents.bmc.com/products/documents/80/80/468080/468080.pdf`

[10] Forster, F. collectd. [Online, accessed: 2017-04-03]. Available from: `https://collectd.org/`

[11] Elastic Inc. Logstash, Elasticsearch, Kibana. [Online, accessed: 2017-04-03]. Available from: `https://www.elastic.co/`

[12] Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. [Online, accessed: 2017-04-09]. Available from: `https://research.google.com/archive/mapreduce.html`

[13] Apache Software Foundation. Apache Hadoop. [Online, accessed: 2017-04-09]. Available from: `http://hadoop.apache.org/`

[14] Apache Software Foundation. Apache Spark. [Online, accessed: 2017-04-09]. Available from: `http://spark.apache.org/`

[15] Karau, H.; Konwinski, A.; Wendell, P.; et al. *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015, ISBN 1449358624.

[16] Apache Software Foundation. Apache Zookeeper. [Online, accessed: 2017-04-15]. Available from: `http://zookeeper.apache.org/`

[17] Apache Software Foundation. Apache Curator. [Online, accessed: 2017-04-15]. Available from: `http://curator.apache.org/`

[18] Apache Software Foundation. Apache HBase. [Online, accessed: 2017-04-09]. Available from: `https://hbase.apache.org/`

[19] Codd, E.F.; IBM Research Lab, San Jose, CA. A relational model of data for large shared data banks. *Communications of the ACM*, volume 13, 1970.

[20] Oracle Corporation. MySQL 5.7 Reference Manual. [Online, accessed: 2017-04-18]. Available from: `https://dev.mysql.com/doc/refman/5.7/en/`

[21] Oracle Corporation. Oracle Database Online Documentation 11g. [Online, accessed: 2017-04-18]. Available from: `http://docs.oracle.com/cd/B28359_01/index.htm`

[22] MongoDB Inc. MongoDB. [Online, accessed: 2017-04-10]. Available from: `https://www.mongodb.com/`

[23] Couchbase Inc. N1QL Opens Couchbase Server to Massive SQL Ecosystem. [Online, accessed: 2017-04-10]. Available from: `https://www.couchbase.com/press-releases/n1ql-opens-couchbase-server-to-massive-sql-ecosystem`

[24] Apache Software Foundation. Apache Cassandra. [Online, accessed: 2017-04-09]. Available from: `http://cassandra.apache.org/`

[25] Hewitt, E. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. O'Reilly Media, 2011, ISBN 1491933666.

[26] Ronacher, A. Flask. [Online, accessed: 2017-04-05]. Available from: `http://flask.pocoo.org/`

[27] Facebook Inc. React. [Online, accessed: 2017-04-20]. Available from: `https://facebook.github.io/react/`

[28] Google Inc. Angular. [Online, accessed: 2017-04-20]. Available from: `https://angular.io/`

[29] Facebook Inc. The Diffing Algorithm. [Online, accessed: 2017-04-20]. Available from: `https://facebook.github.io/react/docs/reconciliation.html`

[30] Chedeau, C. React's diff algorithm. [Online, accessed: 2017-04-21]. Available from: `https://calendar.perfplanet.com/2013/diff/`

[31] Łukasz Holeczek. CoreUI. [Online, accessed: 2017-04-05]. Available from: `http://coreui.io/`

[32] McDonough, C. Supervisor. [Online, accessed: 2017-05-05]. Available from: `http://supervisord.org/`

[33] DigitalOcean Inc. DigitalOcean. [Online, accessed: 2017-05-07]. Available from: `https://www.digitalocean.com/`

Appendix A

Application Dependecies

The application requires following dependencies and is tested with stated versions.

- MongoDB 3.4.1
- Python 3.5.2
- pip3 9.0.0
- npm 4.0.5
- Node.js 4.2.6
- Supervisor 3.2.0
- Python 3.5 modules:
  - pymongo 3.4.0
  - flask 0.12
  - pytz 2016.7
  - schedule 0.4.2
  - simplejson 3.10.0
  - flask_cors 3.0.2
  - python-dateutil 2.6.0
  - kafka 1.3.3
- npm modules:
  - express 4.14.0
  - pushstate-server 3.0.0

Appendix B

Acronyms

CDR	Call Detail Record
SQL	Structured Query Language
MMS	Multimedia Messaging Service
SMS	Short Message Service
JSON	JavaScript Object Notation
API	Application Programing Interface
REST	Representational State Transfer
GSM	Global System for Mobile Communications
LoB	Line of Business
TSP	Telecommunications Service Provider
iBMD	international Billing Mediation Device
RDD	Resilient Distributed Dataset
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
DOM	Document Object Model

Appendix C

Selected configuration files

Listing C.1: Supervisord config file

```
[inet_http_server]
port=*:9001

[program:mongodb]
command=/root/app/db/bin/mongod --dbpath=/root/app/db/data --auth
stdout_logfile=%(ENV_APP_LOG_DIR)s/mongodb.log
redirect_stderr=true

[program:frontend]
command=pushstate-server %(ENV_APP_DIR)s/frontend/build/ 8080
stdout_logfile=%(ENV_APP_LOG_DIR)s/frontend.log
redirect_stderr=true

[program:api_service]
command=python3.5 %(ENV_APP_DIR)s/backend/api_service.py
stdout_logfile=%(ENV_APP_LOG_DIR)s/api_service.log
redirect_stderr=true

[program:daemon]
command=python3.5 -u %(ENV_APP_DIR)s/backend/daemon.py
stdout_logfile=%(ENV_APP_LOG_DIR)s/daemon.log
redirect_stderr=true
```

Listing C.2: Backend configuration file

```
{
  "flask": {
    "debug": false
  },
  "system": {
    "daysOffset": -90
  },
  "mediation": {
    "threadsCount": 8
  },
  "mongo": {
    "user": "user",
    "password": "password"
  },
  "integration": {
    "kafka": {
      "servers": "127.0.0.2:9092,192.168.1.5:9092,",
      "outputTopic": null,
      "inputTopic": null
    },
    "email": {
      "to": "tech-support@provider.com",
      "from": "mediation-monitoring@provider.com",
      "login": "mediation-monitoring@provider.com",
      "password": "password",
      "smtpHostname": "smtp.provider.com"
    }
  }
}
```

Listing C.3: Docker image configuration (Dockerfile)

```
FROM python:3.5-slim
RUN apt-get update
RUN apt-get install screen
RUN apt-get --assume-yes install python2.7
RUN apt-get --assume-yes install python-pip
RUN pip2 install supervisor
RUN apt-get --assume-yes install nodejs
RUN apt-get --assume-yes install npm
RUN npm install express
EXPOSE 27017 28017
VOLUME /data/db
ENV appDir /home/app
RUN mkdir ${appDir}
WORKDIR ${appDir}

ADD mongodb-linux-x86_64-debian81-3.4.1.tar mongodb
ADD defaultRequirements.txt .
RUN pip3 install -r ${appDir}/defaultRequirements.txt
ADD backend backend
RUN pip3 install -r ${appDir}/backend/requirements.txt
ADD frontend frontend
ADD supervisor.conf supervisor.conf
EXPOSE 5000 9000 9001
RUN mkdir -p log
CMD supervisord -c ${appDir}/supervisor.conf
```

Appendix D

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
src.........................the directory of application source codes
text
  thesis.pdf .........................the thesis text in PDF format
  src ................ the directory of LaTeX source codes of the thesis
```