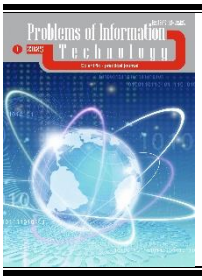

Available online at www.jpit.az

16 (1)
2025

# Development of a conceptual model for ensuring fault tolerance of software systems

Tamilla Bayramova

Institute of Information Technology, B. Vahabzade str., 9A, AZ1141 Baku, Azerbaijan

tamilla@iit.science.az

https://orcid.org/0000-0002-8377-3572

ARTICLE INFO

**Keywords:**

Software system
Software reliability
Fault tolerant
Self-adaptive system
Conceptual model

ABSTRACT

Digital transformation is a comprehensive process of integrating information technology into all areas of human activity. Almost all aspects of our lives, from personal communications to global economic processes, are increasingly integrated with digital technologies. Banking, manufacturing, healthcare, education, and transportation are all increasingly using software to improve efficiency, optimize processes, and provide new services. Modern applications are becoming increasingly complex, consisting of many interconnected components. This increases the likelihood of errors and failures. The relevance of research in the field of software reliability is steadily growing.This article is devoted to the study of the differences and relationships between information and software systems, as well as an in-depth analysis of the key concepts of software reliability and fault tolerance. The main approaches and strategies for ensuring fault tolerance are considered, including redundancy, backup, monitoring, duplication, load balancing, microservices, backup, prediction and detection of errors, as well as their practical application. The aim of the study is to define the principles and methods of self-healing software and to analyze the risks associated with automatic response to failures. To solve the problem of ensuring fault tolerance in dynamic and complex software systems, a conceptual model for designing reliable, self-learning and self-adaptive systems is proposed.

## 1. Introduction

Software systems have become an integral part of our daily lives, driving innovation, efficiency, and transformation, contributing to economic growth, improving social well-being, and solving global problems. These systems have permeated every aspect of our lives, from smartphones to businesses, healthcare, education, government, and critical infrastructure. With the ubiquity of software systems in critical structures and the exponential growth in complexity of modern systems, achieving reliability has become paramount in the field of software engineering. Recognizing that errors will inevitably occur despite efforts to eliminate them is a fundamental principle of reliability. Despite the evolution of technology, the problem of managing faults and failures in software systems remains relevant. The evolution of computing systems has transformed the nature of faults, the complexity of systems, the services they provide, and the nature of their use in society. However, the need to manage heterogeneous fault streams, including component failures, environmental degradation, component mismatches, human errors, cyber-attacks, and software defects, remains a central challenge in software research and engineering (Febrero, Calero, & Moraga, 2016).

*Received 9 October 2024, Received in revised form 10 December 2024, Accepted 24 December 2024*
*http://doi.org/10.25045/jpit.v16.i1.05*

Achieving high software reliability remains a fundamental challenge in software systems development. Researchers use a comprehensive approach that includes preventive measures and error-removal methods. An example of increased attention to software reliability is the Microsoft Trustworthy Computing initiative, launched in the early 2000s. Bill Gates, recognizing society's growing dependence on software, urged his employees to focus on developing high-quality code that would be accessible, reliable, and secure, even if this meant refraining from adding new features (Conti, Schunter & Askoxylakis, 2015).

A variety of architectural solutions, precise description of software requirements using mathematical methods, rigorous design methodologies, program verification, management of the software development team, as well as tools for monitoring compliance with programming standards and code reuse can reduce the likelihood of errors during the development phase. However, even the most advanced preventive measures do not guarantee the complete absence of defects (Nafreen, Bhattacharya & Fiondella, 2020). Bug elimination methods aim to detect and fix defects that remain in the system after development. They include manual code review and automated approaches such as testing and formal verification. Despite the increasing costs of testing and the complexity of the applied methods, the complete elimination of errors remains unattainable due to the complexity of systems and the difficulty of detecting rare or specific defects that cannot be detected using standard scenarios (Kazimov, Bayramova & Malikova, 2021).

Due to the complexity of modern software systems, it is almost impossible to guarantee their absolute error-free operation using testing alone. Even the most thorough and comprehensive tests cannot cover all possible scenarios and operating conditions. Therefore, in applications where reliability is a critical requirement, fault tolerance becomes an integral part of the architecture. This article discusses the differences and relationships between information systems (IS) and software systems (SS), briefly presents the main concepts of reliability and fault tolerance, and considers in detail the key approaches and strategies developed to ensure software fault tolerance, as well as their practical application. A conceptual model for ensuring fault tolerance of software systems is proposed. The purpose of the study is to define the principles and methods of self-healing of software systems and automatic response to failures. The following are used to ensure fault tolerance: redundancy, reservation, monitoring, duplication, load balancing, microservices, backup, methods of prediction and error detection.

## 2. Information system vs software system

The terms IS and SS are often used interchangeably, but they represent fundamentally different concepts in computing. This paper examines their architectures, components, and roles in modern technologies, highlighting how software systems function as subsets of broader information systems. Key differences in scope, dependencies, and applications are analyzed.

With the development of digital technologies, the concepts of information system and software system are used as synonyms, which is a terminological error. There is a significant difference between them. ISO/IEC 24765 (2017) is a standard glossary defining terms used in software engineering. It is based on IEEE 610.12-1990, ISO/IEC 15288:2008 and ISO/IEC 12207:2008, and the ISO/IEC 2382 vocabulary. According to this standard:

- An information system is a system consisting of equipment, network infrastructure, software systems, human and financial resources that ensure the collection, processing and distribution of information.
- A software system is an integral part of an information system and consists of software components designed to perform one or more specific functions, configuration files and a set of documents that include information about the architecture, use and testing of the system.

Information systems typically consist of the following components: hardware, software systems, databases, networks and human resources (Fig.1.). It can operate independently or as part of an IS (Sommerville, 2017).

Software systems are divided into the following groups:

- Software-embedded systems, also called real-time systems or sociotechnical systems;
- Software-intensive systems;
- Computing-intensive systems.

Software-embedded systems are specialized software systems that are built into larger devices or systems to perform specific tasks.

Unlike general-purpose computers, they are designed to perform highly specialized functions and often operate in real time. For example, the

engine management system in a car or the temperature control system in a refrigerator, production line control systems, robot control (Garousi et al., 2018).

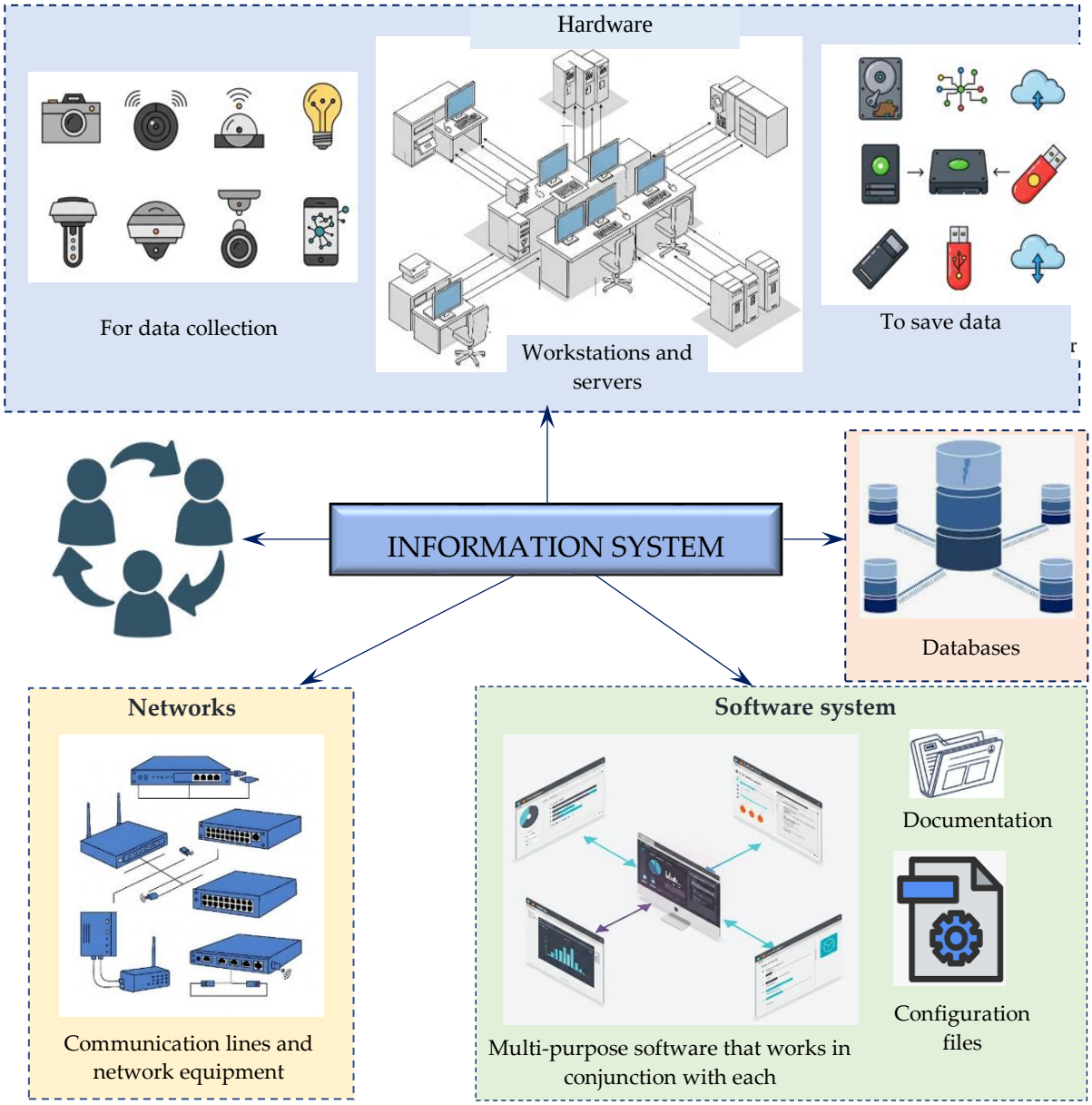


**Fig. 1.** Information system vs software system

Software-intensive systems are systems in which software plays a key role in their functioning, development and evolution. In simple terms, these are systems where software is a critical component that determines their behavior and capabilities. For example, flight control systems, radiation therapy machines, online banking systems, infrastructure management systems, etc. Software-intensive systems require special attention to software development, testing, and quality assurance, since errors in such systems can have serious consequences.

Computing-intensive systems are systems that require significant computing resources to perform their tasks. These systems are characterized by large amounts of data, complex algorithms, and calculations. Examples include climate modeling systems, genome sequencing, and social network analysis. Computing-intensive systems play an important role in many areas of science, engineering, and business, allowing them to solve complex problems and process large amounts of data.

60–90% of errors in these systems are due to software failures (Bernstein, 2003). The study and development of complex software systems is an

important area in software and systems engineering. The software system concept is used in the design and analysis of large and complex software projects. The ACM Software System Award is a prestigious award presented by the Association for Computing Machinery (ACM) for the development of software systems that have had significant impact. The award recognizes systems that have contributed concepts and have achieved widespread commercial acceptance. The award is financially supported by IBM. The Journal of Systems and Software (published by Elsevier) is devoted to the topic of software systems. Software systems are an active area of research.

Applying systems theory principles to software development means that software systems are viewed as complex, interconnected sets of components rather than as separate, isolated parts. This approach allows developers to better understand and manage the complexity of software projects. Key aspects of applying systems theory to software development include: systems analysis, design, complexity and configuration management, feedback and iteration, and system integrity.

## 3. Software system reliability

Software reliability, according to ISO/IEC 25010 (2023), is characterized by the ability of a system, product or component to maintain functionality under specified conditions for a certain period of time. The key aspects of reliability are: maturity, availability, fault tolerance and recoverability.

Software system maturity is compliance with reliability requirements during normal operation. Software system maturity is an important criterion that helps to assess its reliability and readiness for use in various conditions. Mature software is characterized by high stability due to repeated testing and error correction, which minimizes the number of critical errors and vulnerabilities, ensuring more reliable operation. Mature software complies with established standards and best practices, which also contributes to its reliability.

Software system availability determines how functional a system, product or component is and how ready it is to be used when needed. Software availability is determined by the time during which it functions without failures. High availability means that the system operates with virtually no interruptions. Software availability is one of the key reliability criteria, as it determines the ability of a system to provide its functions to users at the right time. Recoverability of software systems is the ability of a system, product, or component to restore data or the required operating mode after failures or errors. It determines how quickly and efficiently the system can return to normal operation after a failure. Using automated tools and scripts to simplify and speed up the recovery process, which reduces the likelihood of errors and speeds up recovery time. To minimize the impact of residual errors in software code, researchers have developed methods to prevent failures, known as fault tolerance. Software fault tolerance refers to the ability of software systems to continue to operate correctly even in the presence of failures or errors (Bayramova, Malikova, 2024).

Fault tolerance is a critical but challenging task in software systems due to their dynamic infrastructure and complex interrelations. To overcome these challenges, we present a conceptual model for designing robust self-learning self-adaptive systems.

## 4. Releated work

The study (de Souza et al., 2025) presents a systematic literature review on fault tolerance methods applied in Smart City applications and classifies possible failure modes. It is the first review to provide a comprehensive view of the research field and point to future directions. Smart City applications are at high risk of failure due to their context awareness, adaptability, distribution, and heterogeneity. Fault tolerance mechanisms are needed to ensure the reliability of these systems. The authors analyzed 43 key studies, performing a primary classification by study type, contribution, application domains, and system architecture. Then, they performed an in-depth analysis of the selected studies, classifying them by types of fault tolerance methods and failure modes. In the paper (Temene et al., 2024), a Mobile Fault Tolerant (MobileFT) architecture is proposed that incorporates fault detection and recovery. The Internet of Things (IoT) and Wireless Sensor Networks (WSN) are prone to failures due to node failures, security attacks, and connection breakdowns. To ensure continuous network functionality and performance despite these challenges, the use of mobile nodes is proposed. In this paper, a Fault Tolerant Node Placement Algorithm (FTNPA) is presented that uses mobile nodes to recover from network failures. Two variants of the FTNPA algorithm are developed: decentralized and centralized. The decentralized

variant uses neighbor detection and local recovery by placing a mobile node in the affected area. The centralized variant uses sink detection and creation of alternative paths for mobile nodes to reach the destination node. The simulation results confirm the effectiveness of the proposed algorithms in fault detection and recovery in IoT and WSN networks. The paper (Kumari & Kaur, 2021) reviews the existing approaches and techniques for ensuring fault tolerance in cloud environments and discusses the unsolved problems and promising research directions. The paper discusses various techniques that can be applied to address fault tolerance issues in the cloud.

Khadse & Karmore (2016) proposed a new approach to ensure the safety of automated vehicles based on the diagnosis and recovery of hardware faults. To improve the reliability of the system, cryptographic authentication and state estimation techniques are used to detect faults in actuators and sensors. The proposed system, applied to an anti-lock braking system (ABS), provides "intelligence" and fault tolerance, which helps prevent accidents and injuries. de Lemos (2009) argues that architectural abstractions significantly improve the efficiency of developing fault-tolerant software systems. The choice of abstraction depends on the failure model and the available resources. These abstractions, implemented as components and connectors, allow the formal definition of structural and behavioral properties, which automates the process of verification and validation of the architecture. In this paper, two architectural abstractions are discussed in detail: the Idealized Fault Tolerant Element (iFTE) based on exception handling and the Homing on Failure Element (HoFE) with fail-on-failure semantics. Using these abstractions simplifies the analysis of error propagation, detection, and handling, as well as fault handling. In the paper (Armoush, Salewski & Kowalewski, 2009), an innovative hybrid fault tolerance method is proposed that combines a recovery unit with a redundant voting mechanism to improve the reliability of systems using a recovery unit with limited acceptance test capabilities. The key element of the method is storing copies of the results of each program version in the cache as backup data. In the case where the recovery unit fails to correctly determine the correct result due to insufficient acceptance test performance, the stored values are used as input to the voting algorithm, which ensures a reliable and correct result. To demonstrate the advantages of the new method, Monte Carlo simulations were performed, which showed a significant increase in the reliability of the system and a decrease in its dependence on the quality of the acceptance test. This makes the proposed hybrid method particularly suitable for mission-critical applications, where developing an effective acceptance test is a significant challenge.

Kulyagin et al. (2015) investigated the problem of developing fault-tolerant software for satellite control. In order to ensure high reliability of the control software, N-version programming based on the redundancy principle is used. The article discusses in detail the design features of software for satellite communication system control. A model is proposed that allows optimizing the structure of N-version software taking into account the constraints on available resources. A specialized algorithm is developed to solve the optimization problem. The effectiveness of the proposed approach is demonstrated using a specific numerical example, which shows how N-version programming can significantly improve the reliability of the satellite communication system control software. The paper (Wang et al., 2021) presents an integrated approach to developing a reliable software system for autonomous underwater vehicles (AUVs). AUVs play a key role in maritime reconnaissance, search and rescue operations, and military operations, which determines high requirements for the reliability of their software. In this study, a fault-tolerant AUV control module based on advanced fault-tolerant software technologies was developed. Efficient transient fault handling using software redundancy techniques was also implemented. An optimized Long Short-Term Memory (LSTM) model was applied to predict potential AUV failures. This can provide a solid basis for further improvement and evaluation of the AUV software reliability. The paper (Rivera & Chandrasekaran, 2021) presents the development of the research on the integration of fault tolerance and self-healing in robotic systems based on the Robotic Operating System (ROS). With the increasing complexity and diversity of robotics applications, ensuring the reliability of robotic systems is becoming critical. This requires timely fault handling and self-healing of the system. The focus is on demonstrating the development of the research through implementation, test cases and results. This research is in progress and the next step will be the hardware implementation of the

robotic system on the Turtlebot.

In the paper (Iftikhar & Weyns, 2014), the ActivFORMS framework is presented, which uses synchronized automata for modeling and analyzing self-adaptive systems. It implements the MAPE loop on a virtual machine driven by base models and includes a goal management layer for adapting these models. Unlike ActivFORMS, which relies on switching between pre-defined adaptation models, our framework is more general-purpose. It allows building applications by defining abstract knowledge models and control data. The adaptation logic is implemented using online learning on run-time models, which allows for automatic derivation of new adaptation rules. In the context of dynamically changing cyber-physical systems (CPS) with their increasing complexity, uncertainty, and changing topology, traditional adaptation mechanisms prove insufficient. Given the critical importance of CPS for safety, the robustness of their autonomous adaptation becomes a key requirement. To address these issues, we propose an extended MAPE-K architecture with meta-adaptation. It structures the knowledge base to continuously evaluate the accuracy of previous adaptations, learn new rules based on executable runtime models, and verify the correctness of the adaptation logic. The effectiveness of the approach is demonstrated using a temperature monitoring system as an example, enabling the design of understandable and robust dynamically evolving adaptation logics (Klös, Göthel & Glesner, 2018).

## 5. Fault tolerance of software systems

Despite all efforts, the complexity of modern software systems, the unpredictability of the external environment, and the human factor make errors inevitable. The complexity of systems increases the likelihood of hidden defects, the unpredictability of the environment leads to unforeseen situations, and the human factor determines the possibility of errors even with strict adherence to recommendations. Various fault tolerance mechanisms are actively researched and implemented in the academic and industrial environment (Qi et al. 2022).

When discussing fault tolerance in computer systems, it is important to understand the difference between proactive and reactive methods. Reactive fault tolerance methods are aimed at detecting and eliminating faults that have already manifested themselves in the system. The main focus is on minimizing the consequences of a failure and quickly restoring the working state of the system (Gokhroo, Govil & Pilli, 2017).

In this regard, it is necessary to integrate fault tolerance principles into traditional software engineering practices at all stages of development. Modern methods, focused on normal behavior and the assumption that all errors can be eliminated at the development stage, require revision. New approaches and tools must be developed that allow explicitly handling abnormal situations. Each development stage must be supplemented with specialized tools for fault tolerance modeling specific to this stage. Since the 1970s, various approaches have been actively studied and implemented, especially in critical systems. Fault tolerance is achieved through redundancy, which can be implemented at the code, data, or infrastructure level. Creating duplicate copies of the system, data, or replicating the environment increases reliability, but increases the costs of development, maintenance, and infrastructure. In addition, redundancy can affect the responsiveness of the system. However, for critical systems where safety is a priority, such costs are justified.

Software fault tolerance is achieved through specialized programming techniques that include component redundancy and independent versioning. This reduces the likelihood of common failures and increases reliability. The most well-known schemes are N-version programming (NVP) and recovery block (RB), which are based on the redundancy principle.

The concept of N-version programming (NVP) was developed by Chen L. and Avizienis A. Avizienis is one of the pioneers in the field of software reliability theory. The concept of N-version programming is the simultaneous development of several versions of the same program in order to increase the overall reliability of the system. The idea is that errors in one version can be compensated for by the correct operation of other versions (Chen & Avizienis, 1978).

N-version programming includes the following key points (Fig. 2) (Teng & Pham, 2003):

1. Development of several versions - N different implementations of the same function or algorithm are created.

2. Parallel execution - all versions run simultaneously, allowing for real-time results.

3. Redundancy - if one version fails, the results of other working versions can be used.

4. Diversification - versions must be different from each other to avoid the same errors. Independence of development is ensured by separating teams and prohibiting their communication during the software development process. It is recommended that each version use different algorithms, methods, programming languages, and tools. All N versions are executed in parallel, and their results are passed to a voter, who determines the "correct" output using a voting mechanism.

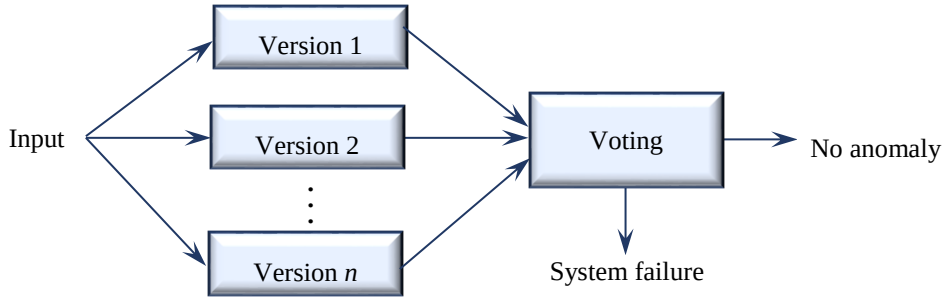


**Fig.2**. N-version programming

Several different voting methods have been proposed. In N-version programming, majority voting is often used to determine the "correct" outcome. In this method, the number of versions (N) is usually made odd to avoid ambiguity. A decision is made if at least (N/2) + 1 versions produce the same outcome (Kovalev, Saramud & Losev, 2020).

The Recovery Block (RB) was developed by Randell B. It is a fault tolerance mechanism using the main and alternative modules with acceptance tests (AT). The main module is tested by AT, and if it fails, the alternative module is launched. The process is repeated until an acceptable result is obtained. If all modules fail AT, the system reports a failure (Fig. 3) (B. Randell, 1975).

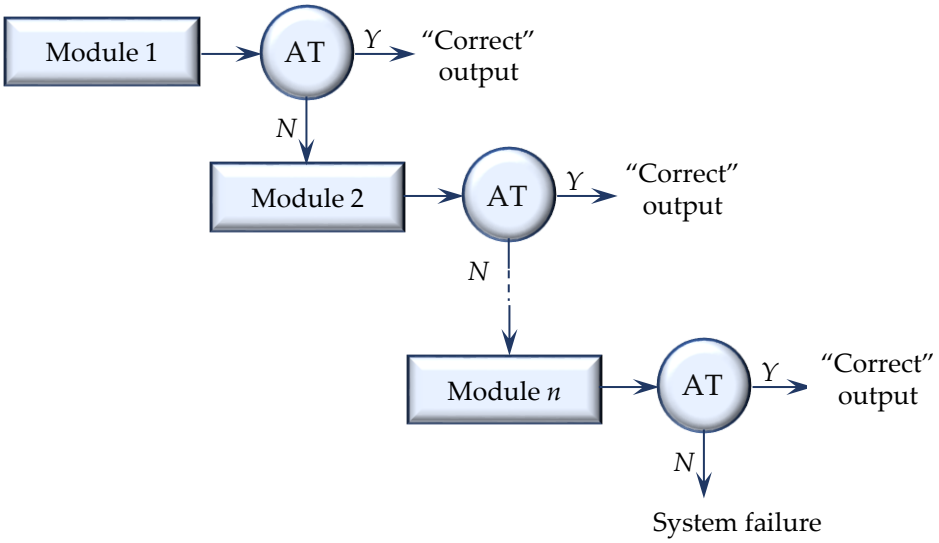


**Fig. 3.** Scheme of the recovery block

The main difference between RB and NVP is the method of executing modules. RB uses sequential execution with the launch of alternative modules in case of failure of the main one, and NVP uses parallel execution. RB also uses acceptance tests (AT) to evaluate the results, unlike voting in NVP. Due to switching to backup modules, the execution time of RB increases, which reduces its effectiveness for real-time systems.

There are some studies aimed at combining the advantages of the Recovery Block (RB) and N-version programming (NVP) to create hybrid methods. These methods aim to leverage the best aspects of both schemes to improve software fault tolerance (Rani & Kaur, 2021; Kumar et al. 2023).

In the context of digital transformation, self-healing systems are not just a convenience, but a necessity to ensure fault tolerance, security and economic efficiency. Self-healing systems are becoming increasingly relevant in the modern

world due to the growing complexity of technological infrastructures, increasing cyber threats and the need to minimize downtime of critical services (Keromytis, 2007). Self-healing systems can automatically detect and block attacks. For businesses (e.g. banks, online stores), even minutes of downtime mean millions in losses. Automatic recovery reduces dependence on the human factor. In driverless cars, industrial robots and smart cities, instant recovery from failures is critical (Bayramova, T.A., 2015).

## 6. Developing a conceptual model

Fault tolerance is a critical but challenging task in software systems due to their dynamic infrastructure and complex interrelations. To overcome these challenges, we present a conceptual model for designing robust self-learning self-adaptive systems (Fig. 4). The operating principle of the proposed model:

**Step 1. Monitoring system.**

The monitoring system continuously monitors the state of software systems and collects various data about its current state:

- Resources (processor, memory, disk);
- Requests processed by the system;
- Data processing speed;
- Information about errors in log files (time of occurrence, type, location, etc.);
- System error notification;
- System availability (uptime and downtime);
- Unauthorized access attempts;
- Detection of malicious activities, etc.

This data is collected using sensors, agent programs installed on servers, log file analysis tools, etc., as well as from specialized software applications**.**

An agent is a small software module installed on servers to collect data, monitor status, and perform specific tasks. Agents monitor the server status and notify about any anomalies. Agents such as Nagios and Zabbix can be used to collect resource usage data.

Log file analysis tools allow you to easily structure and analyze the data collected in log files. It allows you to identify patterns, diagnose problems, detect malicious attacks, and other anomalies. For example, Fluentd is an open source tool for analyzing log files.

The monitoring system works continuously in real time. Data can be presented in the form of graphs, tables, and alarms. The monitoring system plays an important role in increasing the reliability of software systems and allows you to promptly identify problems, prevent failures, and optimize performance.

**Step 2. Anomaly Detection System.**

Anomaly Detection System analyzes the data received from the monitoring system and identifies deviations from previously accepted rules and regulations. This system analyzes big data using various statistical and intelligent methods. Compares current data with historical data or predefined thresholds. When an anomaly is detected, a warning is sent to other system components and personnel. The system that detects anomalies can also be called the Analytical Center. Statistical or intelligent methods are selected to analyze data for anomaly detection. Rules and thresholds for detecting simple anomalies are defined, and tools for visualizing the analysis results are selected. The Anomaly Detection System analyzes data and makes decisions, prevents possible failures and ensures the stable operation of the software system.

Rules and thresholds are simple and effective methods for detecting simple and obvious anomalies. They are used to detect deviations from predetermined norms. Rules are logical conditions that determine which data is considered abnormal. Thresholds are numeric values that define the boundaries of normal behavior. If the data does not meet the rules or exceeds the thresholds, it is considered an anomaly. For example, if the CPU load exceeds 90%, the system registers an anomaly, or if a certain type of error message appears in the log files, the system registers an anomaly. Rules and thresholds are easy to set up and change, but they are not able to detect complex and non-obvious anomalies. Rules and thresholds must be configured manually, which can be labor-intensive. Statistical anomaly detection methods are used to detect anomalies based on an analysis of the statistical characteristics of the data. Statistical methods assume that normal data follows a certain statistical distribution. Anomalies are data that deviate significantly from this distribution.

**Step 3. Diagnostic system.**

The diagnostic system analyzes the anomaly data collected by the monitoring system and the anomaly detection system. It uses various methods and tools to identify the causes of anomalies. The diagnostic system can be thought of as an "expert system" that uses knowledge and experience to identify the causes of anomalies. It may include:

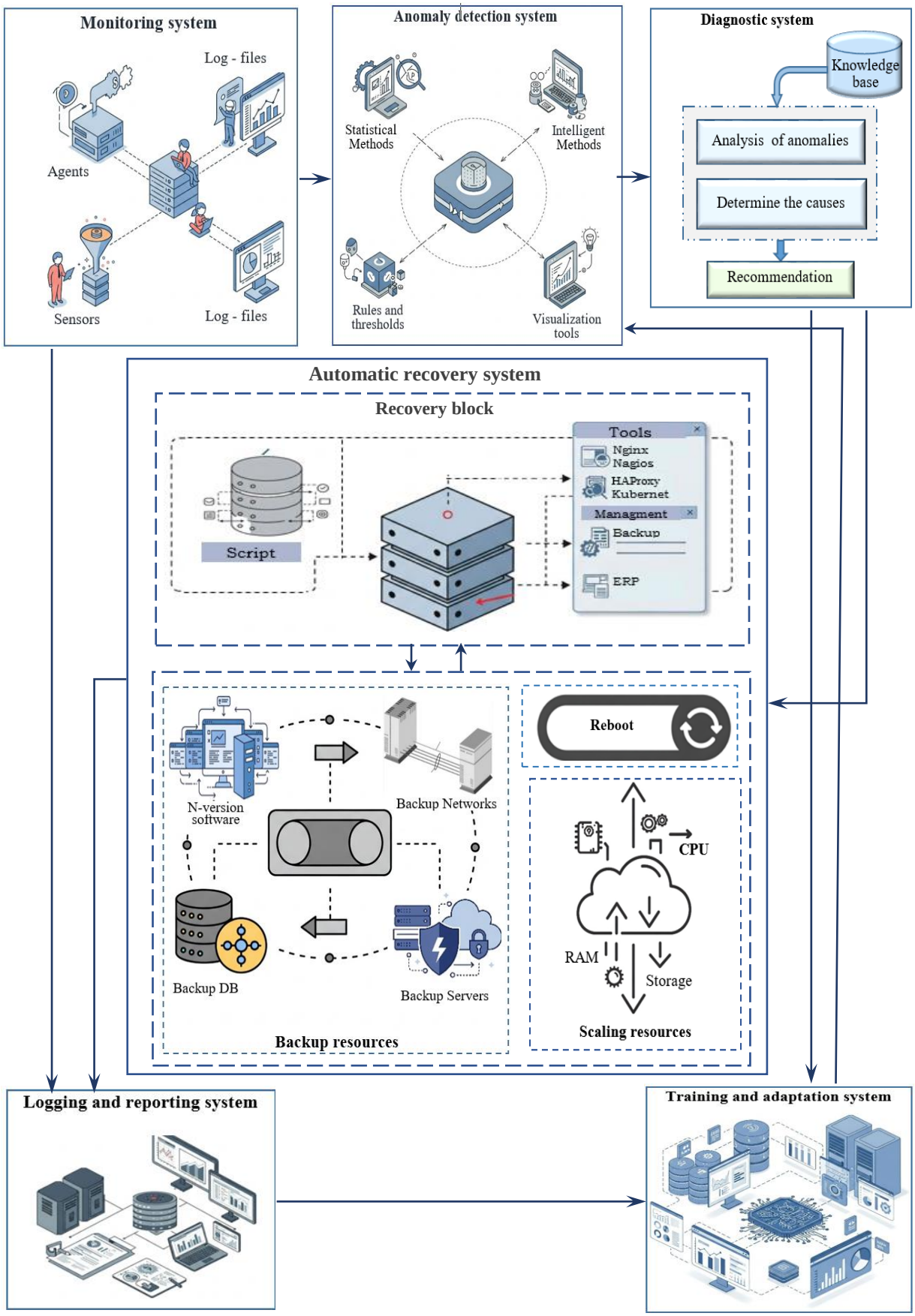


**Fig. 4.** A conceptual model for increasing the software fault tolerance

- Knowledge bases with information on possible causes of anomalies;
- Tools for analyzing logs and other data;
- Algorithms for automatically identifying the causes of anomalies.

Knowledge bases contain information on possible causes of anomalies based on the experience of experts and knowledge of the system. They may include rules, scenarios, descriptions of typical problems and their solutions. Analysis tools allow you to analyze logs, performance metrics, network traffic and other data. Automatic diagnostic algorithms use machine learning and artificial intelligence methods to automatically identify the causes of anomalies. They can analyze large amounts of data and identify complex patterns. The system determines what factors led to the occurrence of an anomaly. These may be errors in the program code, hardware problems, network failures or other reasons. The diagnostic system provides recommendations for eliminating the causes of anomalies. This may be information on what actions need to be taken to restore normal system operation.

**Step 4. Automatic recovery system.**

The automatic recovery system automatically responds to anomalies detected by the diagnostic system and takes actions to restore normal system operation. Its purpose is to minimize downtime and prevent serious consequences of failures.

The automatic recovery system may include a recovery unit (scripts and resource management tools for automatically performing recovery actions), failover tools, and system restart and scaling tools

The system automatically performs predefined actions in response to detected problems. This may include restarting components, switching to backup resources, scaling resources, and other actions.

Restarting components is one of the basic measures for system recovery, restarting components that have stopped responding or are not working correctly. This allows you to quickly eliminate temporary failures and return the system to a working state. Restarting components allows you to restore the operability of components that have stopped responding or are not working correctly. This may be due to temporary failures, software errors, or other problems.

Switching to backup servers or other resources in the event of a failure of the main resources allows the system to continue to operate even if the main components fail. This ensures high availability and reliability of the system. Automatic switching to backup resources allows you to minimize the system downtime. Backup resources can be used for load balancing, which allows you to optimize resource use and prevent overload of individual components.

Types of backup resources:

- Backup servers: Servers that are ready to replace primary servers in the event of a failure.
- Backup databases: Copies of databases that can be used in the event of a failure of the primary databases.
- Backup networks: Backup communication links that can be used in the event of a failure of the primary links.
- Backup data stores: Copies of data stores that can be used in the event of a failure of the primary stores.
- Backup applications: Copies of applications that can be used in the event of a failure of the primary applications.

Resource scaling is the process of automatically increasing or decreasing system resources depending on the current load. This allows you to optimize resource use and prevent overloading of individual components. Resource scaling allows the system to adapt to changing needs. This helps reduce infrastructure costs. Types of resources to scale:

- Processors (CPU): Increasing the number of processors or their performance.
- Memory (RAM): Increasing the amount of RAM.
- Storage: Increase disk space.
- Network: Increase network bandwidth.
- Databases: Increase database resources (e.g. adding replicas).
- An automatic recovery system is an important element of a fault-tolerant system.

**Step 5. Logging and reporting system.**

The logging and reporting system records all important events occurring in the software system and provides reports for analysis. Its purpose is to enable tracking of events, identifying patterns and analyzing problems. The system records information about various events such as errors, warnings, configuration changes, user actions, etc.

The information is recorded in a structured form, which makes it easier to analyze. The system creates reports that allow you to analyze the collected data. Reports can be presented in various

formats (e.g. graphs, tables, charts).

The system stores the collected data for a certain period of time. This allows you to analyze historical data and identify trends.

Types of logs:

- Event logs: Record information about events occurring in the system.
- Error logs: Record information about errors occurring in the system.
- Audit logs: Record information about user actions and configuration changes.
- Performance logs: Record information about the load and performance of the system.

**Step 6. Learning and Adaptation System.**

The learning and adaptation system analyzes the collected data, identifies patterns, and improves the algorithms of other system components. Its task is to make the system more efficient, reliable, and fault-tolerant. The system analyzes the data collected by the monitoring system, logging, and reporting system, as well as the results of other components.

It may include:

- Machine learning algorithms (e.g. neural networks, decision trees).
- Databases for storing training data.
- Tools for analyzing data and visualizing results.

It uses various data analysis methods, including machine learning. The system identifies patterns in the data that can be used to improve the system.

These can be patterns in the occurrence of errors, changes in load, or other patterns.

The system uses the identified patterns to improve the algorithms of other system components. It can improve anomaly detection, diagnostics, or automatic recovery algorithms.

The system allows adaptation to changing operating conditions of the system (can adapt to changes in load, the emergence of new types of errors or other changes).

## Conclusion

The implementation of the proposed integrated approach to developing fault-tolerant systems marks significant progress in creating more reliable and sustainable software. Rejecting the outdated notion of software infallibility and adopting a culture of conscious risk has created a foundation for active management of potential failures.

The development and application of new approaches and tools, including fault-tolerance modeling at all stages of the development life cycle and the creation of specialized tools for simulating abnormal situations, have made it possible to identify and eliminate weaknesses in advance.

The implementation of explicit processing of abnormal situations through the introduction of error detection and handling mechanisms, the development of recovery strategies, and the isolation of failures has significantly increased the system's ability to withstand abnormal situations and minimize their consequences.

The use of specialized tools at each stage - from design to operation - has ensured continuous monitoring and improvement of the system's fault-tolerance characteristics.

Additional recommendations for developing a fault-tolerance culture, training specialists, and funding relevant research and development are key factors for further improving the reliability and sustainability of software products being created.

As a result of implementing this approach, a fault-tolerant system was created that can effectively withstand abnormal situations, ensure the continuity of critical processes and minimize potential risks. Further adherence to the proposed recommendations will strengthen the achieved results and create even more reliable and sustainable software solutions in the future.

## References

Armoush, A., Salewski, F., & Kowalewski, S. (2008). A hybrid fault tolerance method for recovery block with a weak acceptance test. In 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Shanghai, China (pp. 484-491). https://doi.org/10.1109/EUC.2008.102

Bayramova, T. A. & Malikova, N. C. (2024). Developing a conceptual model for improving the software system reliability. Problems of Information Society, 15(1), 42-56. http://doi.org/10.25045/jpis.v15.i1.05

Bayramova, T. A. (2015). The importance of self-management mechanisms to ensure software safety. International Research Journal of Engineering and Technology, 2(3), 1758-1761.

Bernstein, L. (2003). Software fault tolerance forestalls crashes: To err is human; to forgive is fault tolerant. Advances in Computers, 58, 239-286. https://doi.org/10.1016/S0065-2458(03)58006-8

Chen, L. & Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation. In 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8) (pp. 3-9).

Conti, M., Schunter, M., & Askoxylakis, I. (2015). Trust and Trustworthy Computing. In 8th International Conference, TRUST 2015, Heraklion, Greece (pp.300-309). https://doi.org/10.1007/978-3-319-22846-4

de Lemos, R. (2009). On architecting software fault tolerance using abstractions. Electronic Notes in Theoretical Computer Science, 236, 21-32. https://doi.org/10.1016/j.entcs.2009.03.012

de Souza, K. E., Ferrari, F. C., de Camargo, V. V., Ribeiro, M., & Offutt, J. (2025) A systematic review of fault tolerance

techniques for smart city applications. Journal of Systems and Software, 219, 112249. https://doi.org/10.1016/j.jss.2024.112249

Febrero, F., Calero, C., & Moraga, M. Á. (2016). Software reliability modeling based on ISO/IEC SQuaRE. Information and Software Technology, 70, 18-29. https://doi.org/10.1016/j.infsof.2015.09.006

Garousi, V., Felderer, M., Karapıçak, Ç. M., & Yılmaz, U. (2018). Testing embedded software: A survey of the literature. Information and Software Technology, 104, 14-45. https://doi.org/10.1016/j.infsof.2018.06.016

Gokhroo, M. K., Govil, M. C., & Pilli, E. S. (2017). Detecting and mitigating faults in cloud computing environment. In 3rd International Conference on Computational Intelligence & Communication Technology (CICT), Ghaziabad, India (pp. 1-9). https://doi.org/10.1109/CIACT.2017.7977362

Iftikhar, M. U., & Weyns, D. (2014). Activforms: Active formal models for self-adaptation. In 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, (pp. 125-134). https://doi.org/10.1145/2593929.2593944

ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. https://www.iso.org/standard/78176.html

ISO/IEC/IEEE 24765:2017, Systems and software engineering — Vocabulary. https://www.iso.org/standard/71952.html.

Kazimov, T. H., Bayramova, T. A., & Malikova, N. J. (2021). Research of intelligent methods of software testing. System Research & Information Technologies, 4, 42-52. 10.20535/SRIT.2308-8893.2021.4.03

Keromytis, A. D. (2007). Characterizing self-healing software systems. In 4th international conference on mathematical methods, models and architectures for computer networks security (MMM-ACNS) (pp. 1-12).

Khadse, T. S., & Karmore, S. P., (2016). A Novel Approach for Fault Tolerance Control System and Embedded System Security. Procedia Computer Science, 78, 799-806. https://doi.org/10.1016/j.procs.2016.02.059

Klös, V., Göthel, T., & Glesner, S. (2018). Comprehensible and dependable self-learning self-adaptive systems. Journal of systems architecture, 85, 28-42. https://doi.org/10.1016/j.sysarc.2018.03.004

Kovalev, I. V., Saramud, M. V., & Losev, V. V. (2020). Simulation environment for the choice of the decision making algorithm in multi-version real-time system. Information and Software Technology, 120, 106245. https://doi.org/10.1016/j.infsof.2019.106245

Kulyagin, V. A., Tsarev, R. Y., Prokopenko, A. V., Nikiforov, A. Y., & Kovalev, I. V. (2015). N-version design of fault-tolerant control software for communications satellite system. In 2015 International Siberian Conference on Control and Communications (SIBCON) (pp. 1-5). https://doi.org/10.1109/SIBCON.2015.7147116

Kumar, S., Aggarwal, A. G., Gupta, R., & Kapur, P. K. (2023). Software reliability growth model for n-version fault tolerant software with common and independent faults. International Journal of Reliability, Quality and Safety Engineering, 30(06), 2350026. https://doi.org/10.1142/S0218539323500262

Kumari, P. & Kaur, P. (2021). A survey of fault tolerance in cloud computing. Journal of King Saud University-Computer and Information Sciences, 33(10), 1159-1176. https://doi.org/10.1016/j.jksuci.2018.09.021

Nafreen, M., Bhattacharya, S., & Fiondella, L. (2020). Architecture-based software reliability incorporating fault tolerant machine learning. In IEEE Annual Reliability and Maintainability Symposium (RAMS) (pp. 1-6). https://doi.org/10.1109/RAMS48030.2020.9153718

Qi, B., He, Y., Qiu, M., Zhang, P., Cui, Y., & Liu, S. (2022). Research on the design of software architecture based on asynchronous virtual fault tolerance. In IEEE International Conference on Satellite Computing (Satellite) (pp. 60-61). https://doi.org/10.1109/Satellite55519.2022.00023

Randell, B. (1975). System structure for software fault tolerance. In Proceedings of the International Conference on Reliable Software (pp. 437-449).

Rani, S. & Kaur, A. (2021). Automatic test case generation and fault-tolerant framework based on N-version and recovery block mechanism. In K. Khanna, V. V. Estrela, J. J. P. C. Rodrigues (Eds), Cyber Security and Digital Forensics. Lecture Notes on Data Engineering and Communications Technologies (pp. 65–74). https://doi.org/10.1007/978-981-16-3961-6_7

Rivera, L. J. F. & Chandrasekaran, B. (2021). A Software Based Self-Recovering Robotic System Architecture Using ROS. In 7th IEEE International Conference on Mechatronics and Robotics Engineering (ICMRE) (pp. 29-34). https://doi.org/10.1109/ICMRE51691.2021.9384816

Sommerville, I. (2017). Software engineering (10th ed.). Pearson.

Temene, N., Naoum, A., Sergiou, C., Georgiou, C., & Vassiliou, V. (2024). A fault tolerant node placement algorithm for WSNs and IoT networks. Computer Networks, 254, 110835. https://doi.org/10.1016/j.comnet.2024.110835

Teng, X., & Pham, H. (2003). Software fault tolerance. In Handbook of reliability engineering (pp. 585-611). https://doi.org/10.1007/1-85233-841-5_33

Wang, Q., Liu, Y., Xu, T., Sun, X., & Li, J. (2021). Design of Software Fault Tolerant System for Autonomous Underwater Vehicles. In 4th IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC) (pp. 1456-1460). https://doi.org/10.1109/IMCEC51613.2021.9482037