IRSTI 28.27.15 https://doi.org/10.26577/jpcsit20253103

**Mukhit Zhanuzakov[1,*], Gulnar Balakayeva[1], Paul Ezhilchelvan[2]**

[1]Al-Farabi Kazakh National University, Almaty, Kazakhstan
[2]Newcastle University, Newcastle upon Tyne, United Kingdom
*e-mail: zhanmuha01@gmail.com

# MODEL BASED SOLUTION FOR COMPUTING CHECKPOINTING INTERVAL FOR FAULT-TOLERANT ROLLBACK-RECOVERY IN ENTERPRISE SERVERS

**Abstract.** Recently, reliable information processing has become a relevant topic with the increase of digitalization. It is especially essential for enterprises that process huge amounts of data every day. These processes require stability and reliability as their interruption might lead to various security issues. In order to tackle this, there are fault-tolerance algorithms that are specifically designed to prevent or recover faults. This paper focuses on developing a heuristic solution to find optimal checkpointing interval for rollback-recovery fault-tolerance algorithm. The authors propose a model based solution that utilizes CPU capabilities to determine how often checkpointing should be taken for reliable information processing. This research provides statistics and predictions from major research organizations, highlighting the relevance of the topic. Paper also reviews related work devoted to this area of research, providing comparisons and an overall analysis. The results of the work show that the proposed calculation method introduces minimal performance overhead, averaging 0.04 seconds to the average service time, while maintaining fault tolerance of the process. Authors indicate that this solution is suitable for proof-of-concept systems to efficiently determine optimal interval for checkpointing.

**Key words**: checkpointing, fault-tolerance, enterprise, heuristic solution, application, rollback-recovery.

## 1. Introduction

Currently, reliable information processing is becoming relevant with the increase of digital information. The increase in the volume of digital information was due to the exponential growth of data mining and the widespread introduction of digital technologies. In 2024, more than 463 exabytes ($10^{18}$ bytes) of data are estimated to be generated worldwide every day.

Almost all information is processed on servers. According to researchers [1], the enterprise server market is projected to be USD 87.96 billion in 2024 and USD 129.42 billion by 2029, an average increase of 8.03% during the forecast period (2024-2029). Figure 1 shows this data in a graph format for clear understanding.

Enterprise servers are crucial indicators of a country's level of digitalization. Figure 2 displays growth of enterprise server market by regions. We can see that most developing countries in Asia and Australia show high increase in enterprise server market including Kazakhstan. This suggests that the relevance and importance of the reliable enterprise servers in these regions will increase as well.

Enterprise servers can fail while processing any information, leading to data loss and disruption of the entire system. Other researchers from Uptime Institute [2] analyzed outage of servers from 2021 to 2023. An outage is a period of time when a computer system, service, network, or infrastructure is unavailable or not functioning properly. Figure 3 shows that according to their research, average cost per outage increased from 100 000 USD to 200 000 USD during the examined period. Therefore, it is very important for enterprise systems to have methods and techniques that increase the system's resistance to failures.

 

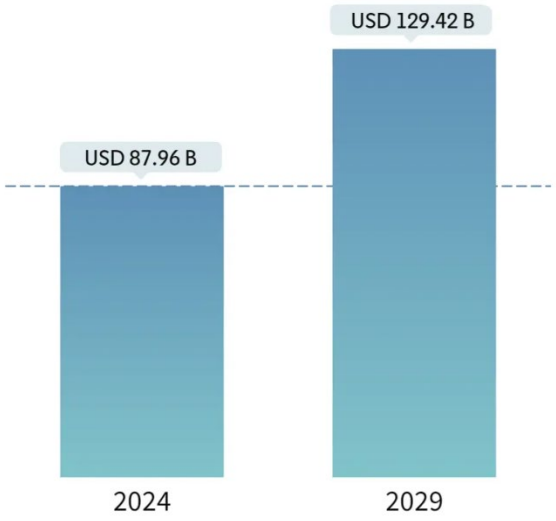


**Figure 1 –** Enterprise Server Market size predictions

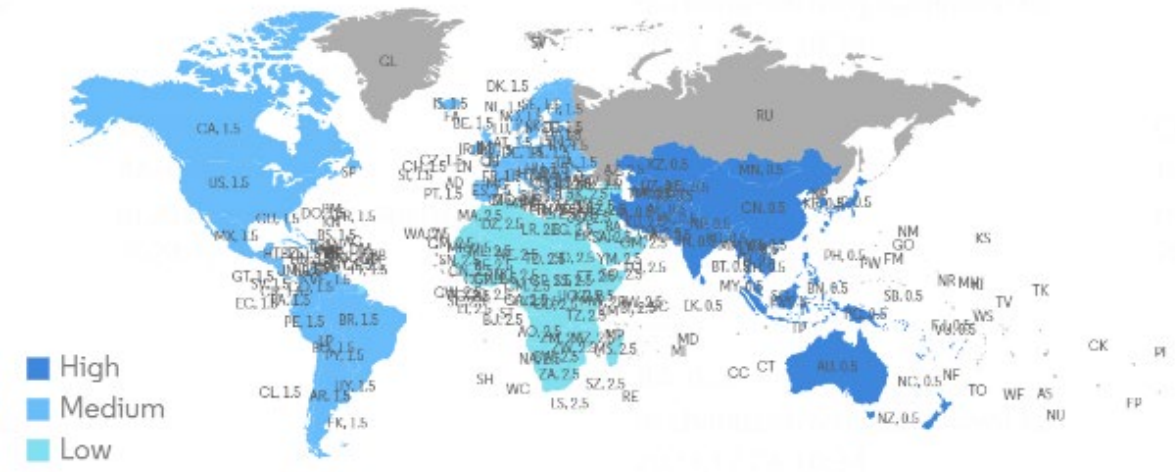


**Figure 2 –** Enterprise server market growth rate by region

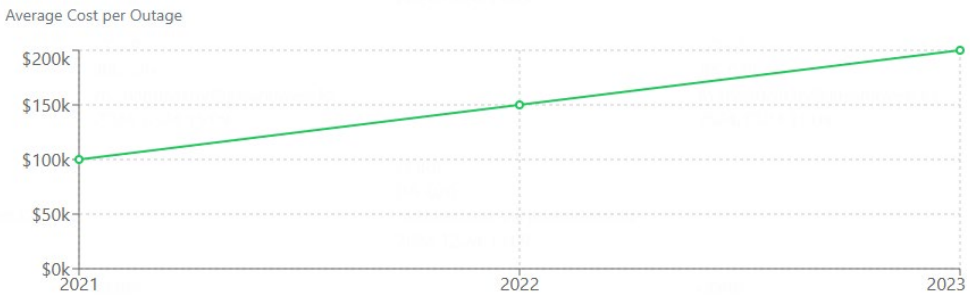


**Figure 3** – Average cost per server outage

A popular solution for handling faults and malfunctions is the integration of fault-tolerance algorithms. Fault tolerance is the ability to eliminate unexpected malfunctions that occur during the operation of the system [3]. Using fault tolerance algorithms, one can achieve reliable operation of the entire system. This is especially important for enterprise information systems that are highly dependent on the operation of servers and perform high computing processes that require high availability.

Rollback-recovery algorithm is one of the fault-tolerance algorithms that allows to recover the server/system to a consistent state using checkpointed states called "checkpoints". These checkpoints are stored in stable storage [3]. One of the hyperparameters of the algorithm is checkpointing interval, that is how often checkpointing should be taken. In this work, we aim to compute the optimal interval between each checkpoint while maintaining minimal impact on performance.

## 2. Related work

In this section, we gathered and reviewed different works related to fault-tolerance algorithms with checkpointing nature.

Yishu et al. [4] addresses the challenge of protecting iterative applications from fail-stop errors. Authors achieve this by developing an optimal checkpointing strategy. The idea of the works is to execute a series of iterations, each comprising multiple tasks with varying execution times and checkpoint costs. The authors propose a dynamic programming algorithm to compute the optimal checkpointing pattern. The results demonstrate that a globally periodic strategy can outperform traditional methods, such as checkpointing after each task or at the end of each iteration.

Research by Jayasekara et al. [5], discusses the inefficiencies of traditional single level checkpointing in stream processing platforms. The authors propose a theoretical framework for a multi-level periodic checkpointing system, where at each checkpoint interval, a specific level is selected probabilistically. They derive optimal checkpoint intervals and associated probabilities by considering factors such as failure rates, checkpoint costs, restart costs, and potential failures during recovery at each level.

Another research by Hérault et al. [6], outlines the challenge of input/output (I/O) contention in high-performance computing (HPC) environments, where multiple applications compete for limited I/O bandwidth. The authors propose a cooperative scheduling policy that optimizes the overall performance of concurrently executing checkpoint/restart-based applications sharing I/O resources. They develop a theoretical model and derive necessary constraints to minimize global waste on the platform. Their findings indicate that combining optimal checkpointing periods with I/O scheduling strategies can significantly improve overall application performance.

Research by Ezhilchelvan el al. [7] examines a server subject to random breakdowns and repairs, providing services to incoming jobs with highly variable lengths. The study focuses on implementing a checkpointing policy aimed at mitigating potentially lengthy recovery periods by periodically backing up the current state. The authors analyze a general queuing model that incorporates breakdowns, repairs, backups, and recoveries to determine optimal checkpointing intervals that enhance performance. They derive exact solutions under both Markovian and non-Markovian assumptions. Through numerical experiments, the paper illustrates conditions under which checkpoints are beneficial and quantifies the achievable advantages in such scenarios.

Overall works [4] – [7] are analyzed and compared with our work. The results of the comparisons are shown in Table 1.

**Table 1 –** Comparison of related works

| Work | Overview | Advantages | Disadvantages |
|---|---|---|---|
| Yishu et al. | introduce a mathematically optimized checkpointing strategy for iterative applications that experience fail-stop errors. | theoretical optimal strategy, strong mathematical solution | assumes controlled conditions, less practical application |
| Jayasekara et al. | implements a multi-level checkpointing approach for exascale systems, optimizing failure recovery through adaptive interval selection. | advanced exascale optimization, multi-level fault tolerance | limited to exascale systems, complex to implement |
| Hérault et al. | proposes a scheduling algorithm that provides an optimal checkpoint period to maximize overall platform throughput. | efficient scheduling algorithm, considers bandwidth constraints | limited to shared HPC environments, complex deployment |
| Ezhilchelvan et al. | introduces optimal checkpointing strategies for systems with task variability and random failures | strong theoretical model based on queuing theory | requires predefined failure and repair rates |
| Our work | provides a heuristic solution for computing optimal checkpointing interval | practical approach, good for proof-of-concept systems | lack of theoretical optimization |

Overall, the reviewed authors have provided good theoretical solutions for checkpointing-based approaches. The main difference between our work and these works is that our solution is more practical with a focus on real-life applications.

## 3. Methodology

To achieve consistency of data processed during process execution, we use a rollback-recovery algorithm. This algorithm mainly works using “checkpoints”.

The idea of a checkpointing is to periodically store the state of a computation in stable storage [8] – [10]. Checkpoints are established while the process is running without failure. This saved state can be used to restart the process, rather than completely restarting the process, which would involve lengthy, repetitive processing and repeated output.

**Problem identification**

The main question here will be to find the optimal interval for taking these control points. Because taking checkpoints too often will increase overhead and response time during uptime periods and will require larger storage capacity. On the other hand, if the checkpointing is performed less frequently, the rollback duration must be long, which subsequently increases the duration of service unavailability after a failure. Thus, the effectiveness of the system's fault tolerance will decrease.

The ideal way to calculate the optimal check interval is to estimate it by simulating the system along with all relevant parameters, which include the request arrival rate, server processing speed, server failure probability, etc. Such modeling work has been extensively done by many authors [11] – [16]. This ideal way of determining the optimal interval is not feasible when we are prototyping a proof-of-concept system because the corresponding values for all model parameters are unknown. Therefore, we decided to estimate the checkpoint-ting interval heuristically, as described below.

**Heuristic solution**

Intuitively, the frequency of checkpointing will depend on the average execution of the process as well as on the frequency of the CPU. This is because: more processor power means shorter execution times. This is derived from fundamental computer architecture and performance modeling equations by John L. Hennessy et. al. [17].

**Equation 1.**

The equation by John L. Hennessy et. al. describes how total execution time is inversely proportional to the CPU frequency, if we assume the number of instructions and cycles per instruction (CPI) remain constant.

$$T = \frac{I*CPI}{f} \quad (1)$$

Where:
T = total execution time
I = number of instructions
CPI = cycles per instruction
f = CPU frequency (gigahertz)

This was based on fundamental principles established by computer architects and engineers

over decades, including Amdahl's Law [18] and pipelining concepts in modern CPUs. Using this model, we propose a solution for calculating checkpointing interval that is based on CPU frequency and mean execution time.

**Definition 1**. $T_c$ is the time in seconds between two checkpoints during a fail-free computational process.

**Definition 2.** The mean execution time $T_p$ refers to the average time required to complete a given computational process on a single-threaded CPU. It is calculated as the difference between the total area of the cumulative distribution function (CDF) and the area under the CDF.

**Definition 3.** CPU frequency – f defines how many cycles per second the CPU can execute in gigahertz. This parameter varies depending on CPU of each individual server.

**Equation 2.**

We use the following Equation (1) to determine the checkpointing interval ($T_c$):

$$T_c = \frac{T_p}{f} \tag{2}$$

*Where:*

$T_p$= mean execution time on a single threaded CPU (seconds)

f= CPU frequency (gigahertz)

The higher CPU frequency the lower checkpointing interval value. Subsequently, lower checkpointing interval means more checkpoints can be established during processing with minimal impact on performance.

**Equation 3.**

The area under the CDF is computed using the Trapezoidal Rule:

$$\text{area under CDF} = \sum_{i=1}^{N-1} \frac{(h_i + h_{i+1})}{2} * (x_{i+1} - x_i) \tag{3}$$

*Where:*

$h_i$ = i-th height of trapezoid

$x_i$ = i-th width of trapezoid

N = total number of trapezoids

This method ensures a fast and accurate approximation of execution time probabilities over time. CDF will also be used to evaluate the effectiveness of checkpointing in the next sections.

**Equation 4.**

$T_p$ is calculated using the following equation:

$$T_p = \text{total area CDF} - \text{area under CDF} \tag{4}$$

*Where:*

Total area CDF represents the longest execution time.

The area under the CDF is the result of Equation (3).

The above Equations (2)-(4) are coded in python and can be used to calculate checkpointing interval (see Listing (1)).

**Listing 1:** Code for checkpointing interval calculation

```
import pandas as pd

response_times = […]

def calc_probabilities(arr):
 df = pd.DataFrame(arr)
 df = df.rank(method='max') / len(df)
 return df[0].tolist()

def calc_area(h1, h2, x1, x2):
 delta_x = x2 - x1
 avg_h = (h1 + h2) / 2
 return avg_h * delta_x

def find_area_under_cdf(x_arr, y_arr):
 res_area = 0
 for i in range(len(x_arr)):
 if (i == len(x_arr) - 2):
 break
 res_area += calc_area(y_arr[i], y_arr[i + 1], x_arr[i], x_arr[i + 1])
 return res_area
```

```
22:
23: def compute_Tp(total_area_CDF, area_CDF):
24:  return total_area_CDF - area_CDF
25:
26:
27: def compute_checkpoint_interval(T_p, f):
28:  return T_p / f
29:
30:
31: probabilities = calc_probabilities(response_times)
32: area_CDF = find_area_under_cdf(response_times, probabilities)
33: total_area_CDF = response_times[len(response_times) - 1]
34: T_p = compute_Tp(total_area_CDF, area_CDF)
35: optimal_checkpointing_interval = compute_checkpoint_interval(T_p, 2.7)
```

Overall, this solution is aimed at providing basic checkpointing implementation for applications that are less prone to long term breaks as the solution is not theoretically optimal. However, this should be enough for small to medium-sized enterprises with limited technical capabilities and specialists.

## 4. Results and discussions

To validate the effectiveness of purposed method, we run simulations of process executions with and without checkpointing to see how much overhead it adds to the processing time. As a testing environment we have two servers with 2.70GHz CPU each. As a testing process we have selected is document template generation process. It includes .word file generation with text input in the contents of the file and converting it to .pdf file that contains QR code of digital signature.

To compute average execution time on a single threaded CPU, we created script for the proposed process in Python. Then, we run the program for 400 time to find $T_p$. Using the results, we plotted CDF (Cumulative Distribution Function) of execution times (see Figure 4). After this, we calculated mean execution time using Equation 4. The result was 0.52 seconds. After that we used Equation 2 to calculate checkpointing interval, which resulted in 0.09 seconds.

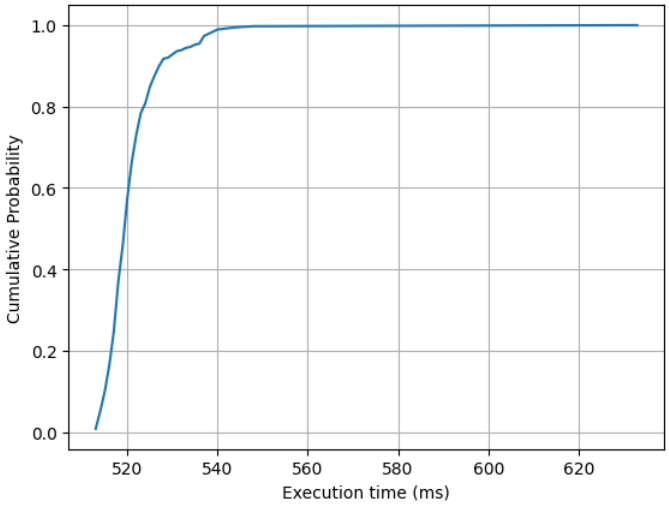


**Figure 4 –** CDF of execution times

After calculating the checkpointing interval, we can start utilizing it for recovery. To test how checkpointing can impact performance when server breaks down, we implemented testing scenario that will crash our application at random moment during execution and perform recovery. In the first case, we do not utilize checkpointing. In the second case we use the checkpointing using our calculation

model. In both cases, the process is recovered in another server using rollback-recovery algorithm.

Figure 5 shows CDF of execution times with and without checkpointing when simulated cash occurs. The total number of repetitions was 400 for each case. As a result, when checkpointing is used most processes are completed under 500 milliseconds. This compares with almost 950 milliseconds in case of no checkpointing.

To show how much overhead is added after enabling checkpointing, we run the testing process again with checkpointing in a crash free environment. Figure 6 displays CDF of execution times with and without checkpointing.

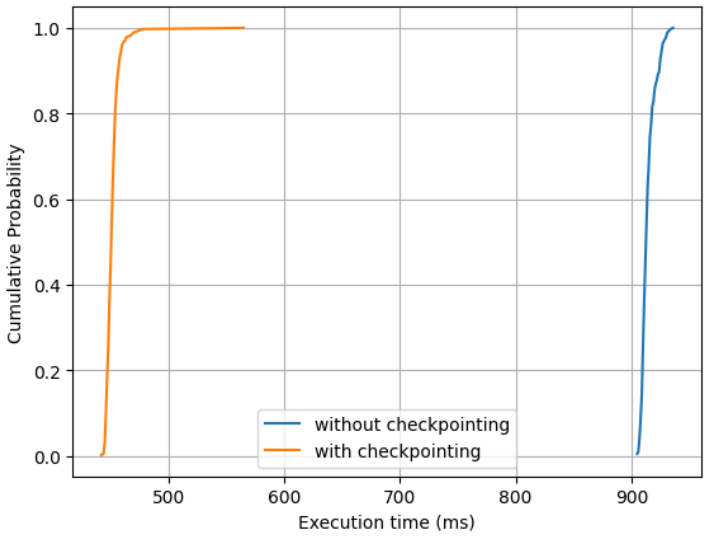


**Figure 5 –** CDF of execution times with and without checkpointing in a crash simulation environment

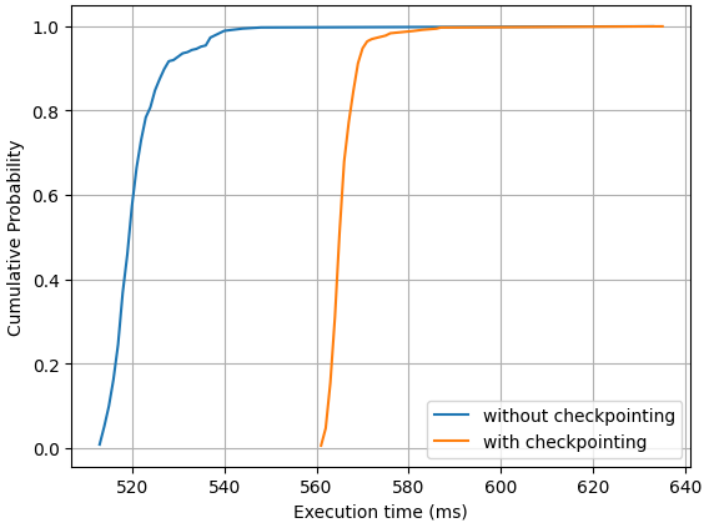


**Figure 6 –** CDF of execution times with and without checkpointing in a crash free environment

As seen in Figure 6, checkpointing adds small overhead to the execution duration, increasing mean execution time by only 0.04 seconds. Note that without checkpointing, upon failure, process would be restarted from the very start and the service time of the process would be a lot higher as seen in Figure 5.

## 5. Conclusion and future work

The research conducted in this article is very relevant and in demand at the present time. The huge increase in the load on servers leads to unwanted failures in the operation of corporate servers, which significantly interferes with the successful and reliable operation of enterprises

In this work we carried out research on developing a heuristic solution for calculating optimal checkpointing interval for well-known rollback-recovery method. We used CPUs capability and mean execution time of process to calculate the checkpointing interval. During our research, we found out that the method greatly improves the performance of applications in a failure scenario, while adding tiny overhead to the performance (0.04 seconds) in a crash-free scenario. By applying this method, small/medium enterprises can easily find the checkpoint interval and use the rollback-recovery method to ensure reliability.

Currently, this solution is experimentally proven to be effective in an environment that is less prone to errors. The limitation of the work is the absence of theoretical optimization for long-term applications. This can be future research topic for our team.

Another future direction of this work can be the performance evaluation of the proposed solution in different workloads such as multi-threaded CPUs and distributed systems.

## Funding

This research received no external funding

## Author Contributions

Conceptualization, M.Z. and G.B.; Methodology, M.Z.; Software, M.Z.; Validation, G.B. and P.E.; Formal Analysis, G.B. and P.E.; Investigation, G.B. and M.Z.; Resources, G.B. and M.Z.; Data Curation, M.Z.; Writing – M.Z.; Writing – Review & Editing, G.B. and P.E.; Visualization, M.Z.; Supervision, G.B. and P.E.; Project Administration, G.B.; Funding Acquisition, M.Z. and G.B..

## Conflicts of Interest

The authors declare no conflict of interest

## References

1. Enterprise Server Market – Share, Trends & Analysis (Mordor Intelligence). Available: https://www.mordorintelligence.com/industry-reports/enterprise-servers-market. Accessed: Feb. 2, 2025.
2. Uptime Institute, "Annual Outage Analysis 2023." Available: https://datacenter.uptimeinstitute.com. Accessed: Feb. 2, 2025.
3. R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, Jan. 1987, doi: 10.1109/TSE.1987.232562.
4. Y. Du, L. Marchal, G. Pallez, and Y. Robert, "Optimal checkpointing strategies for iterative applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 507-522, 2022.
5. S. Jayasekara, A. Harwood, and S. Karunasekera, "Optimal multi-level interval-based checkpointing for exascale stream processing systems," *arXiv preprint*, arXiv:1912.07162, 2019.
6. T. Hérault, Y. Robert, A. Bouteiller, D. Arnold, K. Ferreira, G. Bosilca, and J. Dongarra, "Optimal cooperative checkpointing for shared high-performance computing platforms," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vancouver, BC, Canada, 2018, pp. 803-812, doi: 10.1109/IPDPSW.2018.00127.
7. P. Ezhilchelvan and I. Mitrani, "Checkpointing models for tasks of different types," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 9, no. 3, Article 10, pp. 1-23, Sep. 2024, doi: 10.1145/3658667.
8. P. K. Jaggi and A. K. Singh, "Rollback recovery with low overhead for fault tolerance in mobile ad hoc networks," *Journal of King Saud University – Computer and Information Sciences*, vol. 27, no. 4, pp. 402-415, 2015.
9. S. Kumar T, M. HS, S. S. Mustapha, P. Gupta, and R. P. Tripathi, "Intelligent fault-tolerant mechanism for data centers of cloud infrastructure," *Mathematical Problems in Engineering*, vol. 2022, no. 1, p. 2379643, 2022.
10. M. Kirti, A. K. Maurya, and R. S. Yadav, "Fault-tolerance approaches for distributed and cloud computing environments: A systematic review, taxonomy and future directions," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 13, p. e8081, 2024.

11. X. Wang, C. Zhang, J. Fang, R. Zhang, W. Qian, and A. Zhou, "A comprehensive study on fault tolerance in stream processing systems," *Frontiers of Computer Science*, vol. 16, pp. 1-18, 2022.

12. S. S. Nabavi and H. Farbeh, "A fault-tolerant resource locking protocol for multiprocessor real-time systems," *Microelectronics Journal*, vol. 137, p. 105809, 2023.

13. A. U. Rehman, R. L. Aguiar, and J. P. Barraca, "Fault-tolerance in the scope of cloud computing," *IEEE Access*, vol. 10, pp. 63422-63441, 2022.

14. S. Khan, I. A. Shah, K. Aurangzeb, S. Ahmad, J. A. Khan, and M. S. Anwar, "Energy-efficient task scheduling using fault tolerance technique for IoT applications in fog computing environment," *IEEE Internet of Things Journal*, vol. 11, no. 24, pp. 39009-39019, Dec. 15, 2024, doi: 10.1109/JIOT.2024.3403003.

15. P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *Journal of King Saud University – Computer and Information Sciences*, vol. 33, no. 10, pp. 1159-1176, 2021, doi: 10.1016/j.jksuci.2018.09.021.

16. V. P. M and P. D, "Reactive fault tolerance aware workflow scheduling technique for cloud computing using teaching learning optimization algorithm," *2023 Fifth International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Erode, India, 2023, pp. 1-7, doi: 10.1109/ICECCT56650.2023.10179823.

17. J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Elsevier, 2011. ISBN: 9780123838735.

18. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in Proc. AFIPS Spring Joint Comput. Conf., New York, NY, USA, Apr. 1967, pp. 483–485. doi: 10.1145/1465482.1465560.

***Information About Authors:***

*Mukhit Zhanuzakov (corresponding author) – PhD student at the Al-Farabi Kazakh National University, Faculty of Information Technology (Almaty, Kazakhstan, e-mail: zhanmuha01@gmail.com).*

*Gulnar Balakayeva – Professor, Doctor of Physical and Mathematical Sciences at the Al-Farabi Kazakh National University, Faculty of Information Technology (Almaty, Kazakhstan, e-mail: gulnardtsa@gmail.com).*

*Paul Ezhilchelvan – PhD at the Newcastle University, School of Computing (Newcastle upon Tyne, United Kingdom, e-mail: paul.ezhilchelvan@newcastle.ac.uk).*