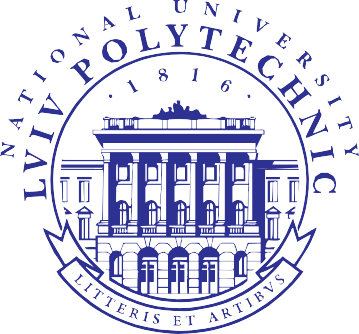
**MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE**

**Lviv Polytechnic National University Department of Computer-Aided Design Systems**



**ABSTRACT for 'BACHELOR'S THESIS**

"Monitoring and observability of distributed systems

in the context of preventive safety"

**Prepared by**

Student of group CS-41Z

*Dmytro Zhluktenko* **Scientific advisor** *Volodymyr Karkulyovskiy* **Consultant**

*Nazariy Andrushchak*

Lviv 2021

Introduction

Monitoring and observability for distributed systems include collecting, analyzing, and presenting various dynamic data regarding some particular processes running in the distributed system. In a modern distributed systems environment, many works and practices are available for making these non-trivial tasks more manageable, even though monitoring remains one of the most complicated aspects to tackle while modeling production-ready software. The work exposes reasons behind such wide adoption of distributed systems for modern industry workloads, focusing on such practices as DevOps and SRE, making it so accessible for a broad audience.

The work also defines characteristics of distributed systems that make monitoring so complicated. These characteristics are used for understanding the methodology of monitoring and comparing related papers.

The main goal of the work is to examine the importance of monitoring and observability in the modern distributed system, taking into account the domain where applied. Also, to investigate collecting, analyzing, and presenting operational data in distributed systems for preventing critical situations, i.e., incidents. Besides technical challenges, the works also evaluate the monitoring from a product ownership point of view, describing the approaches to SLAs, monitoring culture mindset, incidents, et cetera. Also, incidents are discussed in work, focusing on recovering and learning from them, most importantly preventing them in the future. Then, the price of monitoring is discussed at various stages of the software product lifecycle, especially in production environments.

Also, the software product is built based on this study which shows the usage of modern monitoring best practices. The aspects are covered both conceptually and practically: metrics, alerts, logging, dashboards, messages, debugging, tracing. The software product is built for the reader's better understanding of the described concepts. Technologies used to create this software include but are not limited to .NET Core, Prometheus, Docker, Docker-Compose, Kibana, Logstash, Zipkin, OpenTelemetry.

The information given in this thesis is based on the student's experience regarding building and architecting distributed systems and running and supporting such systems in production environments with a strong focus on reliability, stability, and safety.

The work is organized in the following way: Chapter 1 describes related works and the motivation behind the research. Chapter 2 defines the domain of the work, going through distributed systems and their monitoring, which evolved over time. Next, in chapter 3, we focus on implementing monitoring by discussing conceptual aspects and illustrating them with practical examples. Visualization of monitoring is presented in chapter 4. Finally, the work talks on preventive safety in distributed systems which could be achieved in multiple ways discussed in the chapter. Conclusions and future work are presented in chapter 5.

Related works

The field of distributed systems and monitoring is studied ever since parallel computations took place [1]. The terminology of monitoring and observability takes roots in system control theory and measures the level with regards to which the distributed system could be observed. In other words, how the internal state of the system could be identified based on its outputs. The outputs mainly were the results of testing, either manual or automated.

A vast amount of classical research focused on creating own monitoring systems, which were supposed to ensure stability and reliability of a distributed system [2]. These researches inspired us start working on the topic of the proposed research as they affected how monitoring was built up back in the day when none of the modern approaches and frameworks existed.

Besides standard approaches to monitoring, it's critically essential to predict the performance of the system.

Johng et al. [3], doing the research, suggested two approaches to monitoring problems - benchmarking and simulation. These approaches turned out feasible, rational, fast, cheap, and reliable. Following the same route, Lin et al. [4] suggested a new way of exploring and identifying root causes for incidents in microservice architectures based on visualization of the data available (i.e., logs, metrics, traces).

Talking about modern environments such as the cloud, the observability of the system is defined the same way as back in the way, but the outputs of the system are different. The outputs are logs, metrics, traces, et cetera. They are generally used for defining observability and, therefore, for identifying the system's state.

There are also modern works that discuss approaches for monitoring distributed systems [5]. Based on that [5], Mostafa and Bonakdarpour [6] were able to adapt these approaches to anomaly detection based on some properties. Therefore, anomalies could be recognized as incorrect behavior of the system. Some researchers disclose the aspects of monitoring highly applicable to today's state of technology out of the modern academic world. Moreover, they are already used by modern systems developers. A great example of such research is "On Observability and Monitoring of Distributed Systems – An Industry Interview Study" by Sina Niedermaier, Falko Koetter, Andreas Freymann, Stefan Wagner [7]. That research was based on 28 interviews with monitoring and DevOps practitioners. In the end, it managed to identify key pain points for interviewees and define possible future works for the industry, i.e., what could be the progress in monitoring distributed systems.

The preventive aspect of monitoring distributed systems was not addressed among top-notch researchers. Thus this work was created. It's supposed to fill that gap by carefully analyzing related works and discussing how monitoring could be preventive in the context of distributed systems that tend to fail.

Distributed systems as domain field

The field of study is definitely distributed systems. Before we dive into approaches, terminology, and instruments, it's crucial to outline the idea of modern distributed systems, taking into account how they evolved with time since the term 'distributed system' is available in research since the early 80s. [8]. Nevertheless, we could claim that the definition of distributed systems from the 1980s is still applicable to modern systems.

The distributed system contains multiple processes which work on different servers and perform the shared work. That shared work is distributed among servers for more optimal scenario execution. These processes must communicate with each other for better integration via any underlying communication protocol (HTTP, AMQP, gRPC, etc. [9-11].

Systems were distributed by design a long time ago because multithreaded application also counts as a distributed system even though the work is distributed between threads and cores, not physical servers.

Over the last decade (2010-2020), distributed systems became a trend for developing high-load and data-intense systems. Also, a modern distributed systems are the most distributed the world has ever seen since they span multiple servers, cloud providers, countries, companies. They execute various programs based on different platforms and runtimes (such as Go, .NET, Java), but they still are feasible to be run. One would question why distributed systems are so relevant at all? Distributed systems have a list of advantages over traditional centralized systems:

* lowered cost of infrastructure
* improved reliability and availability
* ease of extracting modules and composing them
* flexibility of configuration
* ability to grow step by step while enhancing the system by replacing individual components

Even if each individual system is trivial and straightforward, integrating these components could create drastically colossal complexity. It creates multiple problems that did not exist in traditional systems that worked within one physical machine. In the end, the number of components in play increases statistical equity of error.

These problems include but are not limited to:

* Multiple components can unexpectedly fail, possibly simultaneously, which leads to even more unexpected consequences due to other parts of the system react to invalid input from failed subsystems.
* Communication between subsystems can fail, including transient network failures. In such environments, network quality and stability are paramount.
* If the network fails, the subsystems tend to retry requests/messages, leading to the system's fatally corrupted state.
* The heterogeneity of the systems could be different on so many levels - the code that executes the logic, infrastructure technologies such as Docker images, virtual machines, or even serverless functions. Moreover, the teams could also be heterogeneous. While such a diverse environment could cultivate innovation, allowing multiple teams to pick the most appropriate technologies for their subsystems, at the same time, it creates additional complexity for a consistent approach for monitoring such systems.

Monitoring distributed systems

Looking after trends and observing distributed computations and systems is a fundamental problem since the early ages of distributed systems [1]. In 1987 Joyce [12] identified five aspects representing the complexity of monitoring distributed systems. This could be considered as an early attempt to identify critical characteristics of problems that differentiate monitoring centralized systems from monitoring distributed systems.

Here are these aspects:

* Distributed systems have much more points of touch where it's possible to control them. In some situations, that much more so, it seems unrealistic to cover them.
* The presence of non-deterministic delays in communication between subsystems makes it impossible to make deterministic claims about the state of the system at a given moment.
* Inherited non-determinism from distributed and asynchronous systems.
* Monitoring distributed systems change their behavior similar to the observer effect [13].
* The complexity of interaction between system and systems developers.

Considering the challenge of availability of distributed systems and their popularity, we must mention all instruments which became available with the technical progress. One of the fundamental instruments is Docker, which allows isolating the software product in the environment, which, in its turn, ensures determinism while deploying Docker image (as programming artifact) into another environment.

Similarly, we must mention the ever-growing popularity of cloud solutions, which mainly making a start easier for software products. By itself, cloud solutions are distributed systems in essence since solutions are created by combining multiple components. If before early 2010s cloud solutions were too inappropriate because of the pricing, now they are so cheap, so it's feasible to use cloud instead of maintaining own data centers. Nevertheless, the growing popularity of cloud solutions, big data, orchestrators made us reconsider modeling monitoring for distributed systems because distributed systems themselves changed significantly. The main change was to split the system into much smaller, more concrete subsystems, which obviously lead to increasing the number of components involved. Such change has a direct impact on what to monitor because of the increased number of components. In the end, these components create incredibly vast amounts of data as a resulting monitoring artifact to be investigated.

Discussing the need for monitoring, it's important to mention that both system developers and managers want to have as much as possible data for decision-making and control over the distributed system for making the most out of it. So, such systems should be observed at each stage of their lifecycle - from prototyping to everyday support of production systems. As the monitoring changes, the monitoring changes as well by introducing new instruments, approaches, et cetera. The goals remain the same - to understand where the issues are, their root causes, timely alert the developers, identify the incident, et cetera.

Monitoring and its components

There are multiple problems monitoring could address, which are described in the full version of the work. Going further, we will describe three main stages of which monitoring consists - data collection, processing, and presentation.

**Data collection**: Relevant data from the system must be composed into an understandable format for the monitoring system. For instance, Prometheus collects data from subsystem and represents them via API endpoint in the following format:

|  |
| --- |
| # HELP process\_private\_memory\_bytes Process private memory size |
| # TYPE process\_private\_memory\_bytes gauge |
| process\_private\_memory\_bytes 74604544 |
| # HELP process\_virtual\_memory\_bytes Virtual memory size in bytes. |
| # TYPE process\_virtual\_memory\_bytes gauge |
| process\_virtual\_memory\_bytes 2223070081024 |
| # HELP process\_start\_time\_seconds Start time of the process since unix epoch in seconds. |
| # TYPE process\_start\_time\_seconds gauge |
| process\_start\_time\_seconds 1576244939.1073897 |
| # HELP dotnet\_total\_memory\_bytes Total known allocated memory |
| # TYPE dotnet\_total\_memory\_bytes gauge |
| dotnet\_total\_memory\_bytes 3013928 |
| # HELP process\_cpu\_seconds\_total Total user and system CPU time spent in seconds. |
| # TYPE process\_cpu\_seconds\_total counter |
| process\_cpu\_seconds\_total 0.796875 |
| # HELP dotnet\_collection\_count\_total GC collection count |
| # TYPE dotnet\_collection\_count\_total counter |
| dotnet\_collection\_count\_total{generation="1"} 0 |
| dotnet\_collection\_count\_total{generation="2"} 0 |
| dotnet\_collection\_count\_total{generation="0"} 0 |
| # HELP process\_working\_set\_bytes Process working set |
| # TYPE process\_working\_set\_bytes gauge |
| process\_working\_set\_bytes 56242176 |
| # HELP process\_num\_threads Total number of threads |
| # TYPE process\_num\_threads gauge |
| process\_num\_threads 35 |
| # HELP process\_open\_handles Number of open handles |
| # TYPE process\_open\_handles gauge |
| process\_open\_handles 566 |

*Fig. 1. Example of Prometheus data format*

That data already corresponds to the format, thus could be collected by Prometheus scrapper. Also, the vital thing to mention that data collection should not take up too much of the system resources.

**Processing:** There could be more data than enough in some cases, but that data does not hold much informational value. For instance, there is no need to collect the data each second from some subsystems regarding their CPU load. In such cases, adaptive sampling could be applied. This would reduce the amount of data that is being stored, but, at the same time, it will not be critical for understanding the state of the system. In other words, some data could be neglected by the monitoring system since that does not bring much new information for the users; thus, it's not optimal to have them consume resources. Such change could affect how many subsystems could be monitored and served by the monitoring system.

Also, data should be stored in such data storage, which is optimal for such workloads - vast amounts of data where more data is coming in each second, but much less being read. Nevertheless, data must be optimized to be easily presentable, taking into account some filtering, ordering, aggregation, et cetera. It means that it's essential to index data and maintain corresponding indexes per each update of data storage. As an example of such storage, ElasticSearch is used as an industry standard for storing monitoring data.

Graphical user interface, text, application

Description automatically generated

*Fig. 2. Example of data stored in ElasticSearch.*

**Presentation:** It's a stage where data is getting to the user of the monitoring system. Collected and processed data is displayed to users in the corresponding relevant form. Considering the amount of data that distributed system outputs for monitoring purposes, the monitoring system user should be able to keep up with such cognitive load. That's why the data must be optimized so the user would only explore what's relevant to him. Filtering, ordering, aggregation is used for such purpose. Filtering data is the process of reducing data so that the monitoring system user could receive only data relevant for his level of perception (e.g., get information on what was happening at concrete server for the last 15 minutes). The critical thing to mention is that data stays in the data storage, but the user interface only displays data relevant to the user. Moreover, there are other instruments of visualization which could reduce cognitive load. With the correct approach to visualization, the user can explore the data for an extended period of time, which is adapted, so it's more accessible to percept it.

Already knowing monitoring stages, we can also present the three pillars of monitoring [14] - logs, metrics, and traces. They are deeply described in complete work.

Implementing monitoring

Each of the components mentioned above - logs, metrics, and traces are essential for the distributed system to become observable.

We have created an application that illustrates practical examples of monitoring usage in distributed systems. The application was developed using such technologies as .NET Core, Docker, Docker-Compose, Prometheus, Kibana, ElasticSearch, Logstash. Each of these components is responsible for delivering a particular piece of monitoring. .NET Core application is a host for our samples. By having corresponding program code, we made it emit and expose metrics via API endpoint to be accessible by external systems such as Prometheus scrapper.

A picture containing graphical user interface

Description automatically generated

*Fig. 3. Program listing to emit and expose Prometheus metrics.*

Docker-compose was used to aggregate all the required monitoring components, so it's all deployed at once running a simple command:

**docker-compose up -d**

Prometheus is an open-source monitoring solution widely used for data collection, analysis, and presenting the data related to metrics. Initially, that solution was developed by SoundCloud and then made available as an open-source solution for mass usage via Apache Licence 2.0.

For deploying Prometheus, we used Docker-Compose as the optimal solution for quick environment-agnostic deployment. There are configuration files for Prometheus. They are done in YAML. These configuration files affect the deployment process and the behavior of the system in the environment.

All metrics are aggregated, and the software product exposes them via an API endpoint called "/metrics". The response that the API endpoint sends is similar to what is exposed in Table 1.

Visualizing the results

Prometheus displays the system's productivity in the form of a graph that allows users to quickly understand the system's state at a glance. Graphs enable easy comparison of system performance from a time-series point of view. For more interactive visualization Prometheus Expression Browser is used. It allows us to display a vast amount of metrics collected over time. That part of the system is available via "<http://localhost:9000/new/graph>". With the help of Prometheus Expression Browser, we can explore data, picking the most appropriate metrics to be seen on figures 4-6.

Graphical user interface

Description automatically generated with low confidence

*Fig. 4. Example of Prometheus usage.*

A picture containing chart

Description automatically generated

*Fig. 5. Example of Prometheus usage.*

Chart, histogram

Description automatically generated

*Fig. 6. Example of visualized metrics emitted from our application*

As we have explored the figures 4-6, we can conclude that the software product is not stable since there are apparent falls in the chart, which means the program's shutdown. Also, the important thing to mention is that graph is showing the linear growth of the selected metric - "prom\_ok".

Preventive safety via monitoring in distributed systems

The work describes possible adverse outcomes for the software product and defines what the problems to be prevented by the monitoring are:

* user dissatisfaction with the product
* harm to users or their property (e.g., data)
* loss of trust
* direct or indirect income loss
* negative pressure on brand name
* undesired mentions in media
* suboptimal usage of computing resources, therefore suboptimal financial strategy

Most of these points are highly related to business outcomes since the business is affected by anything happening with a distributed system. Also, some of these points are hardly measurable. So, everyone expects that system will be stable with no losses, no problems, and profit to the company and added value to its users.

The work already discussed some approaches to monitoring in previous chapters. Nevertheless, they all cover only the identification and investigation of problems once the incident has happened or is happening at the moment. Instead, this part of the research focuses on incidents before they happen. It focuses on preventing them from happening. Correspondingly, some approaches to preventive safety are similar to approaches for solving incidents in production systems. Monitoring allows seeing the negative trends in systems performance which, in its turn, will enable them to take appropriate actions for reacting upon these trends. In the context of preventive safety, there is no immediate need to respond to these trends when identifying them. Instead, it makes more sense to analyze them and then prioritize only after discovering their potential damage. Also, special attention should be paid to warnings. The codebase should be enriched with more logging, additional metrics that follow the same goal, defining the granularity of system processes. In other words, it's important to understand situations where it's not an error yet, but at the same time, it's not a happy-path scenario, subject to worry about. Based on monitoring data presented in dashboards and trends, system developers and product owners could make thought-out decisions. There is always a specter of available choices to take. Samples are illustrated in the Table 2.

**Table 2. Feedback cycle loop example**

|  |  |
| --- | --- |
| Data analysis | Decision & Action |
| CPU slowly goes up during last month | Developers decide to scale up, so the service has more CPU available. It reduces the risk of incident for a short period of time |
| Service saturation goes down | Developers decide to scale down by having a weaker server for this service. It optimized resource usage and financial strategy |
| Database is heavily loaded, 2% requests fail | Developers change the architecture by adding caching in front of the database. |

Also, the work discovers preventive safety for software products from a manager's point of view. Key points there:

* Managers are interested in the system being stable so that it can serve users
* There is SLA that systems have to honor as it sets expectations. It's essential to set realistic expectations in this case
* Different domains require different SLAs
* Managers want to make a data-driven decision on the stability
* Non-functional requirements are as necessary as functional for high-load systems
* The main goal for the manager is to balance the feature delivery and investment into system reliability

Also, the work discusses the importance of creating a monitoring culture and monitoring mindset, which predominantly affects the software product being built.

Additionally, postmortem culture is discussed in work as an instrument of preventive safety. While referring to postmortems, we also discuss Corrective and Predictive Actions (CAPA) as a concept from which postmortems are inherited. So, a postmortem document is a document regarding the incident that happened which answers the following questions:

* What happened?
* How system and developers reacted?
* What could we do differently next time?
* What could be done to prevent this from happening?

The work focuses on the last two questions since they are critical for understanding preventing the danger by avoiding it or fighting it more effectively. The work also stresses sharing such postmortems so everyone could get a clear understanding of what has happened. That way, knowledge flows across the organization and prevents the happening of new incidents. In the end, postmortems are the potential to strengthen the distributed system and system developers.

Conclusions

Monitoring is an essential part of modern distributed systems for visualizing, testing, debugging, and development. With no monitoring and observability, modern distributed systems have minimum chances to succeed and to be stable. Considering the performed research in this thesis, it was identified that monitoring and observability of distributed systems are not that much of a technical issue anymore. Instead, it becomes a cross-concern topic that goes through the whole software lifecycle and, therefore, is critical for product success.

The work described distributed systems as a field of study and monitoring to understand the domain better. Also, the work exposed the problem of monitoring by telling the three pillars of monitoring - logs, traces, and metrics. These pillars were presented both conceptually and with examples provided by software product built for illustrativeness. Primarily the work is dedicated to preventive safety as a goal and monitoring as an instrument for achieving that goal. Thinking through monitoring, we also covered managerial point of view on that topic. Finally, we have covered postmortems as an approach to preventing incidents as well. Future works are presented in the following chapter.

Future work

We suggest discovering the usage of Artificial Intelligence for anomaly detection since that's the most potential area and is still not studied well. Such an application might automate many processes related to defining the stable level of the system's health. For instance, AI could automatically determine predicates to be used in alerts for distributed systems. Similarly, it could be used to identify anomalies that are not involved in the alerts system but are relevant for discovering the state of the system and might affect the system's stability. Once the nature of these anomalies is well understood, it's possible to prevent the incidents. Another significant usage of AI in monitoring systems could be the automatic reduction of data presented to the user for investigation, reducing the cognitive load. There is some data to be neglected. Also, the most exciting area of applying AI and machine learning is predicting the root cause of incidents based on available data.

We did not study deeply how monitoring affects the performance of the distributed system, so this could also be another topic for research. This is important since the technologies for monitoring change so fast, so there is the risk of reducing the system's performance upon which monitoring is applied.

In this work, we focused on modern approaches to monitoring preventive safety in distributed systems. The modern world is evolving extremely rapidly, especially in IT. Thus it might be a great idea to create research on changes that happen in preventive safety via monitoring in distributed systems.

References

1. Robert Cooper and Keith Marzullo. 1991. Consistent detection of global predicates. SIGPLAN Not. 26, 12 (Dec. 1991), 167–174. DOI:<https://doi.org/10.1145/127695.122774>
2. Mansouri-Samani, Masoud & Sloman, Morris. (1993). Monitoring distributed systems. Network, IEEE. 7. 20 - 30. 10.1109/65.244791.
3. Johng, H., Kim, D., Hill, T., Chung, L.: Estimating the performance of cloud-based systems using benchmarking and simulation in a complementary manner. In: Intl Conference on Service-Oriented Computing. pp. 576–591. Springer (2018)
4. Lin, J., Chen, P., Zheng, Z.: Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Service-Oriented Computing. pp. 3–20. Springer International Publishing, Cham (2018)
5. Pinal V Chauhan, 2012, Cloud Computing In Distributed System, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 01, Issue 10 (December 2012),
6. Mostafa, Menna & Bonakdarpour, Borzoo. (2015). Decentralized Runtime Verification of LTL Specifications in Distributed Systems. 494-503. 10.1109/IPDPS.2015.95.
7. Niedermaier, S., Koetter, F., Freymann, A., & Wagner, S. (2019). On Observability and Monitoring of Distributed Systems – An Industry Interview Study. Lecture Notes in Computer Science, 36–52.
8. Manzo, M. & Frisiani, Arrigo & Vernazza, T.. (1982). A monitoring distributed system. Microprocessing and Microprogramming. 10. 19–24. 10.1016/0165-6074(82)90118-1.
9. George Coulouris. Distributed Systems: Concepts and Design. Addison-Wesley, 2011
10. Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In Proc. of CCS'11, pages 151–162, 2011
11. 11Hagit Attiya and Jennifer L. Welch. Distributed computing: fundamentals, simulations and advanced topics. Wiley, 2004
12. Joyce, Jeffrey & Lomow, Greg & Slind, Konrad & Unger, Brian. (1987). Monitoring Distributed Systems.. ACM Trans. Comput. Syst.. 5. 121-150. 10.1145/13677.22723.
13. <https://en.wikipedia.org/wiki/Observer_effect_(physics)>
14. Distributed Systems Observability by Cindy Sridharan, [https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html#:~:text=Logs%2C%20metrics%2C%20and%20traces%20are,the%20three%20pillars%20of%20observability](https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html#:%7E:text=Logs%2C%20metrics%2C%20and%20traces%20are,the%20three%20pillars%20of%20observability)