

# Network and Systems Monitoring Application

João Sanches, *j.fial.sanches@tecnico.ulisboa.pt*

**Abstract**—This MSc Dissertation describes the implementation of a monitoring system aimed for universities, enterprises or on prem hosting services. We covered the most relevant time series databases and visualization tools available and in the end we used a Prometheus Time Series Database to collect the metrics from the equipments and to serve them to Grafana, a visualization tool that allows to build dashboards for a holistic observability.

Then we implemented a machine learning algorithm able to detect pattern anomalies in the collected data and trigger warnings automatically.

We evaluated the system for performance and scalability and concluded it is very lightweight in resource consumption which means it could be scaled to large networks. There are different architectural strategies that could be applied depending on the scenarios that allow that the system could be scaled even with the enlargement of the network.

**Index Terms**—Time series database; Observability; Metric Scrapping; Machine Learning

## I. INTRODUCTION

THE motivation for this project comes from the fact that companies nowadays are struggling with the maintenance of their services since IT infrastructures and getting larger and more load demanding with a fast and reliable accessibility and usability for the user being a concern. The agile methodologies implemented in the companies right now, like Scrum and Kanban, enforce quickly and dynamic deployments which sometimes can lead to small bugs that have implications on the system's reliability. So it is important to have monitoring (processes to collect system's metrics) and observability (process of analysing the collected information) methods in place in order to quickly notice errors that may be happening and act fast to fix them.

Having this in mind, we thought it would be interesting to investigate about network management and performance solutions that could be used to achieve that goal.

Our first goal was to establish a centralised monitoring system on three labs, sharing the same subnet, at Instituto Superior Técnico's Computer Department. We should be able to collect data about any of the computers connected inside each lab, as well as the switches that route traffic inside the network.

On the second hand we want to assess the state of the art on the available software to build such systems, specially open-source.

Finally, we would like to understand if it would be possible to implement machine learning on this system to automatically detect anomalies in the collected values.

## II. STATE OF THE ART

### A. SNMP

SNMP [1] stands for Simple Network Management Protocol and is an Internet standard protocol used to collect and mod-

ify management data from network devices such as routers, switches, modems, servers, hubs, workstations, even printers or cameras, and so on. SNMP works over UDP or TCP, but typically UDP is used since it is simpler and SNMP operations have retry mechanisms. In addition to complexity itself, if TCP is used, a manager device must have a dedicated TCP port and socket per agent which it is communicating with, and given that the port number only has 16 bits, the socket number would be limited to 65535 simultaneously open; with UDP a single manager port does the trick.

The SNMP protocol defines several PDU (Protocol Data Units) messages that allow for different operations, either to obtain data or modify it:

- **GetRequest**  
A manager-to-agent message that intends to fetch a specific information value (or list of values) from that device.
- **GetNextRequest**  
A manager-to-agent message that intends to obtain the next field variable value in the MIB data object (further explained ahead). Can be used to walk the MIB structure.
- **GetBulkRequest**  
Similar to GetNextRequest, but with the ability of obtaining a multiple values walking the MIB in a single request.
- **SetRequest**  
A manager-to-agent message that changes a value (or list of values) in the MIB of the managed device.
- **Trap**  
An agent-to-manager asynchronous message intended to send data to the manager without need of it being queried.
- **InformRequest**  
A manager-to-manager confirmation of an asynchronous message. Since UDP does not guarantee success in the delivery of a message, the InformRequest was introduced to notify the receipt of those messages.
- **ResponsePDU**  
An agent-to-manager message used to respond to GetRequest, GetNextRequest, GetBulkRequest, SetRequest and InformRequest message sent by the manager.

Most of the previously mentioned messages are like queries to a MIB (Management Information Base), which is an object database installed in the managed devices that SNMP uses to store and expose the data on those devices. The MIB is structured using SMI (Structure Management Interface), which defines the MIB's format (i.e. the data types that are used and the information transfer rules), resorting to ASN.1 macros. Each object has a OID (Object Identifier). The MIB is a hierarchical tree structure. That structure is somehow standardized, although some vendors may have differences

from each other. Nevertheless, some common MIB objects and their addresses include:

- **system** (1.3.6.1.2.1.1) – includes different objects with the device and its management information;
- **interface** (1.3.6.1.2.1.2) – interface information, for example in and out packets;
- **ip** (1.3.6.1.2.1.4) – IP routing details;
- **icmp** (1.3.6.1.2.1.5) – ICMP protocol details;
- **tcp** (1.3.6.1.2.1.6) – TCP protocol details;
- **udp** (1.3.6.1.2.1.5) – UDP protocol details;
- **egp** (1.3.6.1.2.1.8) – EGP protocol details;
- **snmp** (1.3.6.1.2.1.11) – performance information about its own SNMP;

Some of those fields may include objects inside of them, which you can leap into, like for example with *system*:

- .1.3.6.1.2.1.1.3.1 – **SysDescr** - a textual description of the entity;
- .1.3.6.1.2.1.1.3.2 – **SysObjectID** - vendor's authoritative identification of the network management subsystem contained in the entity;
- .1.3.6.1.2.1.1.3.3 – **UpTime** - time the entity has been running;
- .1.3.6.1.2.1.1.3.4 – **SysContact** - contact of the admin of the entity;
- .1.3.6.1.2.1.1.3.5 – **SysName** - assigned name of this entity;
- .1.3.6.1.2.1.1.3.6 – **SysLocation** - physical location of the entity;
- .1.3.6.1.2.1.1.3.7 – **SysServices** - a value that indicates the set of services the entity offers;

### B. Centralized vs Distributed NMS

In Martin-Flatin [2] divides Network Management Systems (NMS) in four different paradigms, that will be described in the following subsections: centralized, weakly distributed hierarchical, strongly distributed hierarchical and strongly distributed cooperative.

1) *Centralized Paradigm*: The centralized paradigm is characterized by a single manager node that congregates all the management-application processing and "dumb" agent nodes that answer to manager requests to obtain host data.

2) *Weakly Distributed Hierarchical Paradigm*: In a weakly distributed hierarchical paradigm scenario, management application is spread out across different nodes, with agents still answering to data queries by the manager, but in this paradigm, although there still is a top-level manager responsible for the entire system, there are other secondary managers that will help with processing. This increases scalability when compared with the centralized paradigm but still lacks robustness since if the main node fails the entire system stops working properly.

3) *Strongly Distributed Hierarchical Paradigm*: To address the issues with weakly distributed hierarchical paradigm, Goldszmidt [3] came up with an architecture named Management by Delegation which envisions that all agents should not only be answering to data queries but also become active in the management part. This means that a central manager

will still orchestrate the system but agent nodes will perform management tasks upon request.

4) *Strongly Distributed Cooperative Paradigm*: In the case of strongly distributed cooperative paradigm, nodes cooperate with one another in an intelligent social environment: authors say that these nodes should show four properties: autonomy, to act regardless of human interaction; social ability, to understand what the other nodes are doing and how they can contribute to achieve the common goals; reactivity, to understand the environment and act to occurrences; proactiveness, being able to achieve goals by itself. In sum, the main difference between strongly distributed hierarchical and cooperative paradigms is that the first opts for a more imperative fashion, while cooperative is closer to a declarative *modus operandi* at the cost of increased complexity and resources used.

### C. Available Solutions

1) *Cacti*: Cacti [4] is an open-source LAMP (Linux, Apache as server, MySQL as RDBM, and PHP, Pearl or Python as scripting language) network monitoring and graphing tool designed to provide a visual representation of various network-related metrics. It is particularly well-suited for monitoring and graphing the performance of network devices, servers, and other infrastructure components. Cacti mainly uses the Simple Network Management Protocol (SNMP) as its data collection method.

Cacti employs a polling mechanism to collect data at defined intervals, which is then stored in a database, typically using RRDtool (Round Robin Database tool). RRDtool is a powerful tool for efficiently storing time-series data and creating graphs. RRDtool implements a Round Robin Archive which is a fixed sized data structure, where, when full, the oldest data element of the structure will be deleted to make space for the newest sample. Old data but not old enough to be deleted is consolidated using average, maximum or minimum mathematical functions. The application of both these techniques allows having a data history while simultaneously guaranteeing that the size of the database does not increase indefinitely. A representation of this can be seen on figure 1.

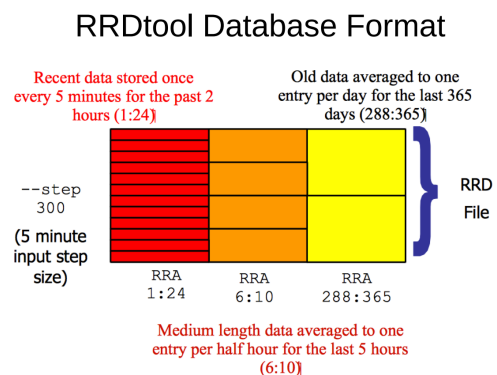


Fig. 1. Round Robin Database Tool Format

The application is handled through a web interface served by an Apache HTTP Server and that interface allows users

to configure, manage, and view graphs in a user friendly way given the access to various configuration options, templates, and graph management. It also supports setting up triggers and alerts.

Cacti benefits from a strong community of users and developers in the form of forums, documentation, and contributions that enhance the tool’s capabilities, not only because, since Cacti supports templates, its probable to find a template for the desired devices, but also because it is possible to install plug-ins on the application and there is a large community library.

2) *Prometheus*: Prometheus [5] is an open-source monitoring tool originated at SoundCloud, a music streaming service, in 2012, to address the specific monitoring and alerting needs of SoundCloud: they needed a modern monitoring system that could adapt to its microservices architecture, dynamic environment, and containerized infrastructure. The project was initiated by a team led by ex-Google engineers who were experienced in dealing with large-scale systems who drew inspiration from Google’s internal monitoring tools, especially Borgmon.

Prometheus was designed with several key principles in mind, including a dimensional data model with labels, a flexible query language (PromQL), a time series database, reliability, and scalability.

An overview of Prometheus architecture can be seen in figure 2.

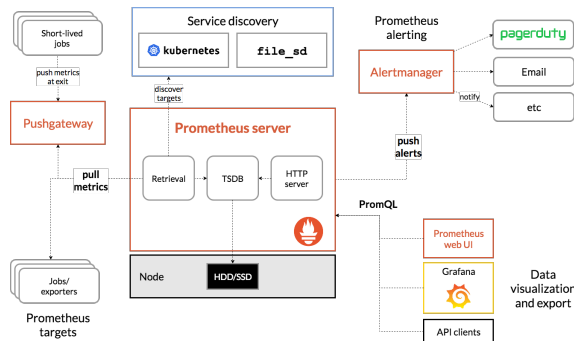


Fig. 2. Prometheus Architecture

The core component of Prometheus is the Prometheus server. It is mainly made of an HTTP server that allows Prometheus to expose its web-based user interface, metrics, and other functionalities, providing an interface for querying, managing, and monitoring Prometheus itself and responsible for data collection and storage. It also contains an embedded time-series database, often referred to as the TSDB (Time Series Database), that stores the collected metrics and compresses time-series data, allowing for efficient querying and analysis.

Prometheus supports service discovery mechanisms to dynamically discover and monitor targets. Common service discovery methods include static configuration, DNS-based discovery, file-based discovery, and integrations with orchestration platforms like Kubernetes.

The data is collected from targets which can be applications, services, or infrastructure components. The tool supports various types of exporters that act as agents to scrape and expose

metrics from different systems. Examples include the Node Exporter for system-level metrics and the Blackbox Exporter for probing endpoints. The Prometheus server periodically scrapes metrics from the configured targets, which expose a `/metrics` endpoint that provides metric data in a text-based format.

Metrics collected by Prometheus have a dimensional data model. Each metric is identified by a unique name and a set of key-value pairs called labels, which allow for a multi-dimensional representation of data.

The Pushgateway is a component that allows short-lived jobs or batch processes to expose their metrics to Prometheus. In the typical Prometheus pull model, the Prometheus server scrapes metrics from configured targets. However, in some scenarios where jobs are short-lived or not directly reachable by Prometheus, a different approach is needed. This is where the Pushgateway comes into play. Instead of pulling metrics from targets, jobs or processes push their metrics to the Pushgateway. Metrics pushed to the Pushgateway include labels for job and instance that help to identify and differentiate the metrics coming from different sources. Metrics pushed to the Pushgateway are associated with a specific expiration time, and after that time, they are considered stale and will not be scraped by Prometheus.

Prometheus provides a query language called PromQL for querying and analyzing metrics data. PromQL allows users to aggregate, filter, and transform time series data. It supports a wide range of functions for complex queries.

There are also alerting features. Users are allowed to define alerting rules, specifying conditions that, when met, trigger alerts, using its configuration file. The alerting system is built into Prometheus, and it can send alerts to external systems via the Alertmanager. The Alertmanager is a separate component that handles alerts sent by Prometheus and that can also perform additional actions, such as grouping, deduplicating, and routing alerts to the appropriate receivers: it supports various notification channels like email, Slack, and others.

Prometheus does not have built-in graphical user interface for creating graphs and visualization so it is often used in conjunction with other visualization tools with Grafana being the most popular open-source tool used. Grafana can connect to Prometheus as a data source, allowing users to create customized dashboards.

In terms of the network architecture, Prometheus supports federation, allowing multiple Prometheus servers to be linked together. This is useful for scalability and high availability. Federated Prometheus servers can scrape metrics from each other and aggregate data.

In matters of storage, there is also a feature that supports remote write and remote read capabilities, allowing the integration with remote storage systems, useful for long-term storage and analysis of metrics data.

3) *Grafana*: Grafana [6] is an open-source platform for monitoring and observability that allows you to query, visualize, alert on, and understand your metrics. It integrates with a variety of data sources, including time-series databases, logging systems, and other data platforms, making it a versatile tool for creating interactive and customizable dashboards.

Grafana supports a wide range of data sources, including Prometheus, InfluxDB, Graphite, Elasticsearch, MySQL, PostgreSQL, and many others. That is due to the support of different query languages depending on the data source (for example, PromQL for Prometheus but SQL in the case of relational databases, etc). This flexibility allows users to consolidate data from various sources into a single dashboard.

To connect Grafana to their desired data sources, users can use the Grafana’s web interface to configure the connections, where the connection details, authentication credentials, and other settings specific to each data source should be provided.

Grafana provides a wide range of visualization options, including line charts, bar graphs, tables, heatmaps, and more. Users can choose the visualization type that best represents their data. Customizable options include axis labels, legends, and color schemes. The graphs can be organized in dashboards to visualize and analyse data. Dashboards are composed of panels, each representing a different type of visualization or data source query and can be customizable to suit the monitoring needs. It is even possible to recur to templating for creating reusable and parametrized dashboards, even allowing users to create dynamic dashboards with variables. Annotations such as marking events, outages or any other relevant informations can be added on the dashboards and provide a way to add contextual information.

4) *Machine Learning*: Until this point we have only addressed data gathering, time series and data visualization. After all, it could also be interesting to explore automated ways to analyse or predict patterns in the collected data. With that in mind we carried out a research about tools that would typically be integrated with Prometheus for that purpose.

Facebook’s Prophet [7] is a tool implemented in R and Python for forecasting time-series data. It simplifies the process of time series forecasting by providing a user-friendly interface and powerful modelling capabilities. It runs on a model that automatically detects patterns and outliers in the data, allowing users to focus on interpreting the results rather than fine-tuning model parameters, even providing intuitive methods for visualizing forecasts and evaluating model performance, making it accessible to users with varying levels of expertise in time series analysis. Prophet has been gaining popularity among data scientists, analysts, and researchers seeking reliable forecasting solutions for a wide range of applications, from sales forecasting and demand planning to resource allocation and capacity planning.

Grafana Labs also developed a predictive time series model for Grafana called Grafana Machine Learning [8]. It is designed to help users extract valuable insights, identify patterns, and make data-driven decisions by leveraging machine learning algorithms. Grafana ML enables users to explore and analyze time series data to uncover trends, anomalies, and correlations. Users can leverage forecasting algorithms to predict future values based on historical data, enabling them to anticipate trends and plan accordingly and to automatically identify unusual patterns or outliers in time series data, helping in anomalies detection. Those insights and predictions can be directly visualized using Grafana’s dashboarding capabilities, allowing users to gain actionable insights at a glance. The

| ML Model | Normalized Mean Absolute Error (%) | Train Time (s) | Prediction Time (ms) |
|----------|------------------------------------|----------------|----------------------|
| Pmdarima | 6.54                               | 106.27         | <b>0.34</b>          |
| Prophet  | 5.31                               | <b>0.39</b>    | 88.48                |
| Ludwig   | 7.99                               | 6.26           | 61.80                |
| DeepAR   | 5.91                               | 174.14         | 7.98                 |
| TFT      | 5.99                               | 430.08         | 3.87                 |
| FEDOT    | <b>4.58</b>                        | 189.18         | 1.22                 |
| AutoTS   | 6.74                               | 11.13          | 7.07                 |
| Sktime   | 8.99                               | 1.19           | 3.09                 |

TABLE I

MACHINE LEARNING TOOLS COMPARISON AS DEPICTED ON [10]

main drawback is that this feature is only available for Pro users (paid subscription).

Weka (Waikato Environment for Knowledge Analysis) [9] is a collection of open-source machine learning algorithms and data analysis tools developed by the University of Waikato in New Zealand. It has tools for all kinds of data analytics purposes: data preparation, classification, regression, clustering, association rules mining, and visualization. Weka is implemented in Java which makes it a portable software capable of running on practically any platform.

Other Python tools could be considered such as the ones mentioned on the paper ”A Comparison of Automated Time Series Forecasting Tools for Smart Cities” [10] with them being, besides Prophet, Pmdarima, Ludwig, DeepAR, TFT, FEDOT, AutoTs and Sktime, which are all freely available Python libraries. A comparative board is shown on Table I. These tools were compared regarding predictive performances and computational efficiency. For the carried out tests, FEDOT outperformed the others providing the lowest average forecasting error (4.58%) and a reasonable prediction time of 1.22 milliseconds. It’s worse performance statistic is the training time of almost three minutes. Prophet also provides a low average forecasting error (5.31%) and the lowest average training time of 0.39 seconds but it is the worse option regarding forecasting, with an average prediction time of 88 milliseconds.

While comparing Prophet with Weka and FEDOT we come to the conclusion that Prophet is easier to use and offers a simpler interface for the given goal that we want to achieve; even though its prediction time is the worse when comparing with the other mentioned tools its 88 milliseconds prediction time mark is acceptable for the wanted task of sending alarms, where the training time being lower is more relevant.

### III. IMPLEMENTATION

#### A. Network Architecture

The network’s connection to the outside internet is done through a gateway built on a computer with ClearOS, which does address translation and applies firewall policies to the inbound and outbound traffic. The gateway is connected to one of the four existing switches; that switch is connected to the other three. From the total of four switches, three are managed and one is unmanaged. The importance of switches being managed or not is that they offer security and monitoring features such as SNMP while the unmanaged do not; the managed switches can be added to the pool of nodes to be

monitored. Laboratory LSD1 has one unmanaged TP-Link TL-SG1016 switch connecting the computers inside, laboratories LSD2 and LSD3 have managed D-Link switches model DGS-1210-24 and in the printer room there is a managed D-Link switch model DGS-1520-28. The entire topology is depicted in figure 3.

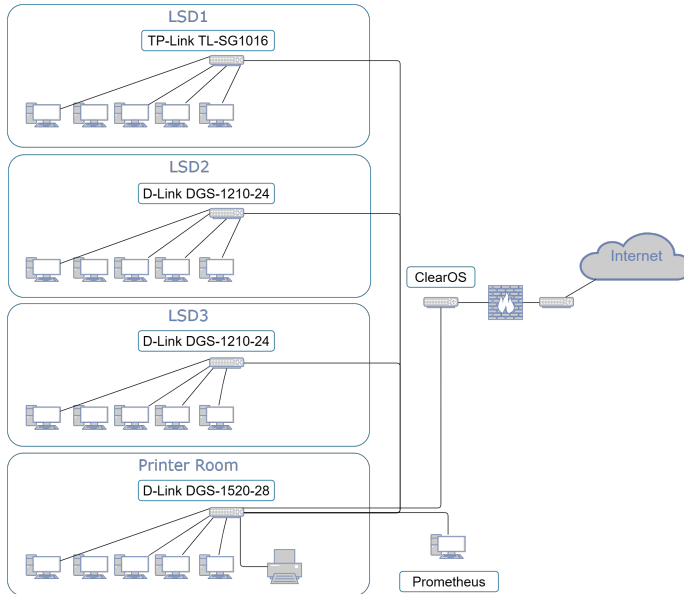


Fig. 3. Laboratories Network Topology

After analysing this network topology, we decided we would like to gather information regarding all devices on it, either computers, the managed switches and even the gateway. From the computers, including the gateway, we want to extract operative system's metrics regarding uptime, CPU, memory, storage, network traffic, etc., while from the switches we want to have data packets information, since the managed ones offer this feature through SNMP and RMON.

### B. Solution Architecture

To achieve that goal, the solution that seemed more suited, was to have a dedicated computer, connected to the same network as the labs, running Prometheus for data gathering and Grafana to display the gathered data. To extract OS information from the network nodes, we will be using the available Node Exporters from Prometheus GitHub page. They can be installed as a service either on Windows or Linux and what they do is translate the OS performance information to Prometheus data model and make it available through HTTP on a specific port. Then Prometheus agent will only need to scrape data from that port and store the data on its database. To cover SNMP devices, like switches, we will need to use another Prometheus component, which is the SNMP exporter: it acts as a proxy between Prometheus monitoring node and the SNMP monitored nodes, so it receives HTTP requests from Prometheus monitoring agent to scrape a given host and a given tree and translates the SNMP tree to Prometheus data model so that it can be promptly stored on the database.

The presented solution is depicted on Figure 4. We have a central node running both Prometheus and Grafana.

Prometheus' role is to extract data from the system and store it and for that it relies on a scrapping agent that is pointed towards the nodes in the network which sends periodic HTTP requests to collect the desired information. To access nodes' performance information the Node Exporters. A Node Exporter must be installed in every monitored host as a systemctl service. Note that the monitoring node is also running a Node Exporter: it is being used to self monitor the central machine. The SNMP Exporter receives, from the core agent, HTTP requests containing the address of the monitored device and the MIB tree objects that we want to query as parameters, performs the SNMP query to the device, translates the tree information and answers to the agent with information ready to store. Finally, Grafana is periodically consuming data from the Prometheus API to present it, updated, on the dashboards that were set up by the users.

After everything just described was settled, we decided we would add alerting and machine learning to the system. We would achieve those goals adding a Python script running the Prophet [11] library to make data forecasting and detect data points that are not within the predictions uncertainty intervals, and Prometheus Alertmanager, to have the system throwing alerts. The Python script could send alerts through the Alertmanager and the Prometheus instance itself could also be connected with it in order to send alerts when certain criteria is not met.

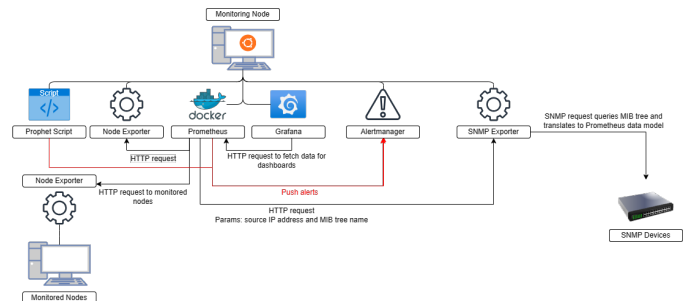


Fig. 4. Solution Architecture

If we wanted to further expand monitoring to a larger scale, we could do it using the federation feature from Prometheus. There are two ways to use Prometheus federation, one is in a hierarchical set-up and the other is cross-service federation. For our use case, we would like to apply hierarchical federation since our goals are scalability and data aggregation. The main idea behind hierarchical federation is to have a central general Prometheus server collecting aggregated metrics, lacking some detail, from child servers that would be collecting data directly from subset of devices or a cluster. This way we could have an aggregated view on the parent server and we could still drill down locally on the child servers, if needed.

### C. Prometheus Installation

As mentioned previously, Prometheus is a time series database open source application and it is going to be the core piece of the monitoring system. It will collect the monitoring data and store it as well.

After thinking on the best way to install the software we came across an exportability detail: what if we wanted to change the machine running Prometheus or update the software with a reduced downtime? To solve both those issues we decided to containerize it using Docker [12]. By doing so and using a volume, we can ensure that the gathered Prometheus data can be backed up and restored to a different host, and the container can also be installed in any machine containing Docker using a Dockerfile. In this case, we used the latest release from the Prometheus GitHub repository [13], corresponding to the version 2.48.1 on the day of the installation.

Optionally and for automation purposes it was decided to create a bash script to stop the current container, build a new image and start a new container. This is helpful to rerun Prometheus after configuration changes or when new monitoring nodes are added to the scraping agent. The script can be read on Listing ??.

#### D. Node Exporter Installation

To expose operative system metrics to be consumed by Prometheus it is necessary to install an agent called Node Exporter in each system (given that the machines to be monitored are dual boot, every machine must be installed with an agent on each subsystem). The Node Exporter is a component of Prometheus ecosystem designed to collect system-level metrics from target hosts: it runs on the target hosts, exposing various metrics related to CPU, memory, disk usage, network statistics, and more in a format that Prometheus can scrape and store. It interacts with the operating system of the target host to gather system-level metrics by leveraging various mechanisms, such as system calls and kernel interfaces provided by the operating system, to access performance and resource usage information.

#### E. Grafana Installation

Grafana was installed on Linux following the steps below:

- 1) Install the packages 'adduser', 'libfontconfig1' and 'musl'
 

```
sudo apt-get install -y adduser
libfontconfig1 musl
```
- 2) Download Grafana's install package
 

```
wget https://dl.grafana.com/
enterprise/release/
grafana-enterprise_10.3.3_amd64.deb
```
- 3) Install from the downloaded package
 

```
sudo dpkg -i
grafana-enterprise_10.3.3_amd64.deb
```

#### F. Grafana Dashboards Installation

To be able to visualize and interact with the data it is necessary to have dashboards to display the data in a way that facilitates reading. To do so we need to query the Prometheus database and then configure the graph to best suit the information that we want to present. To speed up

this process Grafana shares a community library with pre-made dashboards, which only need to be tweaked according to necessity or taste.

To install a dashboard in Grafana we just need to access <https://grafana.com/grafana/dashboards/> and pick the dashboard model that best suits our case.

Then, on our installed Grafana instance, we should go to the menu and click on "Dashboards". The window on Figure ?? shows the page that will open up. Then we click on the button "New" and choose "Import".

A new page will open up, as shown on Figure ?. On this page we will be able to import a dashboard from the community templates available on the website previously mentioned. All we have to do is copy and paste the ID that refers to the dashboard we want to import. After that it will automatically connect with the Prometheus instance (if it is already configured).

#### G. Alert Manager

Now that we have the data being displayed we would like to do something more automatic with it. In a monitoring system it is useful to generate alerts when certain conditions are met. Prometheus has another component called Alert Manager that allows to configure alerts and send them by email, for example.

The installation process starts by downloading the alertmanager from the GitHub page <sup>1</sup>. We should set up alertmanager to run as a service. After that we need to update the Prometheus configuration file for it to be integrated with the alertmanager.

#### H. Anomalies Detection with Prophet

After having the system fully operational we came up with an idea of exploring AI tools to detect anomalies in our collected data. After searching, we found one tool, Prophet, that would suit our needs and so we used it.

Prophet is an open source software released by Facebook's Code Data Science team. Quoting from the Github page [11] "Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. (...) Prophet is robust to missing data and shifts in the trend, and typically handles outliers well." It is available for download on CRAN and PyPI, meaning for R and Python languages respectively.

According to [7] this software was created to fill the need of producing high quality forecasts, since forecasting is a very specialized skill than even analysts may struggle with. The idea was to give analysts a tool that they could easily handle, where they could adapt parameters to best suit the forecasts but without having the necessity of knowing how to set up the computation of one.

For our specific case we are not so interested in doing forecasts but more on finding abnormalities in the data that is collected. Our use case will be exactly to trigger an alarm when we find out any data point that goes over the uncertainty interval that Prophet calculates.

<sup>1</sup><https://github.com/prometheus/alertmanager>

## IV. TESTS AND RESULTS

### A. Monitoring Solution

We will show the "Node Exporter Full" dashboard as a dashboard. The Figure 5 displays the data being gathered on the monitored computer nodes.

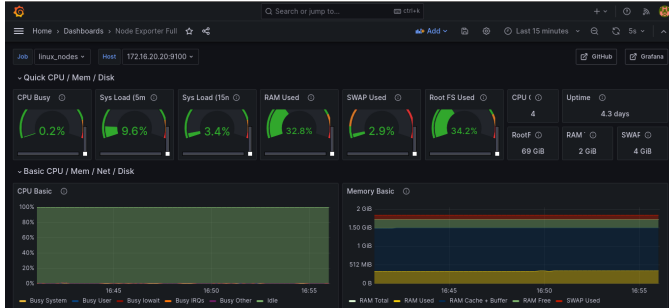


Fig. 5. Grafana Node Exporter Dashboard

On the top left corner we have two dropdown boxes: one for choosing the Prometheus job that is querying the host we want to analyse and the second box to choose the hostname (the options provided on the "host" box depend on what is chosen on the "job" box). On the top right corner we have two parameters that are general to all dashboards: time window, here defined to show the last fifteen minutes, and the refresh period, set up as five seconds. This means each five seconds the dashboard will update its values in order to show the most recent fifteen minutes. That parameter can be easily tweaked by clicking on that drop-down and choose any time frame (since Prometheus only holds data from the last 15 days, if you choose a previous point in time the graphs will become empty).

On that Figure 5 we can observe expanded tabs, for example where "Quick CPU/Mem/Disk" is being displayed. This tab contains graphs that give a quick overview about CPU, memory and disk. On it we have tiles or graphs displaying data gathered. This is the way that dashboards can be organized. We can see six tiles with meters showing CPU, memory and disk usage in current time. We then have other smaller five tiles showing "more permanent" metrics, like the number of cores, installed memory and SWAP and root filesystem storage free. Below we have other tab with information about the same topic but with greater detail, like seen on Figure 6. An example of the graphs contained on the tab "Memory Meminfo" can be seen on Figure 7, which provide more detailed information about memory.

The graphs are interactive and we can have specific information about a data point by hovering the mouse over the graph, like it can be seen on Figure 8. Just for clarification, the negative values observed on the "Network Traffic Basic" graph are meant to represent transmitted traffic, while the positive values are for the received. You can also observe on Figure 8 that it is possible to select a zone (click and drag) on a graph. This applies zoom to the time window, and the result can be seen on Figure ?? (even though on Figure 8 the zoom is done between 11h and 14h, the zoom was then enhanced on the peak). Every graph is affected. Note how the time on

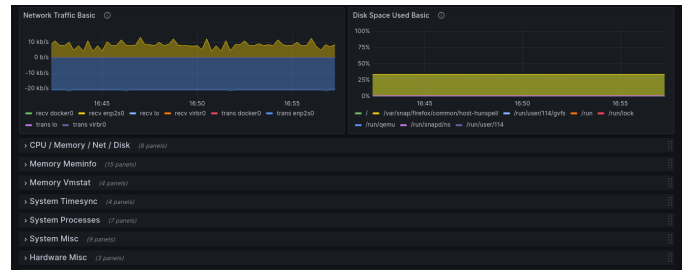


Fig. 6. Grafana Node Exporter Dashboard and other Categories

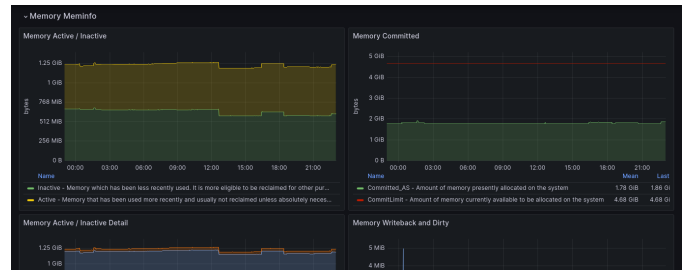


Fig. 7. Grafana Node Exporter Memory Meminfo

the x-axis has changed: Figure ??-a shows a 24-hour window, while Figure ??-b) shows a 11-minutes window.

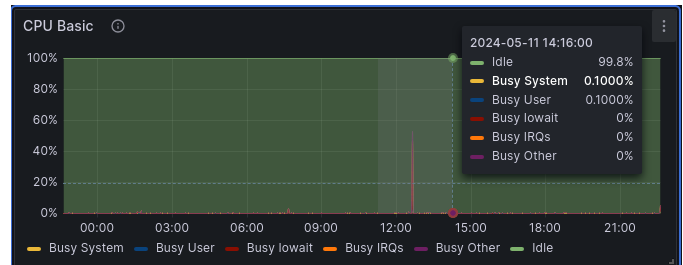


Fig. 8. Grafana Graph Tooltip

The rest of the tabs that are collapsed contain more information about each of the named topics and allow to quickly investigate about them in case it is needed.

### B. Anomalies Detection Solution

Now we will show how Prophet works. Recovering what was told in Section III-H, the anomalies detection starts with Prophet querying the Prometheus database. To get the best possible fitting and data prediction we query all the Prometheus data, correspondent to the last fifteen days. That data is then converted to a DataFrame (object type) and used as input to instantiate the Prophet model. After that we proceed to apply the fitting to that data. After that, we make Prophet calculate a prediction. Since we are not, in this case, looking for a future trend and just want to analyse points that may be giving any clue about any abnormal behaviour, the prediction will only return the original data plus an uncertainty interval where values should be contained. The Figure 9 shows the graphical representation of that result, including the change-points marked. In this specific case, the graph is showing the instant rate of increase of the total number of received bits

(5 seconds delta of received bits) on the Prometheus host machine, on its interface "enp1s0". The Prometheus instance is collecting values every 5 seconds, and we use every data point available. Notice as well that there is a blue margin around the dots marking the data points (dark blue) and the fitting line. This margin is the uncertainty corresponding to the aforementioned 95 percent confidence interval.

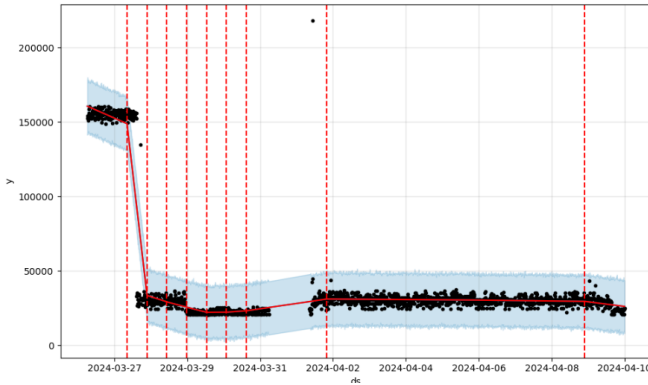


Fig. 9. Prophet Changepoints

After we obtain the "forecast" we compare these values, considering the uncertainty, with the values that were given to the model for its instantiation. Those points are shown in red on Figure 10.

Following, in case there are any data point over the limit (in the span of all the used data points), an alarm request is sent to the alertmanager and it redirects the alert to the proper recipients. The script is sending a POST HTTP request to the Alertmanager API /v2/alerts with a body containing the alert details. The alert will automatically be cleared if after five minutes there is no request refreshing the previous one. On Figure 11 it is possible to observe how the alert appears on the Alertmanager page. In this example, an email was sent.

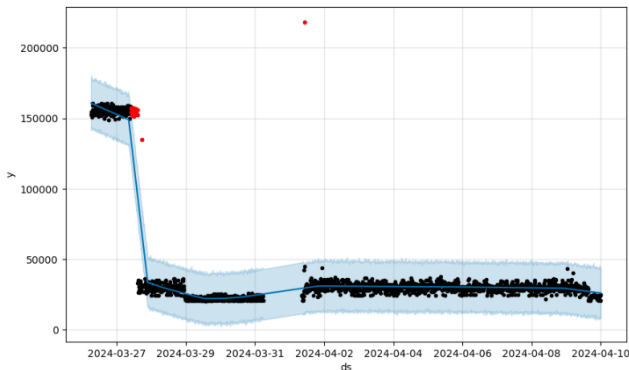


Fig. 10. Prophet Anomalies

### C. Resources consumption

We will study what is the impact of monitoring solution on the hosts and also on the network.

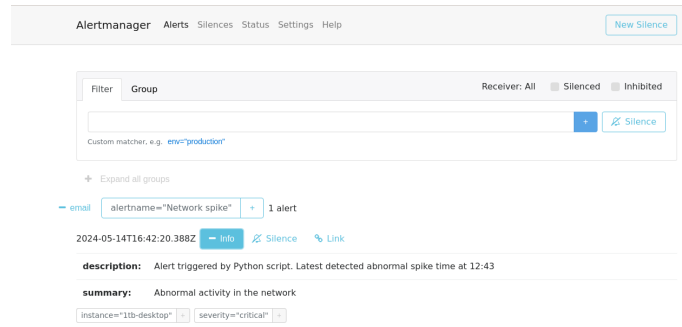


Fig. 11. Alertmanager Prophet Alert

1) *Prometheus Server Machine*: As we can recall from the solution architecture, the Prometheus server is running four different processes: Prometheus, Grafana, Node Exporter and SNMP Exporter. We need to study what is the impact of each one in the host machine.

The host is a computer operating GNU/Linux (Linux Mint 21), kernel version 5.15.0-48-generic, with an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz x86\_64 (four cores) and 8 gigabytes DDR4 memory.

The carried out Prometheus study was done for a configuration of nine scrapping targets: two Linux nodes (one of them is the local Node Exporter), two Windows nodes (one of them was shutdown during the metrics retrieval), four SNMP nodes (one of them is not yet physically installed, but it is configured as a scrapping target), and lastly is the Prometheus instance metrics endpoint. The scrapping interval time for each target is five seconds.

We made measurements to CPU, memory and disk usage during the period of one week. During this period, we made a configuration change, removing all the agent targets but one, for experimental purposes.

The gathered data shows that **Prometheus** uses approximately 1.5% of CPU divided by seven threads (three of the threads registered no CPU consumption) almost evenly. Regarding memory consumption it only uses around 200 MiB of resident memory and around 3000 MiB of virtual memory. This means that resident memory is stable and there is enough storage to accommodate the virtual memory. In matters of data operations, there is no significant load on average, having a steady volume of written data around 15 MB every ten minutes, with some casual spikes. Those spikes are related with data consolidation operations. It is also relevant to study the total volume of data stored in the Prometheus database. The database is being kept in a container volume. A cron job was programmed to track the size of the directory associated with the volume. On average, the database size keeps oscillating between 1.6GB and 1.2GB. Those oscillations are due to the fact that Prometheus compresses some of its old data and deletes data past the expiration period, since we are using the default retention time of fifteen days. Taking all of this into consideration we can estimate that this machine would be able to support more than 450 targets without the need of applying a different configuration like a federated and/or hierarchical solution.

**Grafana** is installed as a service, listening for clients to connect. This will be important to explain some values. The CPU consumption is stable around 0.2% when the web-app is not being accessed and higher, from 1% to 2%, when we were viewing the graphs and doing activity on the web-app. The memory usage is nearly constant, with 150 MiB of resident memory and 1600 MiB of virtual memory. Data write operations can be negligible with values around 0.02 MiB for a ten minute interval. Data reads are also generally low, with peaks around 130 MiB when opening up dashboards.

**Node Exporter** is a process that is constantly running on the background. It seems to be lightweight. The process's CPU consumption is steady and lower than 1%, approximately between 0.7% and 0.8%. It simply exposes metrics in a Prometheus friendly format, so it makes sense that the processing load does not change. The occupied virtual memory throughout the monitored period was always around 1200 MiB and the resident memory was approximately 20 MiB. The data operations were almost zero.

As explained previously **SNMP Exporter** acts a bit differently than the Node Exporter: it receives agent's requests to fetch information from a source, and it then sends SNMP requests to the destination and translates the response.

What we can observe about SNMP Exporter's CPU consumption is that it is almost constant around 1.3%. This is a bit higher when compared with the simple Node Exporter, but it makes sense since it does more work, and on the other hand, it is handling four requests each five seconds (the Node Exporter only handles one). Similarly, memory usage is also constant, with about 1200 MiB virtual memory being used and around 22 MiB of resident memory. Both data read and data write operations are practically zero.

2) *Monitoring Nodes*: The monitoring hosts should also be taken into account. Even though we have already studied what is the impact of the Node Exporter on the Prometheus server, it is important to confirm if its behaviour is the same when interacting with Prometheus remotely. It is also interesting to understand if there are performance differences on monitored nodes related to the operating system they are operating on, and if so, measure it.

Regarding the Node Exporter running on nscotia machine, which is a computer operating GNU/Linux (Linux Mint 21), kernel version 5.15.0-101-generic, with an Intel(R) Core(TM) i3 540 @ 3.07GHz x86\_64 (four cores) and two gigabytes of memory, the CPU usage is steady between 0.5% and 0.6% and the memory usage is also steady, with a consumption of around 1200 MiB virtual memory and about 15 MiB of resident memory.

## V. THESIS CONCLUSIONS

When we started this work we defined that the goal was to implement a monitoring system able to collect information from different networks and different devices, and have a centralised way of seeing the information holistically, which was achieved. A consequent goal was to study the state of the art regarding technology available to perform such tasks, and during this project we were able to study about different

technologies either related with monitoring and time series forecasting.

In the beginning we knew we would need some sort of scrapping agent to collect data from the monitored nodes. First we thought of using the software Cacti for that task, but it is heavily reliant on SNMP, which is not very flexible. We then shifted to Prometheus. Prometheus revealed to be a very flexible solution, since it has already great support for different protocols and data formats. Its community libraries offer lots of adaptation for more or less common monitoring specific cases, and its open-source nature even allows for modification if we need to modify any tool (either official or community made) to fit a more restrictive case. In what consumption of resources concern, our experiences show that Prometheus is very lightweight which opens room for easy scaling. Besides that, the federation feature even facilitates scaling in cases of bigger and more complex architectures. There are even tools like Thanos that are designed to help Prometheus scale, providing the system with larger capacity for historic data and centralized querying.

The next step was to build visualization. To do that we selected Grafana. Grafana is a standard tool to integrate with Prometheus nowadays and it is very straightforward to use. We imported two dashboards from Grafana Labs repository, one regarding Linux machines and the other regarding Windows machines. After installing they were ready to use.

The last task was to integrate a machine learning tool to compute the gathered data and discover anomalous data points. This was achieved using Prophet, a simple configurable model from Facebook that can predict time series trends with an uncertainty range. We used it within a Python script that, after looking for anomalies, triggers an alarm in case any is found.

This projects invites future work in subjects that were not tested with it. One of those would be to implement this monitoring system in a larger network, or even a geographically disperse network, and measure how that would impact resources consumption. Another interesting study would be to develop the anomaly detection process and even include some automatic actions that could fix the detected issues.

## REFERENCES

- [1] W. Hardaker, "Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP)," RFC 6353, Jul. 2011. [Online]. Available: <https://www.rfc-editor.org/info/rfc6353>
- [2] Jean-Philippe Martin-Flatin, "Web-Based Management of IP Network and Systems," PhD thesis no. 2256, Swiss Federal Institute of Technology Lausanne, 2000. [Online]. Available: <http://www.martin-flatin.org/papers/phd2000.pdf>
- [3] G. an Goldszmidt and A. S. Clark, "Load distribution for scalable web servers: Summer olympics 1996-a case study," in *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management, Sydney, Australia*. Citeseer, 1997.
- [4] Berry, Ian and Roman, Tony and Adams, Larry and Pasnak, J.P. and Conner, Jimmy and Scheck, Reinhard and Braun, Andreas, *The Cacti Manual*, The Cacti Group, 2017. [Online]. Available: <https://files.cacti.net/docs/pdf/manual.pdf>
- [5] B. Rabenstein and J. Volz, "Prometheus: A Next-Generation monitoring system (talk)." Dublin: USENIX Association, May 2015.
- [6] Grafana Labs. Grafana Documentation. Accessed on December 07, 2023. [Online]. Available: <https://grafana.com/docs/grafana/latest/>
- [7] S. J. Taylor and B. Letham, "Forecasting at scale," PeerJ Preprints, 2018.

- [8] Grafana Labs, “Grafana Machine Learning Documentation,” <https://grafana.com/docs/grafana-cloud/alerting-and-irm/machine-learning/>, Accessed: 23rd April 2024.
- [9] “Weka Git Repository,” <https://git.cms.waikato.ac.nz/weka/weka/-/tree/main>, accessed: May 2024.
- [10] P. J. Pereira, N. Costa, M. Barros, P. Cortez, D. Durães, A. Silva, and J. Machado, “A comparison of automated time series forecasting tools for smart cities,” in *Progress in Artificial Intelligence*, G. Marreiros, B. Martins, A. Paiva, B. Ribeiro, and A. Sardinha, Eds. Cham: Springer International Publishing, 2022, pp. 551–562.
- [11] Facebook, “Facebook Prophet,” <https://github.com/facebook/prophet/>, accessed: April 2024.
- [12] “Docker Documentation,” <https://docs.docker.com/>, accessed: 8th February 2024.
- [13] Prometheus Authors, “Prometheus GitHub Repository,” <https://github.com/prometheus/prometheus>, accessed: 8th February 2024.