18th International Symposium INFOTEH-JAHORINA, 20-22 March 2019

Lightweight microservice architecture for small data center monitoring supported with RabbitMQ

Milan Malić
Panonit
Novi Sad, Serbia
milanmalic@outlook.comline

Dalibor Dobrilović
University of Novi Sad / Technical Faculty "Mihajlo Pupin"
Zrenjanin, Serbia
dalibor.dobrilovic@uns.ac.rs

Dušan Malić
Technical College of Applied Sciences in Zrenjanin
Zrenjanin, Serbia
dmalic@sbb.rs

Abstract—**The main focus of this paper is on microservice architecture and analyses of its performance in real time environment for monitoring a small data center. Implementations of IoT technologies and standards in different industries, during last few years, bring cost reductions to the companies that implemented them. One of the main benefits of IoT systems unite collection of data in real time and data transfer, using diverse communication protocols, to the centralized application for analyses. Current approaches in developing applications not proved itself to be good enough in scenarios when a significant amount of data needs to be analyzed. In this paper authors present the proposal for lightweight microservice architecture developed with .NET Core and RabbitMQ that use MongoDB and SQLite databases systems for storing data collected with IoT devices. Also, proposed system evaluation and research results in different stress environments are presented. Because of its complexity, only the most significant segments of architecture will be presented in this paper.**

Key words - microservices; IoT; MongoDB; SQL; .NET Core; data center, RabbitMQ

I. Introduction

In the past few years, numerous studies have shown the importance of using IoT devices in industries, as well as the savings they bring. Direct and indirect benefits of such solutions have brought companies to significant financial cost reduction and acquisition of new kinds of information they were not aware of. This approach brings a substantial inflow of new data collected with the help of IoT devices, which have disrupted the functionalities of the applications used so far. The architecture of monolithic applications is unable to respond to the rigorous requirements needed for applications that should function in real time, and therefore it is necessary to respond to this problem in changing approach in application development.

Forbes, in cooperation with Hitachi, conducted a survey [1]. The results of the survey indicate that over 64% of participants consider IoT today as a significant segment of their company and their operations, while 36% remained neutral. None of the participants considered the IoT to be insignificant. When asked about the future, 91% of participants answered that IoT considered a very important segment of their business. Only 9% of participants remained neutral, while none of the participants considered it insignificant. Research [2], conducted by the Capgemini Digital Transformation Institute, points to the growth in the use of IoT devices in many industries around the world. According to the study of the Statistic [3] by 2025, over 75 billion of connected IoT devices are expected, three times more than today.

Previously mentioned statistics point to the importance of IoT devices in the operations of modern companies, both today and in the near future. In order to respond, in a more efficient way to the massive inflow of data that IoT devices generate, it is necessary to develop applications in order to enable easier processing. data storage and prepare them for future real-time analysis. In this paper is proposed lightweight microservice architecture system developed in C# programming language whose main functionality is small data center monitoring. In order to provide cross-platform support, the .NET Core 2.2 framework will be used. For the messaging system, within the microservice system, RabbitMQ will be used. The data storage system consists of NoSQL MongoDB to store all incoming messages, while the SQLite relational database will be used to store processed data.

The contribution of this paper is the proposal of the lightweight system architecture for monitoring small data centers as well as the methodology for its testing. The two stress tests are performed in order to evaluate architecture components, especially DBs used for data storage.

In Section II are presented the previous researches both in the field of microservice oriented architectures and data warehousing. In Section III architecture of the data center as well as the architecture of the proposed lightweight system based on microservices will be presented. The results of the system evaluation, as well as a comparative analysis of two different tress tests, will be presented in Section IV. Finally, Section V will give a concluding remarks as well as future research directions.

II. Related Research

The savings made by usage of IoT devices in everyday life are numerous. The authors [4] present a platform for the monitoring and management of a wide variety of energy-related agents in buildings, emphasizing the importance of proper management and analysis of the collected data. On the other hand, research [5] presents potential opportunities available in the IoT embedded sustainable supply chain for Industry 4.0. As it can be concluded, IoT devices are becoming more and more present in various aspects of everyday life. In order to respond effectively to a large amount of generated data, transferred from the IoT devices, it is necessary to consider other aspects of the application architecture development. The authors in [6] present the challenges which modern applications are facing during their life cycles, and what new paradigms are adopted by the companies during this process. Also, they point out what rigorous standards need to be fulfilled by modern applications. As a potential solution, they propose application based on micro-service architecture. Similar, the authors [7] present the advantages that are applied to microservice architecture such as agility, insulation, and resistance, elasticity, robustness, and tolerance for errors, scalability, etc. On the other hand, the architecture with characteristics such as a higher complexity of the system and management of distributed data is presented.

The central segment of the micro service-based architecture is communication between microservices itself [8, 9, 10]. Message brokers are used to handle the transmission of messages between system components and they represent the backbone of the system. Today, there are many message brokers both in open source and in close source versions, each of which has its advantages and disadvantages. For this research RabbitMQ developed by Pivotal [11] was used. Through research [12], RabbitMQ managed to receive and deliver over a million messages per second. This results place him to the level of high-performance brokers.

Today, more and more companies demand, as one of the main requirements for IT departments, services able to work between different and multiple platforms. In order to achieve this in June 2016, Microsoft developed a new framework, .NET Core, which is open source and can be used on all platforms with identical source code. The advantages that this framework offers are numerous. In the research [13] was indicated that the Bing Internet search engine with switching to version 2.1. of .NET Core framework achieved 31% better performances. According to IEEE research [14], C# is positioned in the five most popular programming languages. Based on this, it was decided that the development of the microservice system will be made with the usage of C# .NET Core with last stable version 2.2.

A large amount of data requires an efficient way of storing data. The research presented in [15] indicates a potential solution for using MongoDB for BigData storage and real-time analytics. NoSQL MongoDB uses BSON [16, 17] to store zero or more key/value pairs as a single entity. This entity is also called a document. In this way, better performance is achieved because there is no conversion of data, from a human-readable such as JSON to a binary. This indicates that MongoDB is a good solution for microservice architecture. In [18], the analysis of the convenience between NoSQL and NewSQL for use in BigData is performed.

III. SYSTEM ARCHITECTURE

The research covers three main elements: Data Center (A), Test Laboratory (B) and micro service-based architecture system (C). The Data Center was implemented as part of the MIS ETC 1379 project "Cross-border access infrastructure to high-level education through webcasts (EduWebCast)" within cross-border cooperation program Romania-Serbia and funded by the European Union under the instrument for pre-accession assistance (IPA) and co-financed by the participating countries.

The test environment is a development laboratory composed of a large number of computers for professional use. The primary purpose of such environment is the development of applications and their evaluation and testing.

Finally, the third element is system for monitoring and data processing and it is based on microservice architecture. The system consists of a number of components (microservices), message brokers, NoSQL, and SQL data storage systems.

A. *Data Center Architecture*

The function and detailed description of the system is given in [19, 20]. The central parts of the system are two stacked Brocade ICX 6610 switches, and they are shown in Fig. 1 as (1). Since the presented application is planned to be used for data center monitoring it is important to describe what environmental parameters can be acquired with sensors currently deployed in the system. These two devices give the following temperature sensor readings: cooler, MAC, CPU, sensor A, sensor B, sensor C, sensor D, and a stacked card. The MAC temperature is used as the reference temperature of the device. Two stacked switches are placed in the central part of the cabinet and the lower Brocade switch has a higher measured temperature, and this temperature was taken as the reference temperature of the entire cabinet and as one of the most important parameter for monitoring.

Next element in the system is network storage device (Fujitsu Eternus DX200 D3) with 12 SAS disks with a capacity of 600GB, giving a total of 7.2 TB storage capacity (2). Ten disks are set in the working mode, and two are backup disks. Disk RAID is set to High Performance (RAID1 + 0). The total disk capacity of 2.67 TB is organized as follows: 900 GB uses server (3), and 1.79 TB uses a server (4).

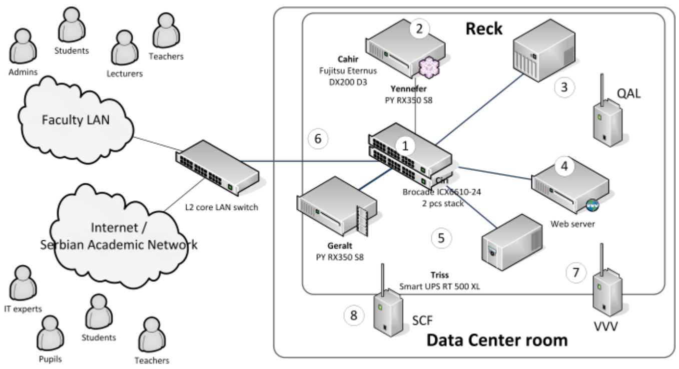


Figure 1. *Data center architecture*

The primary server is the Fujitsu PY RX350 S8 with 64GB of RAM, two Xeon E5-2697 v2 12C/24T 2.7GHz CPUs, and two Tesla K20X GPUs (4). The secondary server is Fujitsu RX300 S8 has with two Xeon E5-2697 v2 with 10 cores (3).

Temperature data of the variety of components of these two servers are collected in current system. In addition to temperature, there is the ability to monitor CPU and DRAM memory energy consumption, and the hard disk and memory utilization. The system is powered with a 5000VA UPS system (5). This unit has sensors for monitoring the temperature, output voltage, frequency, power, and current consumption.

The small data center is additionally monitored with sensor devices integrated in sensor stations built with Arduino/Genuino UNO development boards and open-source hardware. Three stations are used. SCF station uses the DHT11 and BH1750FVI sensors to monitor temperature, humidity, and light levels. The QAL station uses the Grove temperature sensor and the Bosch BMP85 sensor to monitor temperature and barometric pressure. VVV station uses BMP85 and DS18B20 sensors to monitor temperature and barometric pressure. All three sensor stations are able to acquire three sensor readings. Communication between the Arduino/Genuino devices is based on the wireless ZigBee technology.

Three sensor stations are placed at different locations in the data center in order to collect ambient temperature. The QAL station (6) is located inside, the VVV station (7) is located at the top, and the third station SCF (8) is located at a distance of 2m from the cabinet.

Real data collected via SCF, QAL and VVV stations, during the system operation, will be used as dummy data in the development of the new lightweight system based on the microservice architecture for monitoring and analysis. This dummy data will be used for evaluation and testing purposes. Messages generated by stations and transferred with ZigBee tehnology are in the following format: *stationID*, *sensor1 readings*, *sensor2 readings*, *sensor2 readings*. In front of these information, timestamp is inserted to help in analyzing of how much time passes from moment of generating a message to is final storage in database.

B. *Test & Development Hardware Architecture*

The following platform is developed for testing purposes. Its primary function is to be used for analyses of time needed to process and to store data with proposed system architecture and usage of microservices. Also, the big advantage of this platform is in its mobility. All equipment is easily movable and transferable. With such devices in a short period many types of tests can be performed.

This platform contains several computers and network devices. The main computer that runs all microservices has following configuration:

- Intel-based i7-8700k 6 core / 12 threads CPU,
- Asus Z-370 motherboard
- 16 Gb DDR4,
- M.2 NVMe Samsung 960 256Gb,
- 1 Gbit/s network interface
- Nvidia Asus 1070 8 Gb
- Windows 10 x64 OS

Laptop Lenovo T440 is used for running RabbitMQ. The laptop has following configuration:

- Intel i5-4300U vPRO with 2 cores / 4 threads
- 8 Gb DDR3
- Intel 256 SSD
- 1 Gbit/s network interface
- Ubuntu Server 18.04 x64 LTE

As network device, MikroTik router RB2011UiAS-2HnD-IN is used. This device has five 1Gbit ports and five 100 Mbit ports. It also has a WiFi connection. The latest version of RouterOS is installed on the router with a level 5 license. Gigabit network is used for communication between the central computer and Lenovo T440.

Simulation software is used to simulate QAL and VVV stations and to transmit data. This software is deployed on 2 laptops and connected via WiFi 54Mbits network. In addition, the simulator of SCF station is deployed on the central computer. The configuration of these two laptops is irrelevant for this research.

C. *Microservice Architecture*

As mentioned in the introduction section, all microservices are written in the C# programming language supported by .NET Core 2.2. framework. The system consists of several components as it is shown in Fig. 2.

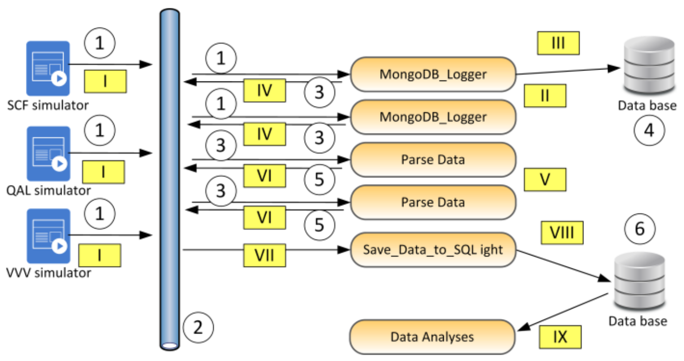


Figure 2. *Schematics of data flow in microservice system with following elements: (1) task_queue, (2) RabbitMQ, (3) data_to_parse, (4) MongoDB database, (5) data_to_save, (6) SQLite database*

As it can be seen from Fig. 2. station simulators are directly connected (I) to *RabbitMQ* (2) and send messages to it. In this model, all messages are sent to the same queue *task_queue* (1).

After publishing the message to the queue, an event is generated on the microservice, and the message is processed further by one of the available *MongoDB_Logger* micro-services (II). For testing purposes in this research, the Round Robin system is implemented to download messages between two instances of micro-services. The downloaded message is forwarded to the *MessagesDB* collection located on MongoDB and placed in the appropriate collection (III). The timestamp is added when the message is taken and sent to the database. In addition to storing a message, the microservice sends a new message (IV) to a new queue *data_to_parse* (3).

Publishing message to the queue *data_to_parse* generates an event on the *ParseData* microservice (V). After that, a message is taken, and the same is processed in the appropriate data model. As in the first case, a timestamp processing is added. After the processing is done, a new message is published (VI) on the queue *data_to_save* (5). Round Robin system is used for taking messages from a queue by two instances of ParseData microservice.

Finally, after setting the message to the queue *data_to_save*, the event (VII) is generated on the microservice *SaveDataToSQLite*. The only task of this microservice is to forward the message with the processed data to store in the SQLite database. The Entity Framework Core was used to communicate with the database (VIII). Table in SQLite has trigger places on Insert, so timestamp is added after insert. The syntax for the trigger is:

CREATE TRIGGER log_insert_new_record_after_insert AFTER INSERT ON Messages BEGIN update Messages SET InsertDateTime = strftime("%Y-%m-%d %H:%M:%f", "now", 'localtime') WHERE ID = NEW.ID; END

After the data is stored in the SQLite database, the data application analytics can use the data for further processing. Communication with the SQLite base, as in the previous case, was achieved through Entity Frameworks Core (IX).

It's important to note that the system is not designed to allow simultaneous connections of several microservices to the database. In the initial phase of the system development, no competing queries and database modifications can be made at the same time. The reason for this approach is that only one record in the current system configuration can be downloaded from the message broker and only one record can be inserted at a given moment. On the other hand, there may be more concurrent readings from the base and this does not affect the system performances.

IV. Research Results

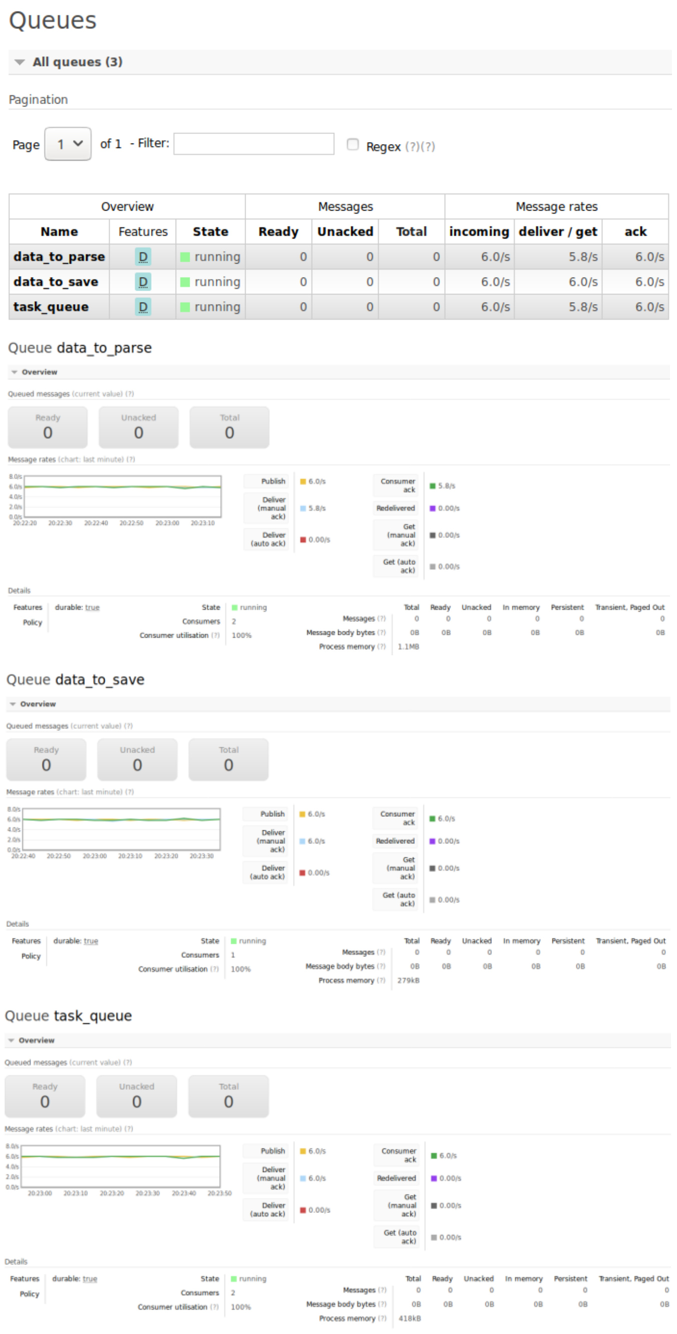
Two stress tests are performed in this research in order to evaluate proposed lightweight architecture. During each test, each of the station simulators sent 10,000 messages so that in the end, each database should have 30,000 records. During the first test simulators generated a message every half of a second and during the second test every 0.1 seconds. In this way, simulation of the stressful environment will be achieved and the arrival of multiple messages on the system will be simulated with the goal to collect performance results in these two modes of operation. At the same time, the identification of the system bottlenecks will be possible in the early stages of development in order to avoid them in the future.

During the execution of each test, RabbitMQ behavior was analyzed using web client monitoring tool that comes with its official installation. In this case, the number of messages on each queue as well as the amount of incoming and sent messages per second was monitored.

The first statistic data show the average time needed between sending the message by the simulator and forwarding this entry to MongoDB. Second statistics, shows the average time between the sending data to the MongoDB and the processing of received message. The third statistical data presents the average time difference between processing and entering the data in the SQLite database.

During the first test, there were no major problems during processing and data storing. RabbitMQ received messages and forwarded them to its subscribers without a problem, as it can can be seen in Fig. 3.

As shown in Fig. 3 statistical data recorded with RabbitMQ monitoring tool showed that in every queue average of 6 messages per second are processed. For every processed message microservices sent the acknowledgment back to RabbitMQ notifying it that message was successfully processed and that it can be deleted from the queue.



Queues

All queues (3)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
data_to_parse	D	🟩 running	0	0	0	6.0/s	5.8/s	6.0/s
data_to_save	D	🟩 running	0	0	0	6.0/s	6.0/s	6.0/s
task_queue	D	🟩 running	0	0	0	6.0/s	5.8/s	6.0/s

Queue data_to_parse

Overview

Queued messages (current value) (?)

Ready 0 Unacked 0 Total 0

Message rates (chart: last minute) (?)

Publish	6.0/s	Consumer ack	5.8/s
Deliver (manual ack)	5.8/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features durable: true
Policy
State running
Consumers 2
Consumer utilisation (?) 100%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	1.1MB					

Queue data_to_save

Overview

Queued messages (current value) (?)

Ready 0 Unacked 0 Total 0

Message rates (chart: last minute) (?)

Publish	6.0/s	Consumer ack	6.0/s
Deliver (manual ack)	6.0/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features durable: true
Policy
State running
Consumers 1
Consumer utilisation (?) 100%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	279kB					

Queue task_queue

Overview

Queued messages (current value) (?)

Ready 0 Unacked 0 Total 0

Message rates (chart: last minute) (?)

Publish	6.0/s	Consumer ack	6.0/s
Deliver (manual ack)	6.0/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features durable: true
Policy
State running
Consumers 2
Consumer utilisation (?) 100%

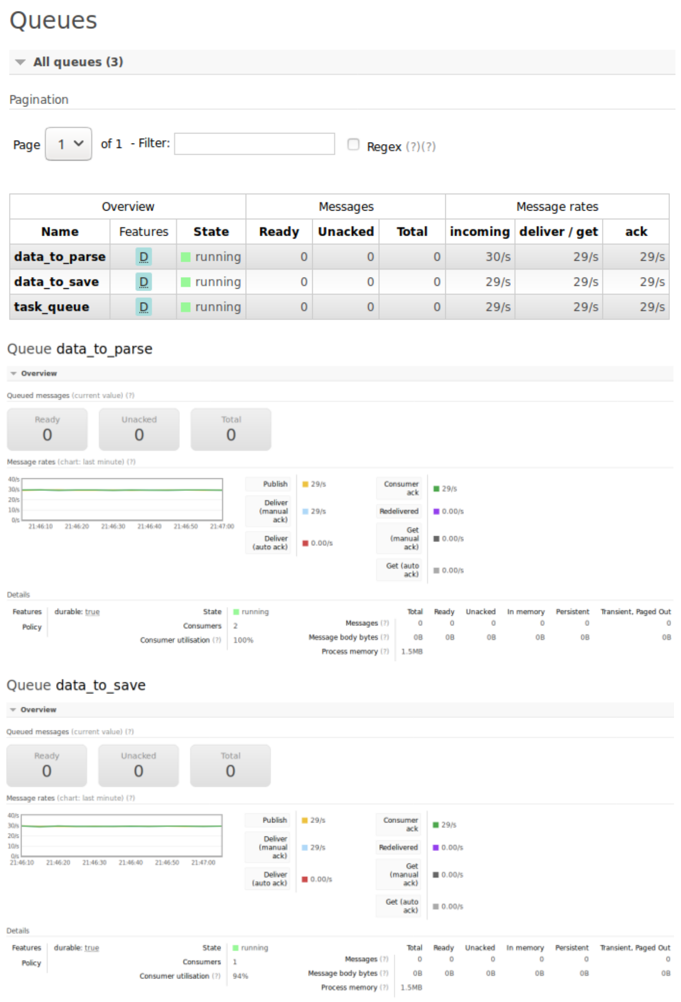
	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	418kB					

Figure 3. *Statistical data of RabbitMQ performance during the first test*

On the other hand, statistical data about computer resources utilization showed that RabbitMQ takes per queue from 279 KB to 1.1MB of memory. All retrieved statistical data indicates that there were no delays and that all messages were forwarded to the processing task in real time. For each received message, its arrival to the destination is acknowledged.

The average time required to store the data to MongoDB from the moment of generation of the message is 331 milliseconds in the first test. It should be emphasized that the data transmission time is included in the message transfer time. The average data processing time is about 3.4 milliseconds, while in the first test the average time for entering data from processing up to the entry in the SQLite database is about 4.7 milliseconds. Also, it is necessary to emphasize that the maximum recorded time required from generating the data up to its registration in MongoDB is 5.337 milliseconds. Also, the maximum recorded processing time is 118 milliseconds and time needed for saving data to SQLite is 614 milliseconds.

During the second test, similar data was recorded, although it should be emphasized that on several occasions there were delays in process of storing data in the SQLite database. As in the first test, RabbitMQ correctly received messages and sent them to their subscribers. Figure 4. gives a statistical overview of RabbitMQ during the second test.



Queues

All queues (3)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
data_to_parse	D	running	0	0	0	30/s	29/s	29/s
data_to_save	D	running	0	0	0	29/s	29/s	29/s
task_queue	D	running	0	0	0	29/s	29/s	29/s

Queue data_to_parse

Overview

Queued messages (current value) (?)

Ready	Unacked	Total
0	0	0

Message rates (chart: last minute) (?)

Publish	29/s	Consumer ack	29/s
Deliver (manual ack)	29/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features: durable: true; Policy

State: running; Consumers: 2; Consumer utilisation (?): 100%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	1.5MB					

Queue data_to_save

Overview

Queued messages (current value) (?)

Ready	Unacked	Total
0	0	0

Message rates (chart: last minute) (?)

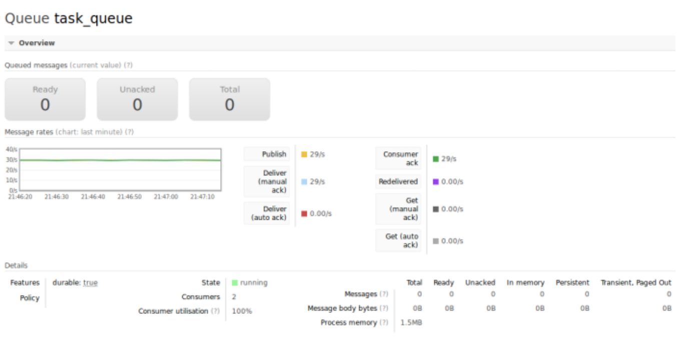
Publish	29/s	Consumer ack	29/s
Deliver (manual ack)	29/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features: durable: true; Policy

State: running; Consumers: 1; Consumer utilisation (?): 94%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	1.5MB					



Queue task_queue

Overview

Queued messages (current value) (?)

Ready	Unacked	Total
0	0	0

Message rates (chart: last minute) (?)

Publish	29/s	Consumer ack	29/s
Deliver (manual ack)	29/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
		Get (auto ack)	0.00/s

Details

Features: durable: true; Policy

State: running; Consumers: 2; Consumer utilisation (?): 100%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages (?)	0	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B	0B
Process memory (?)	1.5MB					

Figure 4. *Statistical data of RabbitMQ performance during the second test*

In Fig. 4 statistical data retrieved with RabbitMQ monitoring tool are presented. In the second test, average of 30 messages per second are processed in every queue. As in the previous test, every microservices sent an acknowledgment message back to RabbitMQ that the messages was successfully processed and that it can be deleted from the corresponding queue. Total of 30,000 messages are received and processed via microservices.

Looking at statistical data on computer resource utilization, it can be noticed that RabbitMQ uses per queue around 1.5MB. In this case, the majority of delays occurred during data storage process in SQLite database.

During the second test, average time required to store data in MongoDB from the moment of generating the message is about 440 milliseconds. About 3.2 milliseconds was needed for data processing, while the average time required to enter data in the SQLite database was 1,330 milliseconds. These results indicate the main problem with proposed architecture and usage of SQLite in it. Also, it is necessary to point out that the maximum recorded time required for the entry of one record in the SQLite database, from the moment of its processing is 29.646 seconds.

V. Conclusion

Modern companies are increasingly taking advantage of IoT devices to collect massive amounts of data. The IoT could lead companies to numerous cost reductions. Together with the advantages, a large amount of data generated by these devices has set rigorous standards for the development of new applications that need to be able to process and store data in real time.

New architectures and approaches in the development of applications with the usage of microservices can respond to imposed challenges. As shown in this paper, processing of data and their storage is possible with the utilization of such architectures. In this paper is presented the proposal of lightweight system developed for small data center monitoring. The development of this system is in the initial phase, and the results indicate the possibility of its application in real environments. Further research will be performed on the data center system described in the paper.

The test platform, developed for testing and evaluation of the the proposed system is also described in this paper. This platform showed that proposed lightweight architecture based on microservices could deal with the larger amount of sensor data in the case of using MongoDB. On the other hand, the usage of SQLite database is not recommended due to the lower performances and test results. The SQLite database shows a lower performance in the case of the higher data amount generated from the sensor stations. For future usage, the SQLite will be replaced with other DBMS.

ACKNOWLEDGEMENT

This research is supported by the Ministry of Education and Science of the Republic of Serbia under the project number TR32044 "The development of software tools for business process analysis and improvement," 2011-2019.

REFERENCES

[1] Forbes Insights, “The Internet of Things: From theory to reality - How companies are leveraging the IoT to move their businesses forward”, [Online]. Available: http://info.forbes.com/rs/790-SNV-353/images/Hitachi%20IoT%20 Report.pdf, 2017. [Accessed: January 2019]

[2] Capgemini Digital Transformation Institute, “Unlocking the business value of IoT in operations”, [Online]. Available: https://www.capgemini.com/wp-content/uploads/2018/03/dti-research-report_iot-in-operations_web.pdf, [Accessed: January 2019]

[3] Statista, “Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)”, [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2019. [Accessed: January 2019]

[4] F. Terroso-Saenz, A. González-Vidal, A. P. Ramallo-González, A. F. Skarmeta, “An open IoT platform for the management and analysis of energy data”, Future Generation Computer Systems, Vol. 92, 2019, pp 1066-1079, https://doi.org/10.1016/j.future.2017.08.046.

[5] E. Manavalan, K. Jayakrishna, “A review of Internet of Things (IoT) embedded sustainable supply chain for industry 4.0 requirements”, Computers & Industrial Engineering, 2018, https://doi.org/10.1016/j.cie.2018.11.030.

[6] B. Götz, D. Schel, D. Bauer, C. Henkel, P. Einberger, T. Bauernhansl, “Challenges of Production Microservices”, Procedia CIRP, Vol. 67, 2018, pp 167-172, https://doi.org/10.1016/j.procir.2017.12.194.

[7] M. Ciavotta, M. Alge, S. Menato, D. Rovere, P. Pedrazzoli, “A Microservice-based Middleware for the Digital Factory”, Procedia Manufacturing, Vol. 11, 2017, pp 931-938, https://doi.org/10.1016/j.promfg.2017.07.197.

[8] S. Newman, “Building Microservices - Designing Fine-Grained Systems”, O'Reilly Media, 2015.

[9] S. Fowler, “Production-Ready Microservices - Building Standardized Systems Across an Engineering Organization”, O'Reilly Media, 2016

[10] R. Rodger, “The Tao of Microservices”, Manning Publications Co., helter Island, NY, USA, 2018. https://www.rabbitmq.com/

[11] Pivotal Software, Inc., “Messaging that just works — RabbitMQ”, [Online]. Available: https://www.rabbitmq.com/, Retrieved January 2019. [Accessed: January 2019]

[12] J. Kuch, “RabbitMQ Hits One Million Messages Per Second on Google Compute Engine”, [Online]. Available: https://content.pivotal.io/blog/ rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine, Posted March 25, 2014. [Accessed: January 2019]

[13] D. Ramel, “.NET Core 2.1 Powers 34 Percent Bing Performance Boost”, [Online]. Available: https://visualstudiomagazine.com/articles/2018/ 08/22/bing-net-core.aspx, Posted 08/22/2018. [Accessed: January 2019]

[14] S. Cass. P. Bulusu, “Interactive: The Top Programming Languages 2018”, [Online]. Available: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages, Posted 31 Jul 2018. [Accessed: January 2019]

[15] A. Celesti, M. Fazio, “A framework for real time end to end monitoring and big data oriented management of smart environments”, Journal of Parallel and Distributed Computing, 2018, https://doi.org/10.1016/j.jpdc.2018.10.015.

[16] BSON Specification Version 1.1, [Online]. Available: http://bsonspec.org/spec.html, Retrieved January 2019. [Accessed: January 2019]

[17] B. Dayley, “NoSQL with MongoDB in 24 Hours, Sams Teach Yourself”, Pearson Education, USA, 2015.

[18] P. Raj, “Chapter One - A Detailed Analysis of NoSQL and NewSQL Databases for Bigdata Analytics and Distributed Computing”, Editor(s): Pethuru Raj, Ganesh Chandra Deka, Advances in Computers, Elsevier, Vol. 109, 2018, pp 1-48, https://doi.org/10.1016/bs.adcom.2018.01.002.

[19] M. Marcu, S. Ficiu, D. Dobrilovic, M. Popa, B. Odadzic, „Cross-border infrastructure for educational webcasting“, Proceedings of International Conference on Applied Internet and Information Technologies AIIT 2015, pp 225-230, October 23, Zrenjanin, Serbia, 2015.

[20] S. Fuicu, M. Popa, D. Dobrilovic, M. Marcu, R. Bogdan, “Developing Distance Learning Environments in the Context of Cross-Border Cooperation”, BRAIN Broad Research in Artificial Intelligence and Neuroscience, pp 52-58, Vol. 8, Issue 1, April, 2017.

[21] D. Dobrilović, Ž. Stojanov, Z. Čović, J. Simon and N. Petrov, "Model of data center temperature monitoring system with the use of open source hardware," 2016 IEEE 14th International Symposium on Intelligent Systems and Informatics (SISY), Subotica, 2016, pp. 221-226. doi: 10.1109/SISY.2016.7601501