International Journal of Global Tech Management (IJGTM)
Volume 1, Issue 2, July-December, 2024
Available online at: https://pgrpublication.com/index.php/ijgtm

Micro services-Based Architectures: A Pathway to Achieving Fault Tolerance in Distributed Systems

Mr. Jatin Vaghela[1], Mitesh Sinha[2]

[1]Data Base Administrator, Rajshree Global Foods & Spices, NJ, USA
[2]Director -Walmart Marketplace & WFS, USA

Article history: Received: 20 September 2024, Accepted: 3 October 2024, Published online: 18 October 2024.

ABSTRACT

Driven by the demand for scalable, robust, and decentralized systems, this article explores novel trends in micro service development, highlighting the shift from monolithic architectures to micro services. It examines how developments in cloud computing and containerization have enabled the shift from Service-Oriented Architecture (SOA) to micro services. The paper examines important patterns that improve data integrity, scalability, and communication in micro services, including the Saga pattern, event-driven architecture, and service mesh. Small, independent services that communicate via well-defined APIs form MSA, which has many benefits over conventional monolithic designs, including increased scalability, maintainability, and agility. In addition to highlighting how industry titans like Amazon and Netflix have embraced MSA, the paper charts the development of MSA from Service-Oriented Architecture (SOA).

Due to inadequate fault tolerance, there have been instances in the past when crucial applications have failed. When designing a distributed system, a number of problems are looked at and appropriately resolved to provide the required degree of fault tolerance. In addition to explaining fundamental ideas about fault tolerance in distributed contexts, this article defines a number of terms, including failure, fault, fault tolerance, recovery, redundancy, security, etc. Additionally, it outlines four types of fault tolerance and how to get them. The article offers a number of designs and strategies that apply fault tolerance to different aspects of distributed computing. A few active research projects are also covered by these solutions. This paper's main objective is to acquaint readers with the latest research in this field and to provide a knowledge of fault-tolerant distributed systems.

Keywords: -Scalability, Service-Oriented Architecture (SOA),Fault Tolerance, Distributed Computing, Failure, Micro Services, Event-Driven Architecture, MSA, Distributed System.

INTRODUCTION

Systems for distributed computing are made up of a range of software and hardware components. The availability of services and unexpected, possibly disruptive behaviour may result from the failure of any one of these elements. There have been instances in the past when key applications have failed due to a lack of fault tolerance [1]. A small number of them just missed. The code contained an uninitialized counter used in a "computed go to" command that caused all four of the redundant flight computers to simultaneously branch off to a memory address containing no code [1, 2].

In one such instance, the scheduled lift-off of the space shuttle Columbia on October 9, 1981, was delayed because of a small fuel spill and a few missing tiles. 17 more identical systematic defects in the flight control software were found during a follow-up software study utilizing the unique information gained from this event, one of which also had the potential to cause a catastrophic failure [2, 3].

Software development has seen a major breakthrough with the introduction of Micro Service Architecture (MSA) [3]. In contrast to conventional monolithic designs, it provides a more flexible and modular method of developing intricate applications [3, 4].

1.1 Architecture Micro service

According to the software design pattern known as architecture, an application is made up of discrete, stand-alone services that interact with one another via well-defined APIs [1, 2]. Every service may be independently built, deployed, and scaled, is self-contained, and is in charge of a particular piece of functionality [2, 3]. Maintainability and agility are improved by this architectural approach, which encourages a decoupled system where services may be changed or replaced without impacting the program as a whole [4].

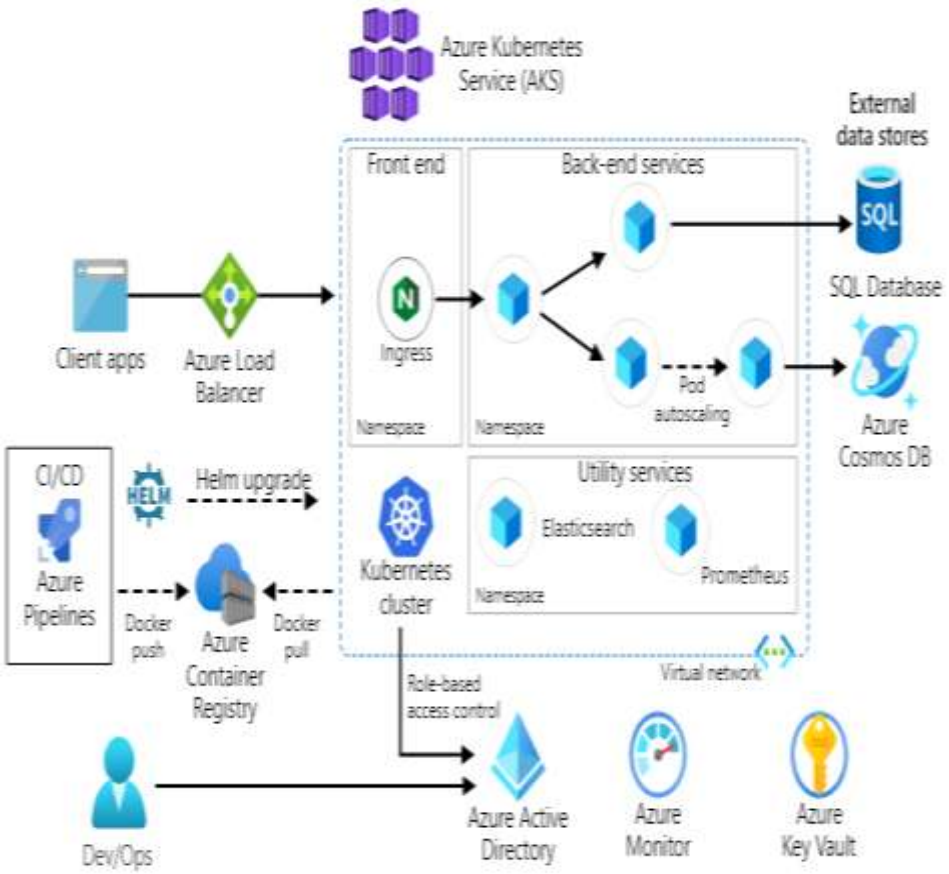


Fig. 1 Microservice Architecture. [2, 4]

1.2 Importance of Microservice Architecture

Because micro services provide so many advantages over conventional monolithic designs, they are becoming more and more popular [2].

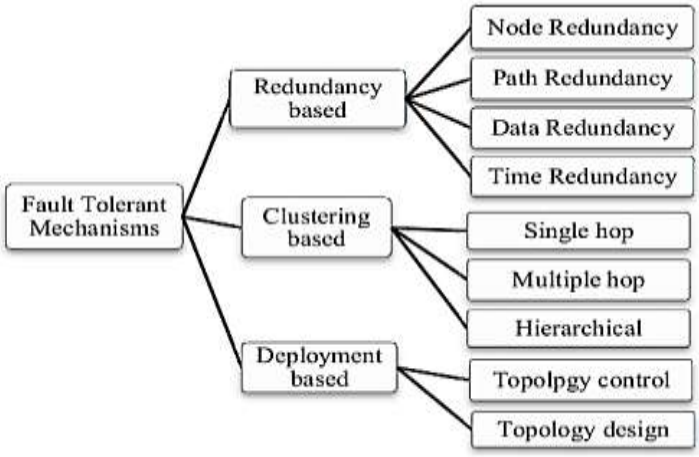
- **Benefits over Monolithic Architectures:** All of an application's components are combined into a single, coherent entity by monolithic designs. Although this method makes early development and deployment easier [2, 5], it presents serious problems as the program expands. It might be challenging to update or grow individual components of a monolithic system when it becomes a complex web of dependencies. Microservices, on the other hand, encourage concern separation, enabling teams to work on many services at the same time without affecting one another [5]. Faster development cycles, simpler debugging, and more effective use of resources are all made possible by this modularity [5]. Microservices may also be scaled individually, guaranteeing cost effectiveness and peak performance.

- **Industry Adoption and Trends:** All of an application's components are combined into a single, coherent entity by monolithic designs. Although this method makes early development and deployment easier [2, 5], it presents serious problems as the program expands. It might be challenging to update or grow individual components of a monolithic system when it becomes a complex web of dependencies. Microservices, on the other hand, encourage concern separation, enabling teams to work on many services at the same time without affecting one another [5]. Faster development cycles, simpler debugging, and more effective use of resources are all made possible by this modularity [5]. Microservices may also be scaled individually, guaranteeing cost effectiveness and peak performance.

Analysis of Impacts on Development Practices

The effects of these novel patterns on development practices will also be examined in the study. This involves investigating the ways in which microservices impact operational procedures, development workflows, and team structures [5]. By comprehending these effects, the study seeks to provide useful suggestions for businesses wishing to implement or improve their microservice architectures [6]. The following will be important areas of analysis:

- **Team Structures:** A move toward smaller, cross-functional teams is often required by microservices [5, 6]. The study will investigate how businesses might set up their teams to optimize microservices' advantages, such as improved teamwork and quicker development cycles [7].



- **Operational Processes:** Strong management, logging, and monitoring procedures are necessary for microservices [6], 7. The study will examine methods and tools for microservice environment management, such as fault tolerance mechanisms, service discovery, and container orchestration [7].

A significant turning point in the development of software design is represented by Service-Oriented design (SOA). SOA is a design pattern in which application components use a communication protocol via a network to deliver services to other components [7, 8]. In order to handle the complexity of large-scale corporate systems, the idea of service-oriented architecture (SOA) gained popularity in the early 2000s [7]. Using loosely linked services to boost software systems' scalability and flexibility is the fundamental idea behind service-oriented architecture (SOA) [8]. Within a SOA architecture, every service is a distinct functional unit that can be scaled, deployed, and managed on its own [8, 9].

Because of its modularity, services may be reused across several systems and applications, creating an environment in which modifications to one service don't need adjustments to others. Improved scalability, increased interoperability across different systems, and the capacity to capitalize on current investments in legacy systems are just a few of the significant advantages that SOA offered [9]. But it too had its difficulties [9]. Significant cost was imposed by the strong dependence on XML-based communications (like SOAP), and bottlenecks and decreased agility may result from the centralized governance often needed for SOA deployments [9, 10].

1.3 Best Practices in Microservice Design

1. **Domain-Driven Design:** The goal of the strategic software development methodology known as Domain-Driven Design (DDD) is to create a strong domain model [9, 10]. When it comes to microservices, DDD facilitates the breakdown of a complicated system into more manageable, smaller services [10, 11]. To improve communication between both technical and non-technology stakeholders, the fundamental objective is to tightly match the software model with the business domain [12]. DDD highlights how crucial it is to comprehend both the core domain and its subdomains. Through the identification of the fundamental domain, supporting subdomains, and generalized subdomains that are developers may ascertain which aspects of the system need more attention and funding [12]. In conclusion, DDD offers an organized method for developing a common language, comprehending the domain, and building a domain model [12]. Building a scalable and stable microservice architecture and establishing distinct service boundaries need these actions [12, 13].

2. **Bounded Contexts:** A key idea in DDD is the notion of bounded contexts, which denote a certain segment of the business subject with a well-defined border [14]. A specific component of the domain model, including its entities, value objects, aggregates, and services, is included in each limited context [15] [13, 14].

Analysing the domain model and comprehending the connections between various system components are necessary for identifying limited contexts [14]. This study helps in deciding how to divide the system into distinct services and where to set the borders. Bounded contexts have the advantage of offering a natural method of handling complexity. Developers may concentrate on one context at a time by segmenting the domain into smaller, more manageable chunks [15], which makes it simpler to comprehend, apply, and maintain.

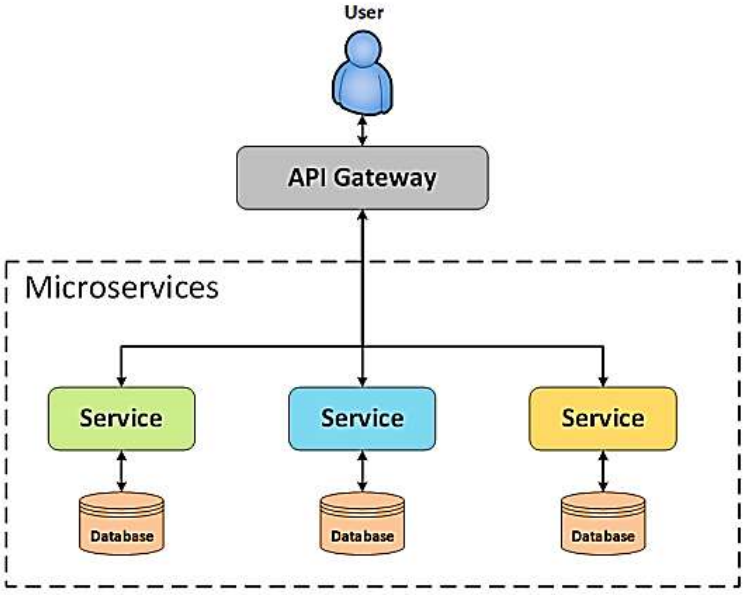


Fig. 2 Service Boundaries. [15]

Additionally, bounded contexts help teams communicate more effectively. Each team may get a thorough awareness of a certain constrained context when they are assigned to it [15,16], which promotes more effective communication and decision-making. To sum up, in a microservice architecture, constrained contexts are crucial for establishing distinct service boundaries [16, 17]. They aid in the management of complexity, enhance communication, and guarantee that every service stays concentrated on a certain facet of the domain [18].

Up until efforts to standardize ideas and terminology, the area of fault tolerant computing has always been application-specific [17]. Later on, however, the field would be "fragmented." Since then, a more formal and abstract approach has improved our comprehension of the issues at hand and is essential to creating systems that are robust to faults. From failure detection to mobile security, researchers have worked on a variety of distributed fault tolerance models. To offer a high-level, simple modelling framework for fault-tolerant distributed computing systems and to analyse how fault tolerance mechanisms affect the perceived dependability of users, one such method outlines action models and path-based solution algorithms [16, 17]. The usefulness of transactions and other fault tolerance techniques in the concurrent programming language Ada is reviewed in this study [18].

There are a lot of programming languages that have fault tolerance built in. Some of them, such Fault Tolerant Concurrent C (FTCC), are expansions of already-existing languages [18, 19]. Instead of only using checkpoint and restart, a new MPI implementation known as FT-MPI suggested that the semantics and related failure modes be fully controlled by the application [19]. The issue of various resource managers sharing logging services is resolved by the Quicksilver distributed operating system's new Log Manager for shared logging service [20].

Deployed a distributed real-time test bed for the Simplified Unmanned Vehicle System (SUV) using fault tolerance approaches at the system level [20, 21]. The creation of a twin server-based paradigm for fault-tolerant servers that offers fault-tolerant services for the RHODOS distributed operating system, which is built on micro kernels. A compiler that preserves the accuracy and secrecy of an input protocol for n semi-honest parties while producing a fault-tolerant protocol [22]. Explains how security and randomness are related in the setting of multiparty calculations.

TYPES OF FAULT TOLERANCE AND FAILURE DETECTION

1.4 Types of Fault

Behaviour of the tolerance system in terms of its liveness and safety. A distributed application has to meet both of the aforementioned requirements in order to function properly [19, 20]. Before discussing the various kinds of fault tolerances, we shall attempt to clarify these ideas. Safety is the absence of a certain "bad thing" in a system. By defining when an execution e is "not safe" given a property p, this may be formalized as follows: if $e \notin p$, there must be a distinct and identifiable event e that forbids any potential system execution from being safe. For instance, updating a shared object simultaneously [20, 21]. If the system consistently stays inside a set of safe states, then the distributed program is safe [23]. A liveness property, on the other hand, asserts that a "good thing" will ultimately occur while the system is operating. A system is officially considered active for property p if and only if it may be lawfully extended to

stay in p. "Legally speaking, this implies that the system itself must permit the expansion. For instance, a process that has been waiting for access to a shared object will be granted it at last [21, 22]. A distributed program A must meet both its safety and liveness properties in order to function properly [19,20].

Now, how does an impacted party's property fare when a defect occurs?

Table 1 Forms of Fault Tolerance. [21, 22]

	Live	Not Live
Safe	Masking	Fail Safe
Not Safe	Non-Masking	None

Although the masking form of fault tolerance is the most desired, its implementation is the most costly. Applications that have this kind of fault tolerance can transparently withstand errors [17, 18]. The third scenario, however, is the least ideal as neither safety nor liveness are assured [18]. Given the significance of leaving the system in a safe state, intermediate fail safe is preferable to non-masking among the two and is a current study topic. Even if the system's output in a non-masking type scenario may not be ideal or accurate, the outcome is nonetheless produced [18, 19]. Self-stabilization, a specialty of non-masking fault tolerance, has been extensively pursued recently [19, 20]. Such programs can tolerate a wide range of temporary errors. However, creating and testing such programs is challenging [19, 20].

1.5 Failure Detection

As you will see later in Section 5, failure detection is crucial to ensuring the system's safety and liveness [20, 22]. Numerous academics have worked to identify the sort of failure in the system and detect it promptly. Using rigorous formalization to solve the consensus and atomic broadcast problems by unreliable failure detectors has been one of the major contributions [23, 24]. Measured their effectiveness utilizing largest lower limits for the primary class of failure detectors and the longest message chain prior to decision. By translating the consensus procedure suggested for one model to another, system models and failure detectors may be used. Transient failure detector hierarchy that identifies the resources needed to deploy them as well as the incidence of transient faults [23].

1.6 Redundancy

It is always conceivable for a system to collapse under severe or frequent fault assaults, regardless of how effectively it is designed to withstand them [23]. Redundancy is a prerequisite for fault tolerance, but it is not enough. It identifies two types of redundancy: one in time and one in space. Redundancy in time refers to a collection of program actions that are never carried out in the absence of errors, while redundancy in space refers to a set of program configurations that are never attained in the absence of problems [22].

It should be noted that accurate defect detection is crucial to the system's ability to function safely. Information regarding state space and/or program activities is required for detection. The system's liveness feature is ensured via correction upon fault detection [24]. Therefore, we can see that it is simpler to ensure safety (by detection) than liveness (by rectification). The fact that these detection and repair systems themselves need to be fault tolerant must also be mentioned. Component replication provides redundancy in space [24]. Software examples of space redundancy include adding parity bits to transmissions, whereas hardware examples include tandem systems. Time redundancy is similar to repeating the calculation.

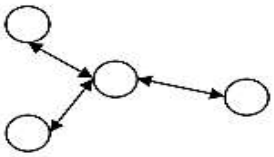
1.7 Rollback and Recovery

For a system to retain its masking fault tolerance, rollback and recovery procedures are necessary. The system's recovery manager employs a number of these strategies to restore the system to a consistent state from which all of its operations may resume [24]. An entry placed in the recovery file at a predetermined period of time is called a checkpoint, and it will cause all of the currently committed data to be stored in stable storage [25, 26]. If the process fails, it will compel any other processes that rely on it to roll back to the most recent checkpoint. The recovery manager is in charge of putting the right check pointing method into place so that, after a failure, the system is restored to a globally consistent state. To provide a dependable distributed environment, a variety of check pointing and rollback recovery techniques are used [18,19]. We'll concentrate on a check pointing technique that relies on the processes' communication patterns.

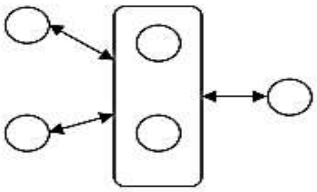
II. CASE STUDY: SOMERSAULT

A middleware called Somersault is used to create and integrate distributed, fault-tolerant software components. Replication transparency may be achieved by combining it with ORB, which offers a fault-tolerant communications transport protocol. The design aims to support applications that need high message throughput, consistent state, assured message delivery, and continuous availability [19, 20]. "Roll forward" strategy, which duplicates the process and makes

it highly accessible in the event that the main fails. According to its claims, it can withstand operating system, hardware, and non-replicated application failures. In order to guarantee identical functionality and consistent state in every pair of replica processes, it offers a C++ middleware library [20]. It is possible to combine this library with CORBA. The two components of Somersault are the recovery unit of duplicated processes dispersed across the network and the basic non-fault-tolerant unit [20, 21]. The recovery unit functions as a single entity in Somersault's implementation of the n to m connection-oriented messaging protocol [21].



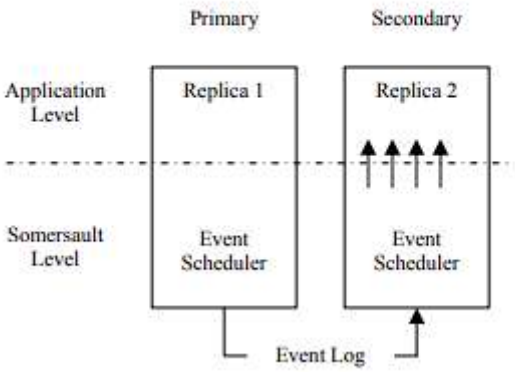
Communication Patterns in Regular Distributed Systems



Communicating Units in Somersault

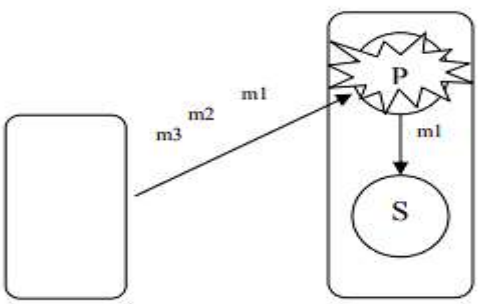
A simple example of three processes—the primary, secondary, and witness processes—will be examined for our comprehension [22, 23]. Primary and secondary are engaged in failure detection and duplicate the application. The witness breaks the tie. Somersault follows these procedures:

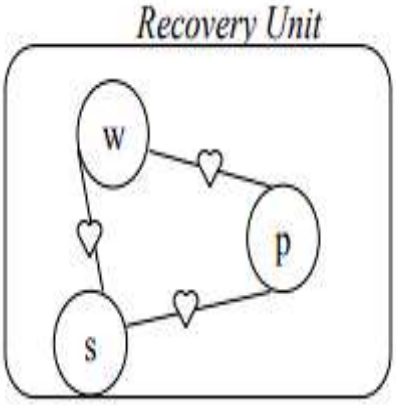
- **Failure Detection:**Processes exchange heartbeat messages in order to communicate. When a heartbeat is absent, failure is identified [24].



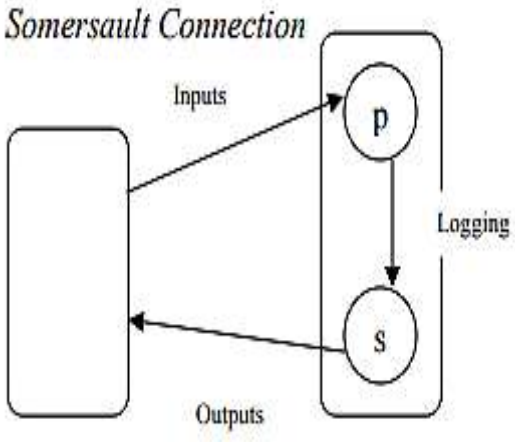
- **Unit Communication and Secondary Sender Protocol:**Only one copy of each outgoing is delivered, even if all replicas create output messages and consume input messages [23, 24]. Aside from [25, 26], the message order is as shown in the figure 1.

Failure of Primary





- **Replication:** With a logging channel connecting the main and secondary processes, process pairs function as replicas [26, 27]. The secondary executes precisely the non-deterministic events that the primary does [28, 29].



- **Failover:** In the case that the main or secondary communication fails, the windowing protocol is utilized at the unit-to-unit communication level to guarantee that messages are not lost or rearranged [29, 30]. The identical input message will be sent to the secondary when the primary fails, and the job will be repeated [30, 31].

- **Recovery:** Secondary will become primary [32] if primary fails [35]. It will instantly duplicate itself to produce a second secondary that will be in sync with it [33, 34].

CONCLUSION

In summary, advancements in e-commerce, financial services, and media and entertainment are critical to improving user experience, scalability, and performance. To remain competitive and satisfy changing user demands, e-commerce platforms must implement API Gateways and Event-Driven Architecture; financial services must apply the Saga Pattern and efficient deployment strategies; and media and entertainment must use scalability solutions and content delivery strategies. It takes careful planning and execution to integrate new technologies like automation and the Internet of Things in the industrial sector. Particularly relevant are the difficulties with integration, financial limitations, and sustaining user interest.

As more essential applications are being used, fault tolerance is receiving greater attention and significance. With the introduction of distributed applications and networked systems, the field has become more complicated. Despite a great deal of theoretical and practical investigation, this field has yet to be thoroughly investigated. Implementing fault tolerance is not difficult, despite some inherent restrictions. As we've seen, formal methods of addressing the issue via safety and liveness have improved our comprehension of the idea and enabled us to apply it precisely. In order to make this topic easier to grasp, researchers have worked hard to codify it. Attempted to determine some fundamental characteristics of a fault-tolerant system. Attempted to utilize a component-based approach to formalize the development of masking fault-tolerant applications.

By gradually adding components to fault-intolerant algorithms, they were able to convert them to non-masking fault-tolerant programs and, ultimately, to masking ones. However, in other situations, just the system's liveness may be ensured and detection is not feasible before the system enters a hazardous condition. Algorithms for fault detection have been successfully used to effectively resolve consensus issues. Replication level, protocols, fault detection algorithms, and other design choices are more often compromised while creating fault tolerance systems in reality. Probability of accurate output as a criterion for assessing and making design decisions.

Future efforts must be made to put the approaches outlined for creating flexible, reliable systems into practice. Despite a changing execution environment, these systems may continue to operate for a long time. It offers a linguistic structure for characterizing reliable systems. In conclusion, it is well known that developing and comprehending fault-tolerant systems is a challenging undertaking, and that anticipating and managing the behaviours of system components and activities is a problem that lies ahead.

REFERENCES

[1]. M., Reza Hoseinyfarahabady "A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform." IEEE Transactions on Parallel and Distributed Systems 29.7 (2018): 1442-1455
[2]. NS Tung, V Kamboj, A Bhardwaj, “Unit commitment dynamics-an introduction”, International Journal of Computer Science & Information Technology Research Excellence, Volume 2, Issue 1, Pages 70-74, 2012.
[3]. VK Kamboj, A Bhardwaj, HS Bhullar, K Arora, K Kaur, Mathematical model of reliability assessment for generation system, Power Engineering and Optimization Conference (PEOCO) Melaka, Malaysia, 2012 IEEE.
[4]. Jani, Y. "spring boot for microservices: Patterns, challenges, and best practices." European Journal of Advances in Engineering and Technology 7.7 (2020): 73-78.
[5]. Parikh, H., Prajapati, B., Patel, M., & Dave, G. (2023). A quick FT-IR method for estimation of α-amylase resistant starch from banana flour and the breadmaking process. Journal of Food Measurement and Characterization, 17(4), 3568-3578.
[6]. Patel, M., Parikh, H., & Dave, G. (2023). Chitosan flakes-mediated diatom harvesting from natural water sources. Water Science & Technology, 87(7), 1732-1746.
[7]. S., Chanthakit "An iot system design with real-time stream processing and data flow integration." RI2C 2019 - 2019 Research, Invention, and Innovation Congress (2019).
[8]. Y.C., Yang "Web-based machine learning modeling in a cyber-physical system construction assistant." 2019 IEEE Eurasia Conference on IOT, Communication and Engineering, ECICE 2019 (2019): 478-481.
[9]. M., Salehe "Videopipe: building video stream processing pipelines at the edge." Middleware Industry 2019 - Proceedings of the 2019 20th International Middleware Conference Industrial Track, Part of Middleware 2019 (2019): 43-49.
[10]. E., Truyen "A comprehensive feature comparison study of open-source container orchestration frameworks." Applied Sciences (Switzerland) 9.5 (2019).
[11]. Navpreet Singh Tung, Gurpreet Kaur, Gaganpreet Kaur, Amit Bhardwaj, Optimization Techniques in Unit Commitment A Review, International Journal of Engineering Science and Technology (IJEST), Volume 4, Issue, 04, Pages 1623-1627.
[12]. Amit Bharadwaj, Vikram Kumar Kamboj, Dynamic programming approach in power system unit commitment, International Journal of Advanced Research and Technology, Issue 2, 2012.
[13]. S., Zhelev "Using microservices and event driven architecture for big data stream processing." AIP Conference Proceedings 2172 (2019).
[14]. Singh, Vivek, and Neha Yadav. "A Study on Predictive Maintenance in IoT Infrastructure by influencing AI for Reliability Engineering."International Journal of Enhanced Research in Science, Technology & Engineering, 2022.
[15]. K., Li "Financial big data hot and cold separation scheme based on hbase and redis." Proceedings - 2019 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking, ISPA/BDCloud/SustainCom/SocialCom 2019 (2019): 1612-1617.