ISSN 2664-9969

UDC 004.046
DOI: 10.15587/2706-5448.2024.310031

Ihor Kasianchuk

ORCHESTRATION OF SERVICE-ORIENTED APPLICATIONS WITH REACTIVE PROGRAMMING TECHNIQUES

The object of research is the modular approach to application development using SOA, as well as the comparison of synchronous and asynchronous request processing methodologies using a reactive programming architecture. SOA allows applications to be divided into independent components, ensuring easy integration and scalability in distributed computing environments. With SOA, it is possible to create a network of loosely coupled services, providing users with the flexibility to develop applications tailored to specific needs.

One of the main issues is thread blocking and system instability under heavy loads when using synchronous methods. The study compares synchronous and asynchronous request processing methodologies using WebFlux, and examines key components of SOA, such as service discovery mechanisms and interaction models, particularly orchestration and choreography.

The results show that asynchronous approaches, using a non-blocking, event-driven architecture, reduce the number of active threads, increase system resilience, and improve performance. This is because the proposed non-blocking, event-driven approach has several features, including reducing thread blocking and enhancing system stability under heavy loads. Synchronous methods, while straightforward, have drawbacks such as thread blocking and system instability under excessive loads.

As a result, there is a high efficiency in processing a large number of requests in real-time. Compared to similar known approaches, this provides advantages such as increased system resilience and efficient resource utilization, making this approach particularly useful for scalable application architectures in distributed computing environments.

Keywords: *service-oriented architecture (SOA), reactive programming, event loop, asynchronous requests.*

Received date: 10.06.2024
Accepted date: 14.08.2024
Published date: 16.08.2024

How to cite
Kasianchuk, I. (2024). Orchestration of service-oriented applications with reactive programming techniques. Technology Audit and Production Reserves, 4 (2 (78)), 24–29. https://doi.org/10.15587/2706-5448.2024.310031

1. Introduction

The growing importance of the services sector in business and the information technology industry makes the study of interactions between services highly significant. Designing a system based on SOA (Service-Oriented Architecture) allows for the separation of responsibilities of individual components (domains) of the application, making their logic decoupled and independent. Based on SOA, it is possible to build a globally scaled network of loosely coupled services that can be easily composed by users according to their scenarios into flexible applications with a dynamic structure, running in a distributed computing infrastructure.

In a broad sense, SOA is an approach to application development where the application is broken down into separate parts [1]. These parts are generally distributed throughout the system and interact with each other over a network or through APIs. Throughout the brief history of service science, management, and engineering, a general scheme for forming «on-demand» applications has been proposed. This scheme includes three participants: the service provider, the broker, and the service consumer.

The basic procedure for service discovery is illustrated in Fig. 1, which shows a comparison between the user's request and the service description in the registry advertised by the provider [2]. Client requests for necessary services, sent to the service registry, contain parameterized descriptions of tasks in the form of a finite set of input parameters. After processing the client's requests, the service returns the result to the client, formatted as a finite set of output parameters.

Information about the service's input and output parameters, which is necessary for the client to interact with the service, is contained in the published service description or provided by the service upon the client's request. A service can be used by multiple different applications. Conversely, several services can be used within a single application.

The functionality of SOA is most easily implemented using web services. Web services are understood to be software systems that use the XML (Extensible Markup Language) data format, Web Services Description Language (WSDL) standards to describe their interfaces, Simple Object Access Protocol (SOAP) to describe the format of received and sent messages, and the Universal Description Discovery and Integration (UDDI) standard to create catalogues of available services [3]. Although the principles of service-oriented architecture for creating information systems do not necessarily require the use of web service technologies, the connection between these two areas in the development

of information technology is quite close. Web services are technological specifications, whereas service-oriented architecture (SOA) is a design principle for software system architecture. The combination of WSDL, UDDI, and SOAP facilitates the application of service-oriented concepts on the Internet, particularly for publishing, discovering, binding, and executing services.

Applications for business processes that integrate various services are built on loosely coupled services. The necessity of invoking multiple services to meet a need requires specific coordination approaches. SOA initiated this trend using Remote Procedure Call (RPC) technology over a network, while maintaining control over the access component. The main approaches to web service coordination are known as orchestration and choreography, the implementation of which resembles the programming process.

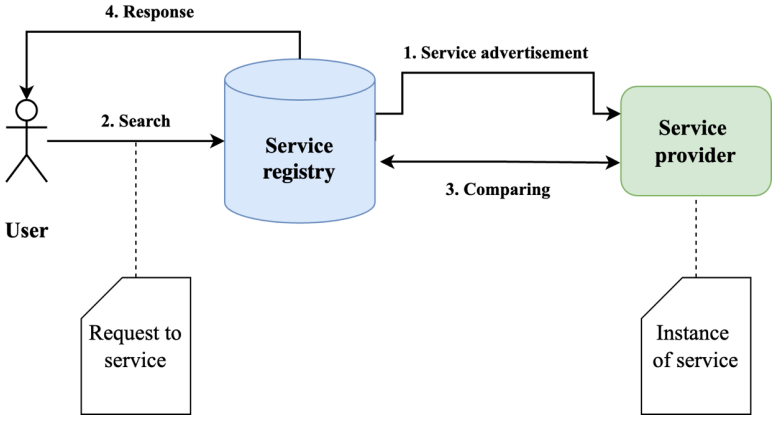

Fig. 1. Service Discovery Structure

The classical approach to service choreography is mostly based on synchronous calls [4]. The client that initiates a business process «freezes» the execution of the software code until a response is received from the invoked service. This necessitates the programming of additional logic to handle errors at the invocation stage.

It makes sense to propagate expected business logic errors through the process so that the user can address the cause and retry the request in the future. However, errors of an infrastructural nature must be corrected at the software level, complicating the process. Such errors include long wait times (due to service unavailability), input data read errors (connection termination), and so on. During synchronous execution, the request may be retried up to a certain threshold of repetitions or until a timeout occurs. Only then is the error propagated to a higher level – the business process – and further steps are blocked.

This behavior pattern is called a «circuit breaker», similar to an electrical circuit that interrupts execution if the signal voltage exceeds a certain threshold (Fig. 2). It represents a finite state machine with three states. The initial state of the object controlling the service call is «closed». If a certain condition that defines the call threshold is met (e. g., the number of failures exceeds 3), the circuit breaker transitions to the «open» state, initiating a timeout period that blocks further calls. When the timeout period elapses, the state changes to «half-open», the request is retried, and the next state depends on the result – «closed» for a successful result or back to «open» for a failure.

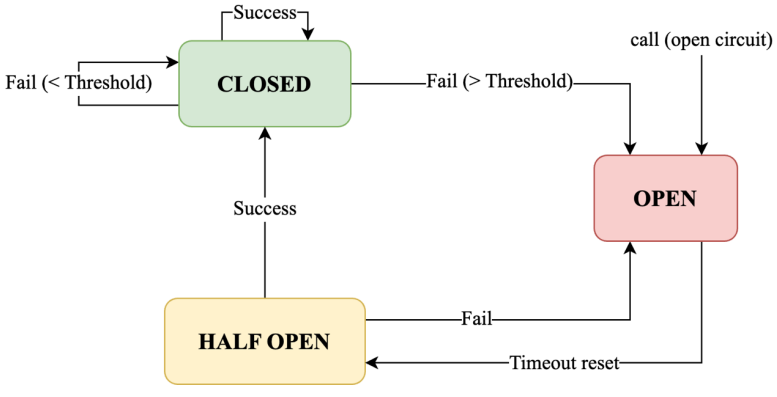

Fig. 2. Pattern «Circuit breaker»

In the worst-case scenario (where calling an external service fails), such an approach generates a sequence of workflows that cease execution with an error state. For small data volumes, after resolving the infrastructure issue, the processes can be restarted and continue execution from the point where the error occurred.

The choreography of erroneous processes becomes more complex with increasing data volumes. For instance, an application that processes new data as it arrives (implemented on the «publisher-subscriber» principle) will instantly launch processes in large quantities. Using traditional synchronous data transfer protocols, the process creator's resources will be limited by the number of threads available in the application instance, managed by the operating system. This leads to the generation of exceptions, which also stop the workflow execution with an error result.

In an application that processes an infinite stream of data, choreography and orchestration based on services managed by synchronous requests can lead to a large number of threads, whose execution becomes frozen. An architectural solution that allows for asynchronous execution can prevent most of these situations. The goal of this approach is to delegate to the system the further processing of requests that halted due to infrastructure errors, leaving the process designer with only the burden of managing the business logic.

The aim of research is to identify the relationship between the architectural patterns for processing service requests and the load on system resources. This will allow for the selection of the appropriate approach for designing service-oriented applications based on the expected workload.

2. Methods and Materials

In general, SOA is a model for component interaction that connects different functional modules of applications (services) through clearly defined interfaces. These interfaces are independent of hardware platforms, operating systems, or programming languages used in the development of these applications. This allows individual services to interact with each other in a standardized yet

ISSN 2664-9969

universally applicable manner, a characteristic known as «loose coupling».

The main approaches to coordinating web services are known as orchestration and choreography. Orchestration, when seeking a single service, is associated with implementing a synchronous «request-response» pattern, while choreography involves interaction between a service and its consumer using a «publish-subscribe» pattern.

The proposed method for web service interaction is based on using reactive programming. Integrating a reactive system reduces the number of active application threads, creating conditions for improved reliability and fault tolerance.

Fig. 3 depicts a traditional architecture of an application using resources (endpoints) based on the REST protocol. According to the nature of this architecture, each request creates a separate thread. Requests that do not find available threads typically enter an internal queue and wait for a thread to become available. If no threads are available within a certain timeframe, the request returns an error.

In the traditional model of handling requests, when a thread reaches the Request handler and executes an operation that blocks further execution (such as a database call or cache retrieval), the thread enters a waiting state. This consumes hardware resources and blocks the execution of new requests. As the number of requests increases, this model can lead to a large number of blocked threads, which in turn freezes the execution of processes and degrades system responsiveness.

Reactive programming enables the structuring of applications around data streams and the propagation of changes [5]. This paradigm, when implemented in a fully non-blocking environment, can lead to improved concurrency and more efficient use of system resources.

In reactive programming (Fig. 4), synchronous operations transform into asynchronous event streams [6]. For example, a database read doesn't block the thread while fetching data. Instead, it immediately returns a publisher that others can subscribe to. Subscribers process events as they occur and may generate further events, creating a chain of reactive interactions. This approach enhances concurrency and resource efficiency, allowing systems to handle multiple operations simultaneously without waiting for each to complete [7].

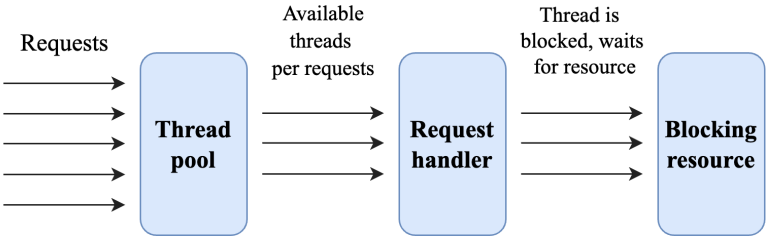

Fig. 3. Typical model of REST requests handling

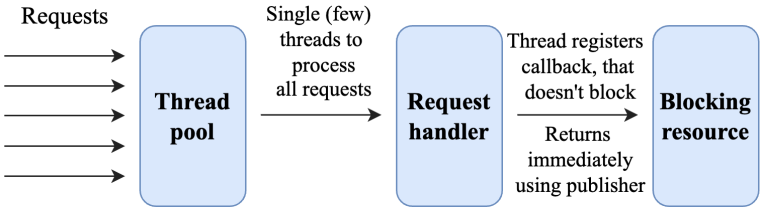

Fig. 4. Reactive model of REST requests handling

At a higher level of abstraction using business processes, this approach allows controlling data exchange events and signaling processes about changes in requests accordingly. This enables monitoring the stages of process execution, tracking states, understanding potential problem areas in case of errors, and helping to build more flexible metrics [8].

The proposed signaling implementation in Fig. 5 is based on sending «ACK» signals each time a specific stage of the process is completed. Such a signal may contain minimal information about the process (process ID), the execution stage (step ID), and the state of that stage (pending, active, blocked, etc.).

The chosen model for implementing asynchronous requests is based on the Event Loop (Fig. 6):

– *Request registration:* the request is placed in the event queue, from which events are processed using the event loop.

– *Single-Threaded Execution*: an event loop operates continuously within a single thread. Multiple event loops can run concurrently, typically one per available CPU core.

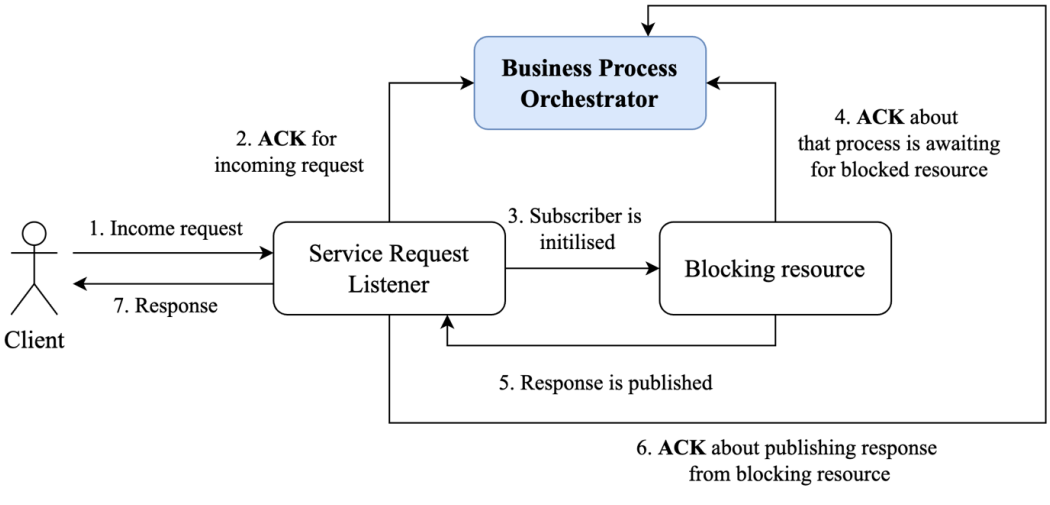

Fig. 5. Signaling stages of reactive requests execution

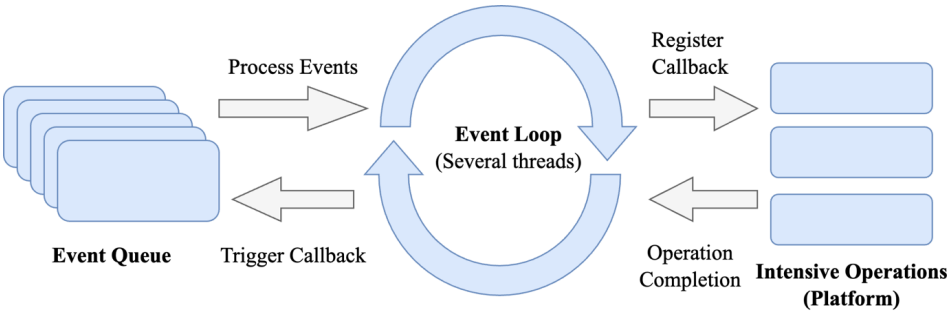

Fig. 6. Event loop model of asynchronous requests

– *Non-Blocking Event Processing*: the event loop sequentially processes events from an event queue. It registers callbacks with the underlying platform and returns immediately, without waiting for operations to complete.
– *Platform-Driven Completions*: the platform (e. g., operating system, runtime environment) manages long-running operations like database queries or external service calls. It notifies the event loop upon operation completion.
– *Callback Invocation*: when notified of an operation's completion, the event loop triggers the associated callback. This mechanism allows the system to process results and propagate them back to the original caller asynchronously.

This model enables efficient handling of concurrent operations without the complexity of traditional multi-threading, forming the foundation of reactive systems.

3. Results and Discussion

As an experiment, a business process was developed to estimate the losses of the Russian occupation army during the war from 2022 to 2024 in terms of personnel and equipment, summarizing the total amount. Due to a lack of precise data, approximate figures were interpolated using data from the Ukrainian Ministry of Finance website [9]. The goal of the experiment is to assess the direct load on the system depending on its architecture. The following conditions were set:

– although resource [9] provides losses in personnel on a daily basis (e. g., +1130 personnel as of June 1, 2024), these losses are simulated by submitting one request per person to increase the load. The same approach applies to losses in equipment such as aircraft and artillery systems;
– the immediate number of losses up to the current moment is evaluated, meaning the application consumes a data stream where data arrives instantly.

This experiment aims to test how well the system handles a high volume of real-time data inputs, simulating ongoing updates and calculations based on available but interpolated data. The implementation of service discovery is not within the scope of this article. The article supports two services: soldiers and tanks, with an example ontology depicted in Fig. 7.

Fig. 8 depicts the architecture of the application. As mentioned earlier, the idea is to create artificial load on the Business Process Orchestrator service. Its task is to generate a large number of processes that will be managed through synchronous and asynchronous requests.

The application was implemented in the Java programming language. To simulate a limited number of threads for each service, the *Executors.newFixedThreadPool* method was used, which creates a thread pool and provides access to process the next request once a thread becomes available [10].

The Task Price DB and Soldier Rank DB are blocking resources. When handling synchronous requests, the number of blocked threads was significant, posing a threat to system stability. In the case of asynchronous requests, the system was continuously ready to accept new threads, allowing for less thread time spent on processing requests and incoming data frames.

Fig. 9, 10 depict the number of active threads in the application under uniform load (approximately 55,000 requests per second). The visualization shows the maximum number of threads during the specified interval.

As can be seen, the asynchronous model with a maximum of 8 active threads allowed for handling a larger number of incoming requests. This was achieved by using a Subscriber to wait for the completion event of the blocking resource, thus not occupying space in the active threads.

Based on this, it can be concluded that asynchronous request processing based on reactive system architecture does not exhaust all available CPU resources, does not block the system, and is therefore ready for further loads.

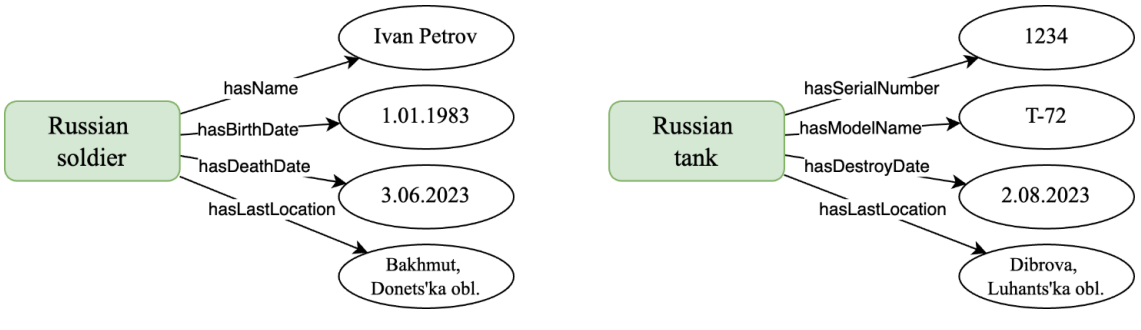

Fig. 7. Example ontology for Service Discovery

ISSN 2664-9969

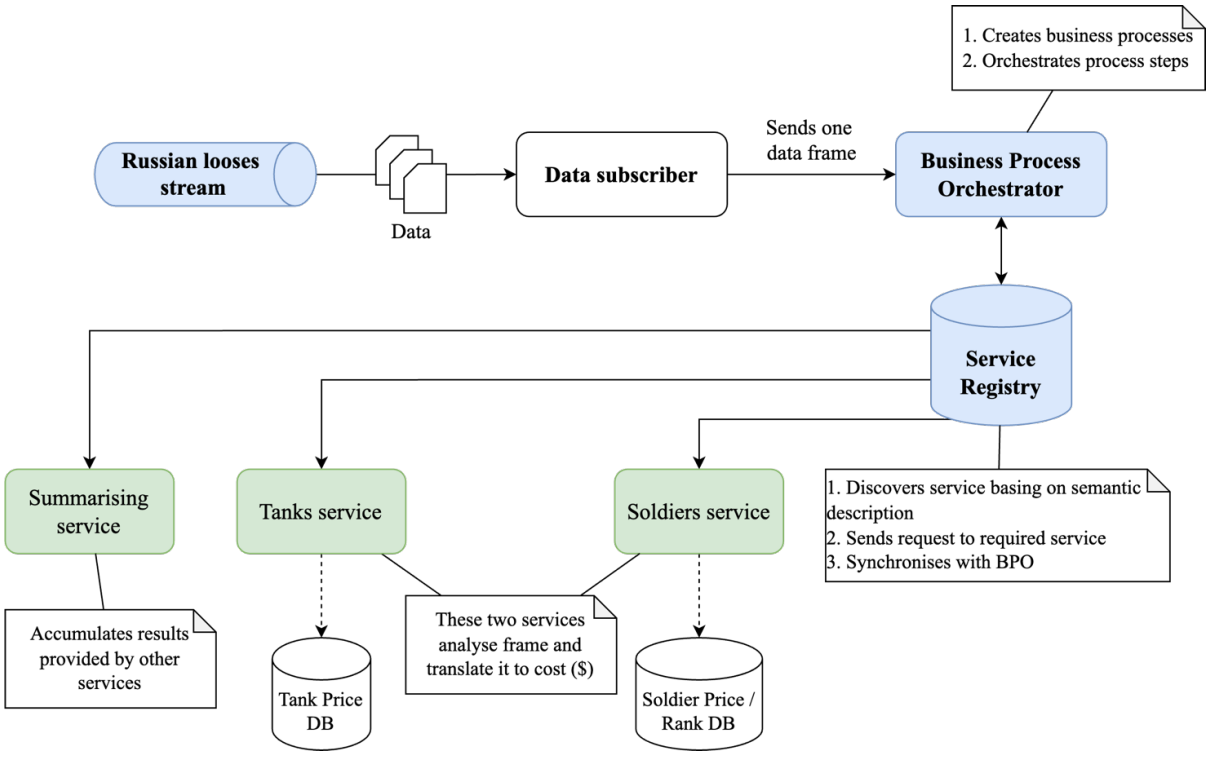

Fig. 8. Application architecture for calculating army damage costs

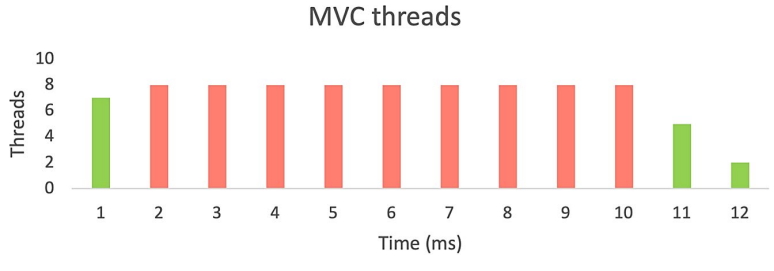

Fig. 9. Active threads during synchronous requests

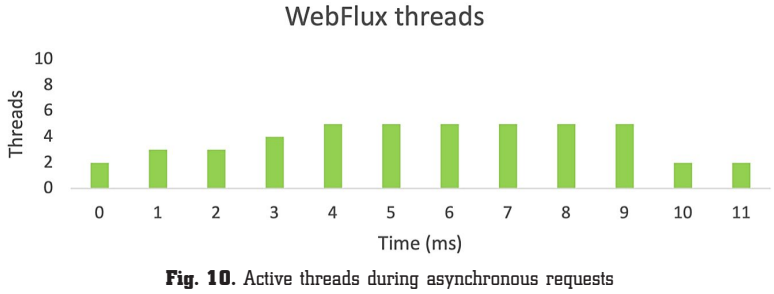

Fig. 10. Active threads during asynchronous requests

This is useful in systems with a continuous data flow and under conditions where the load is unstable. The proposed example of accounting for enemy losses is just a part of a large battle management system, where data on troop movements will be received in real-time, and under such conditions, the system must be highly reliable and resilient. It should be noted that the complexity of implementing this architecture is too high, therefore, for systems with small or linear loads, it is recommended to adhere to traditional synchronous interaction.

Further research will focus on implementing a reactive architecture in a network of services managed by semantic descriptions using OWL-S or WSDL. The reactive approach will make the service discovery operation non-blocking, which is an important optimization of the architecture as a whole.

4. Conclusions

The article explored request processing models using synchronous and asynchronous approaches with an Event Loop based on the WebFlux framework. Both methods were tested using an application that processes data from the Ministry of Finance website regarding the number of destroyed personnel and equipment of the Russian army, returning their cost. This was done using two services governed by semantic description and orchestrated by a dedicated orchestration service. Service discovery processes

were simplified due to the presence of only two services for orchestration, which was not the focus of this article.

The thread activity was compared between traditional request handling and reactive programming. Since the reactive programming architecture is less CPU-intensive, it showed 45 % less active threads, avoided system overload situations (where the thread count equals the maximum) comparing to traditional one (that had 8 milliseconds of downtime), and demonstrated better resilience.

Conflict of interest

The author declares that he has no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

The study was performed without financial support.

Data availability

Manuscript has no associated data.

Use of artificial intelligence

The author confirms that he did not use artificial intelligence technologies when creating the current work.

References

1. Lewis, J., Fowler, M. (2017). *Microservices: A definition of this new architectural term*. Martin Fowler. Available at: https://martinfowler.com/articles/microservices.html
2. Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR.
3. Peng, H., Shi, Z., Qiu, L. (2007). *Matching Request Profile and Service Profile for Semantic Web Service Discovery*. Available at: https://www.researchgate.net/publication/239813737_Matching_Request_Profile_and_Service_Profile_for_Semantic_Web_Service_Discovery
4. Deakin, T., Cook, R. (2018). Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools. *Journal of Systems and Software, 137,* 116–130.
5. Reactive Streams (2013). *Reactive Streams Initiative*. Available at: http://www.reactive-streams.org
6. Hohpe, G., Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.
7. Pardon, G., De Backer, R. (2011). Building Scalable Applications with Event-Driven Architecture. *Proceedings of the International Conference on Software Engineering*.
8. Hohpe, G., Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 736.
9. *Ministry of Finance of Ukraine* (2024). Available at: https://mof.gov.ua/uk/
10. Richards, M. (2006). *Pro Java EE 5 Performance Management and Optimization*.

Ihor Kasianchuk, *PhD Student, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: kasyk3@gmail.com, ORCID: https://orcid.org/0009-0000-2215-149X*