UDC 004.5:004.65
DOI: 10.15587/2706-5448.2021.238460
Article type «Reports on Research Projects»

**Valerii Tkachenko,**
**Svetlana Lukianiuk**

# ANALYSIS OF THE USE OF THE REDIS IN THE DISTRIBUTED ORDER PROCESSING SYSTEM IN THE RESTAURANT NETWORK

*The object of research is a distributed order processing system for a restaurant chain. The subject of the research is the analysis of the use of Redis for managing event queues in distributed systems.*

*When implementing a distributed order processing system in a restaurant chain with a possible load of up to 20,000 users per day, the Redis system was used. Management of 9 distributed subsystems was organized through Redis. This solution showed an increase in the performance of the system under heavy load (from 50 transactions per second), but the response time of the system in some cases of its operation was longer than without using Redis. When working systems using Redis, it is necessary to take into account the amount of data with which Redis will work, since it does not exceed the amount of RAM, the absence of differentiation into users and groups, and the absence of a query language, which is replaced by a key-value scheme.*

*This research is aimed at analyzing the operation of the system during trial operation under real load. We compared the operation of a configured system with Redis enabled and disabled. The main indicators for the analysis were the system response time and the maximum request execution time. The research was carried out for 2 weeks, the first week using the system settings with disabled Redis, the second – with enabled Redis. We selected 2 days with a similar load on the system to each other. Especially indicative are the results of comparing the durations of the longest queries, which show an almost constant value of the duration for the system in the mode of enabled Redis. The hypothesis of an increase in the system response time at low loads was confirmed, but this value not only leveled off at a load of 500 unique users but also became less at loads of 1000 unique users.*

**Keywords:** *microservice, service-oriented architecture, order processing, Redis, software development, software engineering.*

*Received date: 21.04.2021*
*Accepted date: 07.06.2021*
*Published date: 30.09.2021*



***How to cite***
*Tkachenko, V., Lukianiuk, S. (2021). Analysis of the use of the Redis in the distributed order processing system in the restaurant network. Technology Audit and Production Reserves, 5 (2 (61)), 39–43. doi: http://doi.org/10.15587/2706-5448.2021.238460*

## 1. Introduction

The Redis system (from Redis Labs) was used to implement a distributed order processing system in the restaurant chain, which has more than 20 establishments, and the load on network services ranges from 15 to 20 thousand users per day. Redis organizes event storage using an immutable mechanism [1]. An immutable mechanism is an analog of a transaction log that only supports the addition of new data to existing data, but not the modification of existing ones.

The implemented order processing system uses 9 distributed services, the operation of which is regulated by Redis [2]. Management is implemented on the frontend in the form of a library, which makes requests for keys directly to the desired servers, which work only with local data. The method of secure access was used to work with databases [3].

When implementing the updated system at the testing stage, this solution showed an increase in the speed of the system under heavy load (from 50 transactions per second), but the response time of the system in some cases was longer than the previous solution.

This may be due to the increased response time of the system, as additional resources are spent on marshalling and unmarshalling. Marshaling means the process of converting an object into a special data format for storage or transmission [1].

The relevance of the research is due to the popularity of the functionality Redis in online systems, but the use of Redis in distributed systems with a microservice architecture requires a comprehensive analysis for each individual implementation.

*The object of research* is a distributed order processing system of a restaurant chain, and *the subject of research* is the analysis of the use of Redis to manage event queues in distributed systems. *The aim of research* is to analyze the work distributed order processing system during experimental operation under real load. It is planned to

ISSN 2664-9969

compare the performance of the configured network with the Redis system on and off. The main indicator will be the response time of the system and the maximum execution time of the request.

## 2. Methods of research

The peculiarity of the implemented solution is the division of the system into 9 separate subsystems (services), each of which is responsible for a separate range of functions (Fig. 1). The separation of subsystem functions took place according to the paradigms of system development based on microservices [4–6].

When a customer, product or order is created/deleted, the event must be transmitted asynchronously [7, 8] to CRM (Customer relationship management) using RESP (Redis Serialization Protocol) to manage interaction with current and potential customers [9, 10]. In the implemented structure, the CRM service can be started and stopped at runtime without any impact on other microservices. This means that all messages sent to the CRM during downtime are pinned for processing.

According to the developed architecture of the distributed order processing system, the client gets access only to the API, which determines which service to transfer data for processing. In Fig. 1 dotted line indicates data transfer via RESP, other data transmissions are conducted via HTTP.

Using Redis in the API subsystem reduces the number of database queries (DB), but requires constant conversion of data to types supported by Redis [1, 9]:

– strings (implemented using the library of dynamic strings C);
– lists (implemented as linked lists);
– sets and hashes (implemented as hash tables);
– ordered sets (implemented as lists with spaces – a special type of balanced trees).

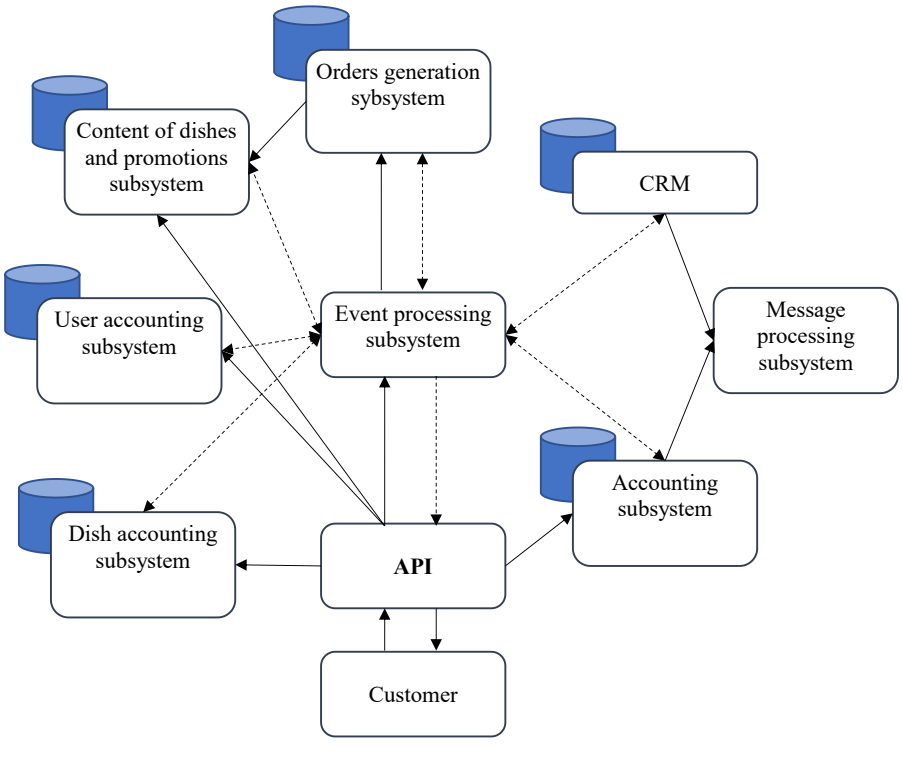


**Fig. 1.** The architecture of the developed system of order processing of the restaurant chain

The main advantages of using Redis, which led to its choice, include:

1) possibility of a delay of up to milliseconds. Redis supports response time in milliseconds. Storing data in memory can read data faster than disk-based databases;

2) ease of use by developers. Redis is syntactically easy to use and requires a minimum amount of code to integrate into the program;

3) data splitting between nodes. This allows to scale the system and process more data as the number of requests increases;

4) support for a wide range of programming languages. Has many open source clients available to developers. Supported languages include Java, Python, PHP, C, C++, C#, JavaScript, Node.js, Ruby, Go and many more;

5) advanced data structures. Redis supports rows, lists, sets, sorted sets, hashes, bits and hyperlog logs;

6) multi-threaded architecture. Because the cache is multithreaded, it can use multiple processing cores. This means that it is possible to process more operations by increasing computing power;

7) ability to take snapshots of the system. With Redis, it is possible to save data to a snapshot disk that it is possible to use for backup or recovery;

8) replication. Redis allows to create multiple copies of the main Redis. This allows to scale database readings and create highly accessible clusters;

9) geospatial support. Redis has specially designed teams to work with real-time geospatial data. It is possible to perform operations such as searching for the distance between two elements (for example, people or places) and searching for all elements at a certain distance from a point.

The main costs of microservices, which they bear in connection with the use of Redis, can be divided into the following groups:

1) CPU resource consumption for marshalling and unmarshalling. The application, when accessing the cache, should «marshalize» data and request. The cache must unmarchalize this data, determine what the request is, process, and marshalize the response. The same marshaling and unmarshalling operations will take place when the application interacts with the database. Studies, which are covered in [1, 9, 10] show that marshalling and unmarshalling increase the load on the processor by 80–85 %;

2) excessive reading and writing time. This is due to cases where the application reads more data from the cache than necessary. According to a research [9], if the record contains only 10 % of the required information, then spend 46 % more CPU resources and 86 % more network resources than is really necessary;

3) network delays. The system wastes time on constant remote access to the cache and databases of services, as well as on the transfer

of the required amount of data in the request-response, especially in combination with unnecessary readings. These losses increase in direct proportion to how many cache and database calls are needed to service a single request from the customer.

Given the shortcomings presented, it is possible to determine the main requirements for system performance in the analysis:

– analysis of the system with on and off Redis should be subject to similar loads;

– load in the system must be determined both by the number of unique users and by the number of transactions they create;

– the main parameters for analysis are the response time of the system and the maximum execution time of the request.

Since most of the workload falls on memcache and its connection to applied logic, the industry has recently focused most of its efforts on optimizing the cache and communication with it.

Redis try to reduce excessive readings by supporting the data structure on the memcache side, which allows to select not all values under lock and key, but only part [9].

It was determined that most of the problems arise precisely because of the remoteness of the required data from the applications. And so it is possible to solve problems simply by transferring data to our logic. Thus, a system with microservice management is implemented as of [10].

Due to the distributed architecture of the system, the standard metrics for evaluating the use of resources (CPU, RAM) do not provide sufficient information on the quality of the implemented solution. That is why it is possible to choose work shifts with a similar load and analyze the response time of the system and the maximum execution time of the request throughout.

To simplify the visualization of the analysis results, it was decided to group the data by hours.

To analyze the operation of the system in similar operating conditions, it was decided to use statistics of working days with similar input conditions. The similarity of the system load for the restaurant chain under analysis was determined by the difference in the number of unique users. The average difference with the grouping by hours within the work shift should not exceed 50, and the total difference within the work shift should not exceed 300. The second parameter of similarity is the difference in the intensity of transactions. This parameter is calculated as the average of the difference between the average values of the number of transactions per second with the grouping by hours should. This parameter should not exceed 5.

## 3. Research results and discussion

The research was conducted for 2 weeks, in the first week the system settings were used with Redis disabled, the second – with Redis enabled. 2 days with a similar load on the system were selected. The data in Tables 1, 2, which are visualized in the graphs of Fig. 2 and Fig. 3 prove the similarity of the selected days. The average difference with the grouping by hours within the work shift is equal to – 15 (on average, 15 users more per day of the system with Redis enabled). The total difference within the work shift is –261 (261 users more per day with Redis enabled). The difference in transaction intensity was –2.55. These indicators and the similarity of the system load schedules confirm the correctness of the selected operating changes to compare the system operation in different configuration modes.

**Table 1**

Comparison of the average number of simultaneous unique users with grouping by hours of work shift

| Time | The average number of simultaneous unique users | |
|---|---|---|
| | at Redis Off | at Redis On |
| 6:00 | 204 | 220 |
| 7:00 | 226 | 196 |
| 8:00 | 202 | 217 |
| 9:00 | 429 | 480 |
| 10:00 | 649 | 726 |
| 11:00 | 642 | 734 |
| 12:00 | 841 | 814 |
| 13:00 | 934 | 1011 |
| 14:00 | 1075 | 1128 |
| 15:00 | 1087 | 1142 |
| 16:00 | 1075 | 1154 |
| 17:00 | 1193 | 1213 |
| 18:00 | 1180 | 1226 |
| 19:00 | 1310 | 1240 |
| 20:00 | 1338 | 1214 |
| 21:00 | 1015 | 945 |
| 22:00 | 904 | 935 |
| 23:00 | 499 | 468 |

**Table 2**

Comparison of the average number of transactions per second with grouping by hours of work shift

| Time | Average number of transactions per second | |
|---|---|---|
| | at Redis Off | at Redis On |
| 6:00 | 29.56 | 28.92 |
| 7:00 | 29.63 | 29.09 |
| 8:00 | 29.48 | 28.84 |
| 9:00 | 29.63 | 29.99 |
| 10:00 | 30.88 | 32.24 |
| 11:00 | 32.33 | 34.69 |
| 12:00 | 35.08 | 35.44 |
| 13:00 | 36.33 | 35.69 |
| 14:00 | 36.58 | 35.74 |
| 15:00 | 35.93 | 36.09 |
| 16:00 | 33.28 | 38.74 |
| 17:00 | 32.83 | 38.49 |
| 18:00 | 31.08 | 37.64 |
| 19:00 | 32.83 | 38.39 |
| 20:00 | 35.38 | 42.94 |
| 21:00 | 41.43 | 44.99 |
| 22:00 | 36.38 | 41.94 |
| 23:00 | 29.33 | 33.99 |

ISSN 2664-9969

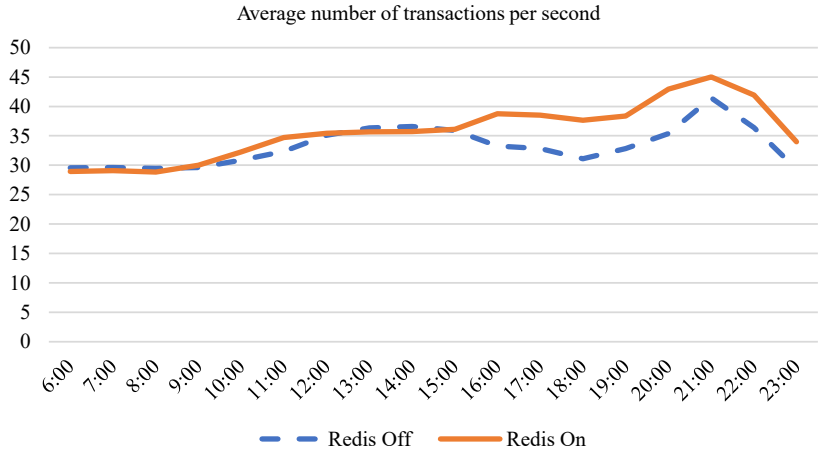


**Fig. 2.** Graphs of the average number of simultaneous unique users per work shift

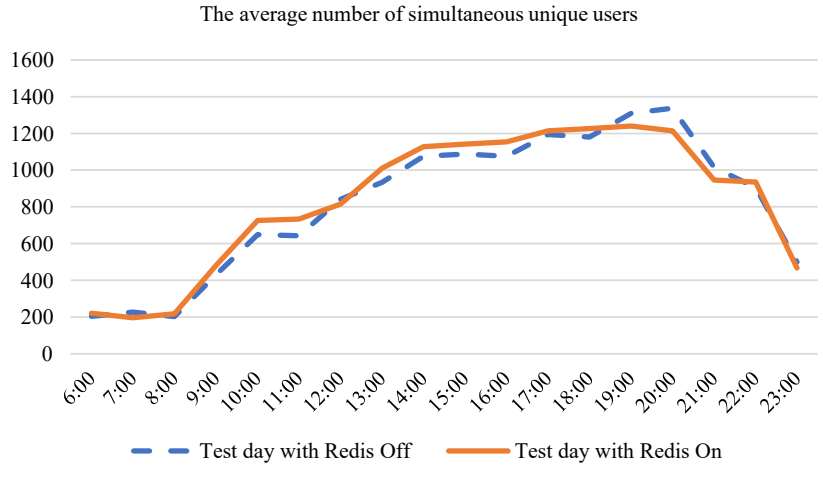


**Fig. 3.** Graphs of the average number of transactions per second per work shift

system in the Redis mode (Fig. 5). The hypothesis of increasing the response time of the system at low loads was confirmed, but this value not only equalized at a load of 500 unique users, but also decreased at loads from 1000 unique users.

**Table 3**

Comparison of the average value of the response time with grouping by hours of work shift

| Time | Response time, p | |
|---|---|---|
| | at Redis Off | at Redis On |
| 6:00 | 0.021 | 0.091 |
| 7:00 | 0.023 | 0.092 |
| 8:00 | 0.026 | 0.093 |
| 9:00 | 0.032 | 0.094 |
| 10:00 | 0.092 | 0.107 |
| 11:00 | 0.112 | 0.108 |
| 12:00 | 0.099 | 0.109 |
| 13:00 | 0.120 | 0.125 |
| 14:00 | 0.182 | 0.132 |
| 15:00 | 0.283 | 0.137 |
| 16:00 | 0.355 | 0.144 |
| 17:00 | 0.333 | 0.135 |
| 18:00 | 0.370 | 0.137 |
| 19:00 | 0.366 | 0.123 |
| 20:00 | 0.248 | 0.095 |
| 21:00 | 0.121 | 0.083 |
| 22:00 | 0.059 | 0.081 |
| 23:00 | 0.027 | 0.080 |

The results of comparing the average value of the response time (Table 3 and Fig. 4) and the maximum duration of the longest queries (Table 4 and Fig. 5) showed the need to use a system with Redis enabled.

Particularly significant are the results of comparing the values of the longest request duration, as shown by the virtually unchanged value of the request duration for the

This research was conducted without taking into account the settings of Redis, which would take into account the frequency of queries to the tables of the database of microservices. Implementing these settings can affect the duration of queries. This will require separate monitoring of the execution time of data queries, which with Redis can limit the number of queries to the database.

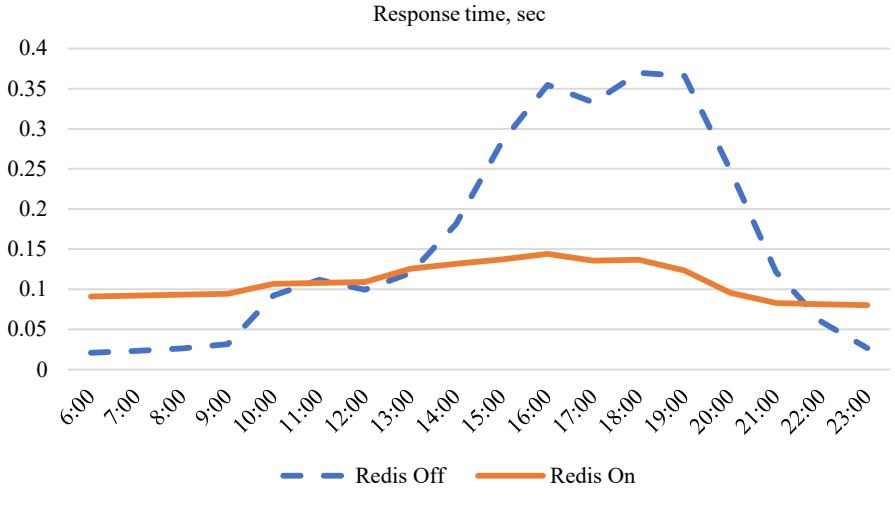


**Fig. 4.** Graphs of the average value of the response time with grouping by hours

**Table 4**

Comparison of the duration of the longest queries

| Time | The longest request, ms | |
|---|---|---|
| | at Redis Off | at Redis On |
| 6:00 | 6.02 | 2.14 |
| 7:00 | 6.68 | 2.16 |
| 8:00 | 7.49 | 2.19 |
| 9:00 | 11.50 | 2.22 |
| 10:00 | 44.54 | 3.43 |
| 11:00 | 64.71 | 3.47 |
| 12:00 | 57.59 | 3.51 |
| 13:00 | 60.33 | 4.04 |
| 14:00 | 61.27 | 4.24 |
| 15:00 | 64.18 | 5.11 |
| 16:00 | 65.31 | 5.08 |
| 17:00 | 61.33 | 4.77 |
| 18:00 | 54.58 | 4.82 |
| 19:00 | 22.43 | 4.34 |
| 20:00 | 15.23 | 3.36 |
| 21:00 | 7.45 | 2.92 |
| 22:00 | 7.09 | 2.17 |
| 23:00 | 6.18 | 2.11 |

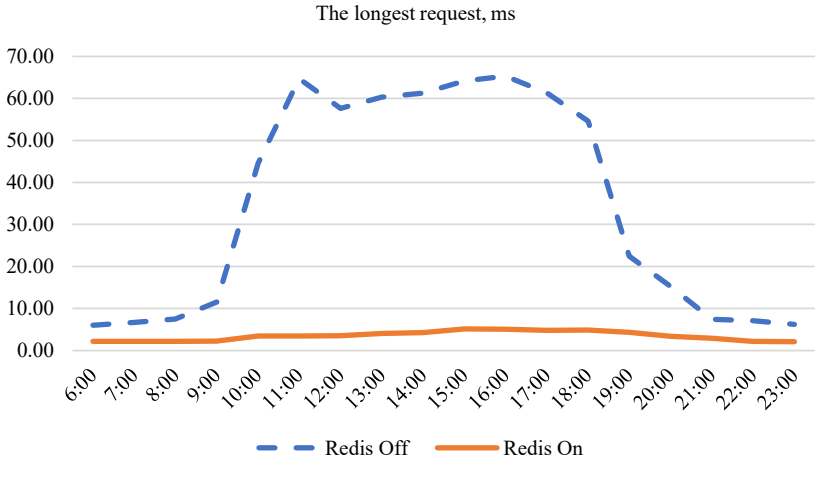


**Fig. 5.** Graphs of the values of the longest queries with grouping by hours

Further research involves finding the optimal setting Redis for work with ultrahigh loading.

## 4. Conclusions

A research of the use of Redis to manage queues in a distributed order processing system in a restaurant chain showed the reasons for the main cost of hardware resources by microservices. The main CPU costs are marshaling and unmarshalling, excessive read and write time, network delays.

The results of comparing the average response time and the maximum duration of the longest queries showed the need to use a system with Redis enabled. The system response time affects the time of the first system response to start generating response data, which affects the page load speed. The execution time of the longest queries shows not only the tendency to increase the processing time of queries at high load on the system but also reports the need to increase hardware resources for servers database management systems (DBMS) in systems without Redis.

The research confirmed the hypothesis of a greater value of the system response time at low loads (91–94 milliseconds with the Redis system on and 21–26 milliseconds when the Redis system is disabled). On loads of 1000 unique users, the system response time with Redis enabled is less than 2–3 times the response time when Redis is off. When the Redis system is enabled, the response time is 123–144 milliseconds, with the Redis system 283 disabled 283–370 milliseconds.

The results of comparing the values of the longest duration of queries show an almost unchanged value for the system in the Redis mode (3.43–5.11 ms) against abrupt change with Redis disabled (44.54–65.31 ms).

## References

1. Ji, Z., Ganchev, I., O'Droma, M., Ding, T. (2014). A Distributed Redis Framework for Use in the UCWW. *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery.* doi: https://doi.org/10.1109/cyberc.2014.50
2. Reagan, R. (2017). Redis Cache. *Web Applications on Azure,* 257–300. doi: https://doi.org/10.1007/978-1-4842-2976-7_7
3. Artamonov, Ye. B., Bieliakov, O. O. (2013). Elektronni skhovyshcha danykh iz zakhyshchenym dostupom. *Naukoiemni tekhnolohiyi, 4 (20),* 402–405.
4. Vural, H., Koyuncu, M., Guney, S. (2017). A Systematic Literature Review on Microservices. *Lecture Notes in Computer Science,* 203–217. doi: https://doi.org/10.1007/978-3-319-62407-5_14
5. Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software, 35 (3),* 24–35. doi: https://doi.org/10.1109/ms.2018.2141039
6. Di Francesco, P., Lago, P., Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software, 150,* 77–97. doi: https://doi.org/10.1016/j.jss.2019.01.001
7. Auer, F., Lenarduzzi, V., Felderer, M., Taibi, D. (2021). From monolithic systems to Microservices: An assessment framework. *Information and Software Technology, 137,* 106600. doi: https://doi.org/10.1016/j.infsof.2021.106600
8. Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., Safina, L. (2018). Microservices: How To Make Your Application Scale. *Perspectives of System Informatics,* 95–104. doi: https://doi.org/10.1007/978-3-319-74313-4_8
9. Liu, F., Li, J., Wang, Y., Li, L. (2019). Kubestorage: A Cloud Native Storage Engine for Massive Small Files. *2019 6th International Conference on Behavioral, Economic and Socio-Cultural Computing (BESC).* doi: https://doi.org/10.1109/besc48373.2019.8962995
10. Chen, S., Tang, X., Wang, H., Zhao, H., Guo, M. (2016). Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. *2016 IEEE Trustcom/BigDataSE/ISPA.* doi: https://doi.org/10.1109/trustcom.2016.0255

✉ ***Valerii Tkachenko,*** *PhD, Associate Professor, Department of Computerized Control System, National Aviation University, Kyiv, Ukraine, e-mail: tkachenkvg@gmail.com, ORCID: http://orcid.org/0000-0002-1759-7267*

***Svetlana Lukianiuk,*** *Researcher, Ukrainian Research Institute of Special Equipment and Forensic Science of the Security Service of Ukraine, Kyiv, Ukraine, ORCID: http://orcid.org/0000-0002-7469-8144*

✉ *Corresponding author*