Multipurpose remote monitoring system based on microservice architecture

Luka Bjelica, Miloš Panić, Marko Pejić

***Abstract*—This paper presents a cloud-native, multi-purpose, and reusable system for collecting, processing and storing data, with the aim of monitoring an arbitrary physical system. The proposed system can be divided into three main parts: a private network containing a set of microservices that perform complete data processing, applications that implement the low-level logic for collecting data from remote sensors, and a web client which enables interaction between the user and the rest of the system. The final product of this paper is a system based on the microservice architecture named *isobar.ot*, that allows monitoring of the chosen set of values of an arbitrary physical system, through a simple and functional user interface. Using the system presented in this paper, the user is able to control the entire course of remote monitoring: from the selection and specification of the collected data scheme, through the definition of alarm values, to displaying changes of values and alarms in real-time.**

Keywords: distributed systems, microservice architecture, remote monitoring systems, cloud-native systems, Internet of Things

I. Introduction

Monitoring the various parameters of arbitrary physical systems is a crucial part of every industrial facility. Supervisory Control and Data Acquisition (SCADA) systems are ubiquitous in almost all industries: from the food industry to the power industry, which results in a need for continuous improvements of the existing, and development of new solutions [1].

This paper concerns the development of the modern solution for remote monitoring systems that can be used for monitoring an arbitrary physical system. It is based on a microservice architecture with cutting-edge tools and technologies. Motivation for choosing this topic came from a necessity for a system that can work with large amounts of data and is flexible in a relation to a supervised physical system, which makes it usable as a part of Internet of Things (IoT) systems [2].

One of the biggest challenges when designing such a system is the scalability, i.e., the ability of the system to work with a large number of sensors and serve a large number of clients without a drop in performance. Furthermore, such a system requires a simple and functional user interface in order to provide an operator with an efficient way to monitor changes in the collected data, have insight into the alarming events in real-time, as well as defining new locations, alarm types, schemes of the data that is collected, etc. For the above-mentioned goals to be fulfilled, the proposed system is designed according to the principles of microservice architecture.

After the Introduction, basic principles, advantages and disadvantages of the microservice architecture and the architecture of the proposed solution are explained in Section II. Technical details about the implementation, along with the tools and technologies that were used are introduced in Section III. Results and user interface are shown in Section IV and the concluding remarks along with future plans are given in the final Section V.

L. Bjelica (bjelicaluka@uns.ac.rs), M. Panić (panic.sw19.2018@uns.ac.rs), M. Pejić (markopejic@uns.ac.rs), University of Novi Sad, Faculty of Technical Sciences, Department of Computing and Control Engineering, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia.

II. Architecture

Two mandatory requirements that the proposed system must meet are working with a large amount of data and serving a large number of clients. The architecture of the proposed system is designed so that the mentioned requirements are satisfied for the arbitrary amount of data and number of clients.

Microservice Architecture

Microservice architecture implies the development of applications in the form of small, isolated, and independent services that communicate with each other via clearly defined protocols. Such a method for developing systems came about due to the aim of overcoming flaws and problems that come with monolith architecture.

The traditional approach to developing software implies the use of monolithic architecture. One of the flaws of monolithic architecture is that it poorly copes with overload [3]. One approach to handling overload is to vertically scale the existing machine on which the application runs. That is expensive and not efficient enough to solve the problem completely. The other approach is to use horizontal scaling and create multiple instances of the application. This approach leads to inefficient use of hardware resources because there is no possibility to scale only those parts of the system that require it [3]. In addition, the degree of reusability of individual components is reduced because they are tightly coupled with the system they were initially developed for.

The main advantage of the microservice architecture-based systems is that the individual microservices can be scaled independently. That way, better utilization of hardware resources is achieved so that only the parts of the system that are affected by overload get scaled. Additional advantages that the microservice architecture brings are independent development of individual components as well as the high degree of their reusability [3].

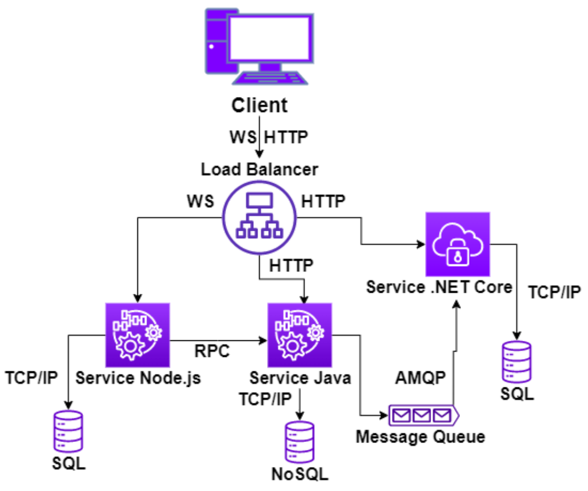


Fig. 1. Microservice Architecture Scheme

Fig. 1 presents the example of the microservice architecture scheme. It can be noticed that system based on the microservice architecture is suitable for scaling individual parts that require it because each part represents an independent application. There is also the possibility of using different technologies for implementing individual services, as well as the use of different communication mechanisms and protocols depending on the need.

System development follows the *divide and conquer* principle, which divides a large and complex system into smaller units, which are easier to develop. It is important to mention that dividing the system into smaller entities does not solve the complexity problem, but rather it delegates it to a level above, that is, to connect the system's components and their orchestration.

Previously stated benefits surpass hardware limitations of a single machine on which the system is running and thus make the microservice architecture an adequate solution for implementing remote monitoring systems.

System Architecture

The proposed system represents a set of components, each being an independent application with a unique role. The system is made out of services that are responsible for: authentication, user groups and user profiles, schemes of data that is collected from arbitrary physical sensors, validation, persistence and aggregation of the collected data, detecting alarms, generating reports from the aggregated data, and displaying the user interface.

The architecture of the proposed system is presented in Fig. 2. The system runs on the cloud and is completely independent of the physical system from which it receives the data.

The responsibility for collecting data from physical sensors and sending it to the system is encapsulated within the Local Processing Unit (LPU). LPU acts as an intermediary

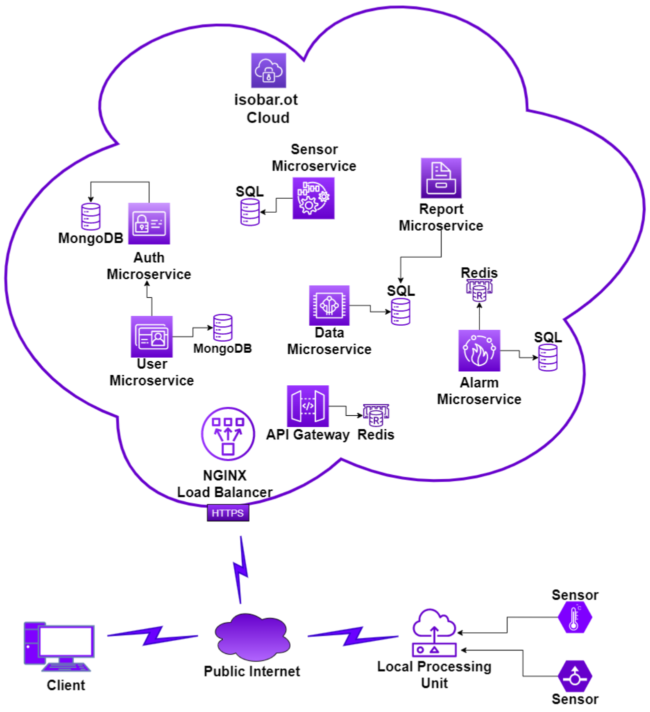


Fig. 2. System Architecture Scheme

between physical sensors which collect raw data, and the part of the system which is launched on a cloud.

The services are a part of the private, isolated network and it's not possible to address them directly from the public internet. The ingress service, acting both as a reverse proxy and load balancer [4], is the only way to access services from the outside. All inter-service communication is done in a synchronous manner with one exception. The only asynchronous communication is data ingress where the API Gateway service sends the data to the Data and Alarm services separately and is not interested in the content of the response.

A brief description of individual services that make up the system follows:

1) Authentication Service: The role of the authentication service is to provide a safe way to persist user credentials. This service implements the logic for generating access tokens that are used by other services for restricting unauthorized access to individual resources. It's worth pointing out that the authorization is highly dependent on the context it's used in and implies that details of role-based access control (RBAC) used for restricting access to individual resources need to be defined in the service that owns the resources. Otherwise, every service that implements RBAC would be tightly coupled to the service that implements authorization logic. The problem with that approach is that the authorization service may represent a bottleneck of the system. Tight coupling of services is not a problem if they are running inside the same process. Knowing that the presented system is distributed, this approach presents a big problem because it increases the degree of inter-service communication and causes the known problem "chatting" [5]. The problem mentioned above is the reason why the authorization is not implemented within the authentication service.

2) User Service: The service responsible for working with user profiles and user groups is a very important component of the system because it enables isolating data at the level of user groups. This service is responsible for protecting user's personal information and controlling which user group does the user belong to. Considering that most of the resources on the system are tied to a specific user group, user has access limited only to those resources tied to the group it belongs to.

3) Sensor Abstractions Service: A key component of the proposed system is a service that provides a way for arbitrary sensors that collect data on a supervised physical system. The role of this service is to persist information about the schemes of data that is being collected, as well as information about the concrete sensors that send that data. The entities used by this service are sensor abstractions, i.e., schemes of data that is being collected, and the information about the concrete physical sensors along with their locations. The idea behind defining data schemes is the possibility of validation of incoming data on the service, as well as the possibility of using the same data scheme for multiple different sensors. The scheme represents a set of individual tags (physical values of interest), each containing a name and a primitive data type. Apart from the name and the simple type, for every tag in the data type, aggregation methods are listed, based on which, aggregation service knows how to process raw data.

4) Data Validation Service: Received data first goes through a validation process, which is carried out based on the previously defined data type. This service is also responsible for verifying the validity of the public API token, using which the LPU unit proves authenticity. Apart from validation, this component presents a suitable place for dispatching events about received data in real-time. Events are dispatched through previously defined bidirectional communication protocol with the aim of achieving *publisher-subscriber* mechanism [6].

5) Data Persistence and Aggregation Service: The component which contains markedly the most complex logic and which requires the most hardware resources is data persistence and aggregation service. Before it gets aggregated, the raw data are persisted inside a temporary data store that is being cleared after a fixed period of time. The reason why raw data aren't stored permanently is that the amount of data is immensely large and that storing it isn't efficient. The more efficient solution is doing periodical data aggregation, such that users are able to define a time period after which the aggregation is performed. Additionally, users are able to define the methods by which data aggregation is performed, which later allows them to follow trends and generate reports of interest. By that, the system gets better performances, not only in terms of memory usage but also in decreasing the time needed for generating certain reports.

6) Report Service: The purpose of the persisted data lies in the ability to generate certain reports from them, with the aim of monitoring trends and presenting behavior of arbitrary values that are collected. This service implements the logic for generating reports on the aggregated data that is permanently persisted in the system. The report takes into account the specific frequency at which the data was aggregated as well as the time interval within which the data was collected. It also provides the ability to define and store report types that contain all the information needed to generate a particular report, except for the time interval.

7) Alarm Service: Detecting critical values, that is, data values which deviate from predefined boundaries can be very significant for physical systems which the proposed system is monitoring. The responsibility of this component is the detection of critical values and dispatching events about them, in real-time. Critical values are detected by rules previously defined in alarm types. Alarm type contains priority, a threshold value, and the information about whether the threshold presents an upper or lower limit of the normal state. A property from the data type can have a set of predefined alarm types tied to it. During alarm detection, every alarm type that is tied to a certain property is taken into consideration. When the critical value is detected, an event is dispatched through a predefined, real-time communication protocol. After the alarm event is dispatched, the client has the option of caching that alarm for a certain time period and thus preventing the system from dispatching more of the same events tied to the alarm of a certain priority, type, and limit value.

8) User Interface Service: This service provides elements needed for the graphical presentation of real-time data and generated reports. In addition to that, it contains elements that can be used to create certain resources, set certain rules, and take care of users and user groups.

III. Implementation

The microservice architecture allows the usage of numerous technologies for implementing individual components so that the most suitable technology for the requirements specific to that component is used. Fig. 3 shows an overview of all technologies used for implementing certain parts of the system.

Implementation of Individual Components

The authentication service implementation was realized using the .NET Core [7], while the MongoDB [8] database was used for the persistence of user accounts. Each user account consists of a unique name, password, and role. In case of the data leak, *hashing and salting* [9] of passwords is applied with the aim of preventing their misuse.

For the purpose of implementation of the service for working with user groups and profiles, .NET Core was used. User groups and profiles are in a one-to-many relationship, i.e., a profile belongs to exactly one group, while a group can contain several user profiles. The user group contains a name, surname, and e-mail address. To ensure that the

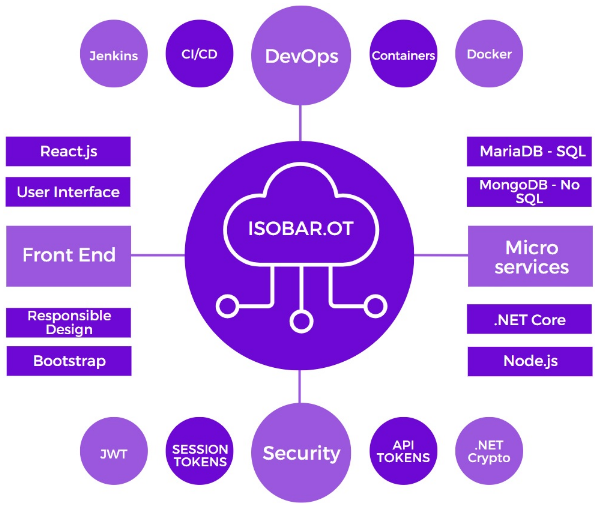


Fig. 3. Overview Of The Used Technologies

connection between user groups and accounts is modeled properly, MariaDB [10] relational database was used.

Another of the services implemented using .NET Core is a service for defining abstractions of physical sensors, i.e., schemes of data that come from remote sensors. Entities relevant to this microservice are sensor schemes, physical sensors, and locations. Since these entities are interrelated, a relational database MariaDB was used for data persistence. Primitive data types which are supported are real numbers, boolean values, enumerations, and text. Each sensor has its own public API token, which is a randomly created string of letters and digits that can be altered. Altering the public API token of a sensor is a significant function since it provides a mechanism of protection from receiving false data from a party that has obtained the token in an unauthorized way.

The implementation of the data validation logic is separated into the data validation service implemented in the Node.js [11] environment. This service uses Redis [12] cache as temporary storage for a public API key as well as the data scheme of the authenticated sensor. This decreases communication with the service in charge of validating public API tokens and evades creating the system's bottleneck. Sending data in real-time is done by using a WebSocket, using the socket.io library [13].

Data persistence and aggregation service is implemented using .NET Core and MariaDB database. One of the main reasons for choosing MariaDB as a relational database is the native support for built-in mechanisms that can be used for storing and manipulating data in dynamic JSON [14] format. The collected data is stored in a temporary table whose content is deleted after an all-day cycle of aggregations. The system supports several different aggregation time periods, of which the smallest is five minutes, and several different methods for aggregating real numbers including minimum, maximum, sum and mean. The aggregated data is stored in separate tables in the database, each corresponding to a single resolution.

.NET Core was used for the implementation of the report service. This service reads data from the database in which the persistence and data aggregation service has stored the processed data. Report generation was realized with the help of mechanisms for manipulating data in JSON format that are supported by the MariaDB database.

Implementation of alarm service is done inside the Node.js environment, while the MariaDB database was used for storing information about the alarm schemes and concrete critical values themselves. Sending data in real-time is done using WebSocket, which makes users promptly informed about every critical value of the monitored system. This service also uses Redis cache for storing critical values to avoid notifying the client unnecessarily. Another role of Redis is to synchronize socket.io events between multiple instances of the application.

The user interface was implemented using React.js [15] and Bootstrap [16] libraries.

Automation of Development Processes

Developing a system that is based on a microservice architecture increases the maintenance complexity because the source code of the system is made up of multiple smaller and often independently maintained code bases. Continuous Integration (CI) represents a necessary part of the development process of systems composed of many components with independently maintained code bases. That includes validation and testing of individual functionalities, as well as the rebuilding of components affected by changes. Continuous Deployment (CD) is the process of automatic reflection of changes to the final system which is used in production. In systems that are subject to frequent changes, the CD represents a necessity and can be very important in both the development and deployment phases of the system. In the development process, an instance of the staging application is created in order to provide access to the application to everyone that is involved. That significantly increases the degree of error detection in the development phase and reduces the chances of a bug in production.

Developing the system comprised of components that are implemented in different technologies, complicates the requirements for the environment in which individual components can be started. The concept of containers is introduced with the aim of providing a virtual environment at the operating system level, which can be predefined, packaged, and quickly launched. Such an environment has a high degree of portability and can be run on any platform on which a container engine can be run. The proposed system uses Docker [17] for the containerization of individual components. Docker provides an API for defining, packaging, transferring, and running virtual environments in form of containers. It also supports the creation of isolated private networks within which containers can intercommunicate and reference eachother using the local DNS server [18].

High availability and fault tolerance deserve special attention for distributed systems that are running in a production environment. A highly available system tends to minimize service downtime while a fault-tolerant one ensures that no data is lost during minor or major failures. The goal is to have both a highly available and fault-tolerant system that ensures high service uptime without data loss. The fact that the proposed system can be deployed on a multi-node cluster and that the data replication on multiple nodes is supported ensures that the aforementioned goals are met in a production environment. By distributing traffic across multiple nodes, the uptime of the individual services, resilience to data loss, and the request processing capacity are increased.

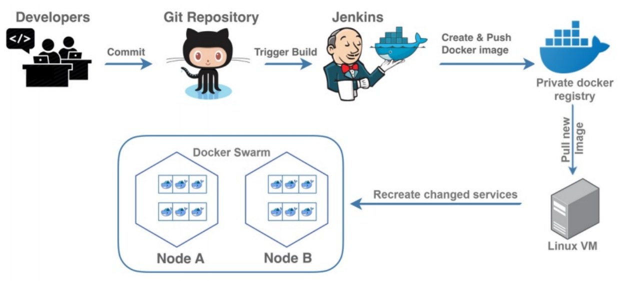


Fig. 4. Automatic Development Process Scheme

Fig. 4 illustrates the scheme of the automated CI/CD pipeline used by the proposed system in a development environment. Once the changes are made, they are synchronized with the remote codebase located on GitHub. The application that implements the CI/CD pipeline, i.e. Jenkins [19] gets notified, via the WebHook submission mechanism, about the changes that have been made on a certain branch. When an event containing information about the changes reaches Jenkins, a predefined pipeline is started. The pipeline runs the process of validating changes and rebuilding Docker Images if the changes prove to be valid. Additionally, the affected parts of the system get synchronized with newly integrated changes by a command that gets executed using SSH (Secure Shell) on the machine on which the system is running.

The proposed system can be instantiated using the *docker-compose* tool, for the needs of the development environment or using the *kubectl* tool for production environments. The role of the aforementioned tools is to take care of pulling, configuring, starting, and shutting down previously defined services, creating private networks within which the services communicate and scaling certain services depending on the needs of the system.

IV. Results

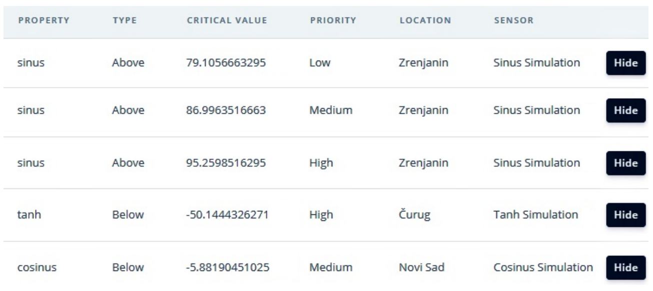
The final product of this paper is a functional remote monitoring system, based on microservice architecture and modern technologies, called *isobar.ot*. The system described in this paper supports the processing and persistence of arbitrary data sets, as well as tracking of trends in the collected data, i.e., periodical changes in the data values and the detection of critical values in real-time. In addition, the system allows the generation of reports from the persisted data for a certain period of time, which allows the user to analyze the behavior of the physical system that is monitored.

The most valuable aspects of the proposed system are the fact that it can be reused for many different physical systems, and also its scalability which is ensured by the microservice architecture. The system solves one of the basic problems that come with the IoT systems, which is giving semantics to the raw data collected from the physical sensors so that storing and processing of data is realized in a uniform way, independent of the nature of data.

Besides various simulations, such as collecting data on weather conditions, that were used to test the system's reusability, there are two successful use-cases of the proposed system. Both of them are Road Traffic Monitoring Systems (RTMS) that are deployed in India and Croatia. The first receives data from two different sensor types: laser and radar. The laser sensor sends detailed information about the passed vehicles such as speed, class, and it's dimensions. The radar detects vehicle's speed and relative position and send them in real-time.The second use-case receives data from a camera that detects which class of vehicle has passed. The previously mentioned use-cases prove that the proposed solution can be used to monitor arbitrary physical systems. In both cases, the system ensures high availability and fault tolerance, i.e. it is deployed on a multi-node cluster and uses data replication in order to prevent data loss.

User Interface

Fig. 5 shows the appearance of the user interface, which monitors critical values in real-time. The hide option gives users an opportunity to declare that they are aware of a particular alarm, to take all necessary steps, and not want the system to notify them more about the alarm so that they can pay attention to other alarms.



PROPERTY	TYPE	CRITICAL VALUE	PRIORITY	LOCATION	SENSOR	
sinus	Above	79.1056663295	Low	Zrenjanin	Sinus Simulation	Hide
sinus	Above	86.9963516663	Medium	Zrenjanin	Sinus Simulation	Hide
sinus	Above	95.2598516295	High	Zrenjanin	Sinus Simulation	Hide
tanh	Below	-50.1444326271	High	Čurug	Tanh Simulation	Hide
cosinus	Below	-5.88190451025	Medium	Novi Sad	Cosinus Simulation	Hide

Fig. 5. Alarms Monitoring in Real-Time

The *Dashboard* section (Fig. 6) presents the appearance of the user interface through which data arrived from sensors is tracked. Selecting the sensors is done from the drop-down menu, where all the sensors belonging to the corresponding user groups are listed. It is possible to choose more than one sensor and to display data in real-time graphically and in a table.

For the purpose of testing the proposed system, simulation sensors are implemented, which represent individual applications independent of the rest of the system. Their role is to simulate the operation of LPU units by sending random values or values of certain mathematical functions, instead of collecting data from physical sensors. Some of the functions supported by simulation sensors are sine, cosine, sigmoid, ReLU (Rectified Linear Unit), and the like.

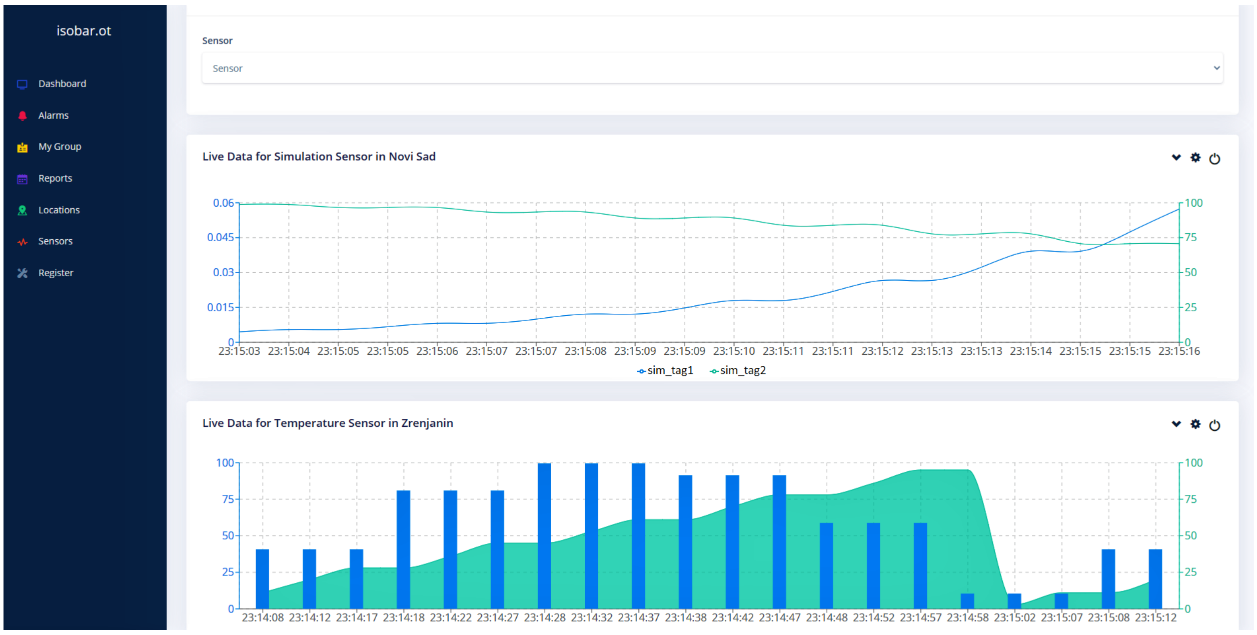


Fig. 6. Live Data Display

V. Conclusion

This paper presents a cloud-native, multi-purpose remote monitoring system, based on microservice architecture. The implemented system is named *isobar.ot* and consists of three main parts: local processing units (LPU) used for implementing low-level logic for collecting data from physical sensors and sending it to the cloud, an isolated private network with a set of microservices that perform the entire processing and persisting data on the cloud, a client web application that allows users to interact with the rest of the system.

The biggest advantage of the proposed system lies in its ability to monitor arbitrary physical systems, and the ability to work with a large number of sensors and serve a large number of clients. These advantages are achieved by relying on microservice architecture and modern technologies.

Despite the fact that inter-service communication is brought to a minimum, there are cases where a better solution for communication would be to use asynchronous protocols, such as AMQP [20]. Another disadvantage of the proposed system is that some services use data that are not owned by them, i.e., they have to turn to services that have that data. This reduces the failure resistance of interdependent parts of the system. A potential solution to this problem is to use caching more frequently or replicate the data used by multiple services while maintaining asynchronous consistency.

In addition to overcoming the previously mentioned shortcomings of the proposed system, the plan for further development is the implementation of a data export service in the form of a file of a certain format, such as PDF, CSV, JSON, and the like. Also, the plan is to implement support for receiving data in several different protocols, and not only in HTTP. Another possibility for further development of the proposed system is the implementation of a uniform control component. The task of this component is to provide a mechanism for managing the monitored physical system, at a high level. That way, the proposed system would be extended to a fully functional supervisory and control system, which is certainly in the plan for future development.

References

[1] A. Goel and R. Mishra, “Remote data acquisition using wireless - scada system,” *International Journal of Engineering*, vol. 3, 03 2009.
[2] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey,” *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
[3] O. Al-Debagy and P. Martinek, “A comparative review of microservices and monolithic architectures,” in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 149–154, 2018.
[4] “Nginx documentation.” https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/, 2021.
[5] J. Ghofrani and D. Lübke, “Challenges of microservices architecture: A survey on the state of the practice,” 05 2018.
[6] C. R. Ozansoy, A. Zayegh, and A. Kalam, “The real-time publisher/subscriber communication model for distributed substation systems,” *IEEE Transactions on Power Delivery*, vol. 22, no. 3, pp. 1411–1423, 2007.
[7] “.net core documentation.” https://docs.microsoft.com/en-us/dotnet/, 2021.
[8] “Mongodb documentation.” https://docs.mongodb.com/, 2021.
[9] “Adding salt to hashing: A better way to store passwords.” https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/, 2021.
[10] “Mariadb documentation.” https://mariadb.com/kb/en/documentation/, 2021.
[11] “Node.js documentation.” https://nodejs.org/en/docs/, 2021.
[12] “Redis documentation.” https://redis.io/documentation, 2021.
[13] “Socket.io documentation.” https://socket.io/docs/v3/index.html, 2021.
[14] “Json documentation.” https://www.json.org/json-en.html, 2021.
[15] “React documentation.” https://reactjs.org/, 2021.
[16] “Bootstrap documentation.” https://getbootstrap.com/docs/5.0/getting-started/introduction/, 2021.
[17] “Docker documentation.” https://docs.docker.com/, 2021.
[18] D. Jaramillo, D. V. Nguyen, and R. Smart, “Leveraging microservices architecture by using docker technology,” in *SoutheastCon 2016*, pp. 1–5, 2016.
[19] “Jenkins documentation.” https://www.jenkins.io/doc/, 2021.
[20] “Amqp documentation.” https://www.amqp.org/, 2021.