A FAULT-TOLERANT DEPLOYMENT APPROACH FOR MICROSERVICES IN THE DESIGN PHASE OF SOFTWARE DEVELOPMENT LIFE CYCLE

YAZILIM GELİŞTİRME YAŞAM DÖNGÜSÜNÜN TASARIM AŞAMASINDA MİKROSERVİSLER İÇİN HATA TOLERANSLI BİR DAĞITIM YAKLAŞIMI

MUSTAFA YILMAZ

PROF. DR. AHMET BURAK CAN

Supervisor

ASSIST. PROF. IŞIL KARABEY AKSAKALLI

2nd Supervisor

Submitted to

Graduate School of Science and Engineering of Hacettepe University

as a Partial Fulfillment to the Requirements

for the Award of the Degree of Master of Science

in Computer Engineering

May 2024

ABSTRACT

A FAULT-TOLERANT DEPLOYMENT APPROACH FOR MICROSERVICES IN THE DESIGN PHASE OF SOFTWARE DEVELOPMENT LIFE CYCLE

Mustafa YILMAZ

Master of Science, Computer Engineering

Supervisor: Prof. Dr. Ahmet Burak CAN

2nd Supervisor: Assist. Prof. Işıl KARABEY AKSAKALLI

May 2024, 64 pages

The concept of microservices architecture has gained significant interest in recent years, primarily due to its capacity to develop large-scale and extensive software systems while offering improved scalability, flexibility, and maintainability. Ensuring fault tolerance during the design phase of the microservices architecture is essential to achieve an efficient and reliable product. This paper introduces a methodology to create a fault-tolerant deployment model of microservices on limited capacitated nodes (servers) in a model-driven architecture. Our approach aims to minimize the total communication cost while maintaining a reliable and fault-tolerant system in the design phase of the software development lifecycle. To evaluate this approach, we conducted various experiments for different capacities of nodes and different numbers of microservice instances. As a result of the experiments, the proposed fault-tolerant algorithm showed improvement rates between 3% to 29% in communication cost compared to the Best Fit, Next Fit, Hungarian and Round Robin algorithms in many cases.

Keywords: fault tolerance in design phase, deployment optimization, microservices architecture, communication cost in microservices deployment

ÖZET

YAZILIM GELİŞTİRME YAŞAM DÖNGÜSÜNÜN TASARIM AŞAMASINDA MİKROSERVİSLER İÇİN HATA TOLERANSLI BİR DAĞITIM YAKLAŞIMI

Mustafa YILMAZ
Yüksek Lisans, Bilgisayar Mühendisliği
Danışman: Prof. Dr. Ahmet Burak CAN
Eş Danışman: Dr. Öğr. Üyesi Işıl KARABEY AKSAKALLI
Mayıs 2024, 64 sayfa

Mikroservis mimarisi kavramı, gelişmiş ölçeklenebilirlik, esneklik ve bakım kolaylığı sunarken aynı zamanda büyük ölçekli ve kapsamlı yazılım sistemleri geliştirme kapasitesi nedeniyle son yıllarda önemli bir ilgi kazanmıştır. Ancak mikroservis mimarisinin tasarım aşamasında hata toleransının sağlanması, yürütme süresi sürecinde verimli ve güvenilir dağıtım için çok önemlidir. Bu çalışmada model güdümlü bir mimari üzerinde mikroservisleri sınırlı kapasiteli düğümlere (sunucular) dağıtmak için hataya dayanıklı bir yaklaşım sunulmaktadır. Bu yaklaşım, yazılım geliştirme yaşam döngüsünün ilk aşamasında güvenilir ve hataya toleranslı bir sistemi korurken toplam iletişim ve yürütme maliyetlerini en aza indirmeyi amaçlamaktadır. Önerilen yaklaşımı değerlendirmek için farklı kapasitelere ve farklı sayıda mikroservis örneklerine göre çeşitli deneyler gerçekleştirilmiştir. Yapılan deneyler sonucunda önerilen hataya toleranslı algoritma, birçok durumda Best Fit, Next Fit, Hungarian ve Round Robin algoritmalarına kıyasla iletişim maliyetinde %3 ile %29 arasında iyileşme oranları göstermiştir.

Keywords: - tasarım aşamasında hataya tolerans, dağıtım optimizasyonu, mikroservis mimarisi, mikroservis dağıtımlarında haberleşme maliyeti

ACKNOWLEDGEMENTS

Tez çalışmamda bana yol gösteren ve en başından beri desteğini ve bilgilerini esirgemeyen tez danışmanlarım Sayın Prof. Dr. Ahmet Burak CAN, Sayın Dr. Öğr. Üyesi Işıl KARABEY AKSAKALLI ve Sayın Dr. Turgay ÇELİK'e,

Tüm eğitim hayatım boyunca maddi ve manevi desteklerini hiçbir zaman esirgemeyen kıymetli aileme,

Sevgisini ve desteğini her zaman hissettiğim sevgili eşim Elif Nur YILMAZ'a

sonsuz şükranlarımı sunuyorum.

CONTENTS

Page

ABSTRACT i

ÖZET iii

ACKNOWLEDGEMENTS v

CONTENTS vi

TABLES viii

FIGURES ix

1. INTRODUCTION 1

2. BACKGROUND 3

  2.1. Microservice Concept 3

  2.2. Monolith versus Microservice 4

    2.2.1. Monolithic Architecture 4

    2.2.2. Microservices Architecture 5

  2.3. Fault Tolerance 6

  2.4. Importance of Design Phase in Software Development Lifecycle 7

  2.5. Cloud Technologies 7

  2.6. Virtualization 8

  2.7. Containerization 8

    2.7.1. Docker and Kubernetes 9

3. RELATED WORK 10

4. FAULT-TOLERANT DEPLOYMENT APPROACH FOR DESIGN PHASE 14

  4.1. Mathematical model of the proposed approach 16

  4.2. Comparative Microservice Assignment Algorithms 20

    4.2.1. Best Fit 20

    4.2.2. Next Fit 21

    4.2.3. Hungarian 22

    4.2.4. Round Robin 24

    4.2.5. Proposed Fault-Tolerant Algorithm 25

5. EXPERIMENTAL RESULTS 27
5.1. Case Study: Online Book shopping 27
5.2. Simulation Environment 30
5.3. Experiment 1: Identical Microservices and Nodes 30
5.3.1. Occupancy Less Than 50% 31
5.3.2. Occupancy More Than 50% 32
5.4. Experiment 2: The effect of memory capacities of the nodes 34
5.4.1. Occupancy Less Than 50% 34
5.4.2. Occupancy More Than 50% 36
5.5. Experiment 3: The effect of the number of microservice instances 38
5.5.1. Occupancy Less Than 50% 38
5.5.2. Occupancy More Than 50% 40
6. DISCUSSION AND CONCLUSION 43

TABLES

Page

Table 3.1 Comparison of fault-tolerant approaches that are used for microservices in the literature and this thesis 13

Table 5.1 Sample publish/subscribe relation schema for the case study........... 28

Table 5.2 Number of microservices and capacities of each instances during deployment for all algorithms 30

Table 5.3 Improvement rates and total cost values of the algorithms below 50% node occupancy in identical situation 32

Table 5.4 Improvement rates and total cost values of the algorithms above 50% node occupancy in identical situation 33

Table 5.5 Improvement rates and total cost values of the algorithms after the capacity change in the nodes 36

Table 5.6 Improvement rates and total cost values of the algorithms after the capacity change in the nodes 37

Table 5.7 Number of microservice instances and capacities of each instances during deployment for all algorithms 38

Table 5.8 Improvement rates and total cost values of the algorithms after number of instances changes 40

Table 5.9 Improvement rates and total cost values of the algorithms after number of instances changes 41

FIGURES

		Page
Figure 4.1	Flow diagram of the proposed fault-tolerant approach	14
Figure 5.1	Created microservices for the online shopping case study	29
Figure 5.2	Identical processors created for the experiments	29
Figure 5.3	Deployment structure using Fault Tolerant algorithm	31
Figure 5.4	Deployment structure using Round Robin	31
Figure 5.5	Deployment structure using Fault Tolerant algorithm	32
Figure 5.6	Deployment structure using Best Fit	32
Figure 5.7	The different memory capacities according to the nodes	34
Figure 5.8	Deployment structure using Fault Tolerant algorithm	35
Figure 5.9	Deployment structure using Round Robin algorithm	35
Figure 5.10	Deployment structure using Fault Tolerant algorithm	36
Figure 5.11	Deployment structure using Best Fit algorithm	37
Figure 5.12	Deployment structure using Fault Tolerant algorithm	39
Figure 5.13	Deployment structure using Round Robin algorithm	39
Figure 5.14	Deployment structure using Fault Tolerant algorithm	40
Figure 5.15	Deployment structure using Best Fit algorithm	40

1. INTRODUCTION

Microservices architecture (MSA) is a design approach for creating distributed applications that can scale effectively [1]. This functionality is achieved using independent, highly cohesive, loosely coupled components [2]. The MSA can encompass many services, potentially numbering in the thousands, that work together to form the overall system. These services communicate with each other, since they use shared data. To reduce total communication cost and improve scalability, deploying frequently communicating services to the same physical resource is reasonable and effective in reducing network utilization [3]. Besides, when optimizing the cost of inter-service communication in microservice architectures, the fault tolerance of the system must be taken into account. Usually, server and service failures are likely to occur as microservice architectures are deployed in personalized services which expose to frequent network failures. Therefore, fault tolerant approaches help to ensure uninterrupted availability and reliability of the system. In a microservice architecture approach, each service is responsible for a specific functionality and loosely coupled to other services. Although it is thought that a failure in a service will not affect the others, in practice, services that share data with each other can fail in succession and destabilize the system [4]. In the case of fault-tolerant approaches, a failure service can be isolated and prevented from spreading to other services. Testing this approach in the design phase before the development and implementation of the system minimizes the risk of errors in the runtime phase. If the failure of a single service or server affects the operational performance of the entire system, service dependencies and communication infrastructures need to be carefully evaluated in the system architecture. Potential sources of failure in the design phase of the software development lifecycle, and distributing service instances in case of failure in a balanced manner should be considered in accordance with the communication cost among microservices.

In this thesis, the deployment of microservices to servers with limited resources in the design phase, which is the first phase of the software development life cycle, is performed in a fault tolerant manner. In order to minimize the cost of inter-service communication, the proposed

approach distributes each service instance to the servers in a balanced manner, and tolerates server failures. Furthermore, backup servers are provided taking into account the memory capacity of the microservices and the total memory capacity of the servers to ensure fault tolerance. Specifically, our objective function is generated to minimize communication cost while ensuring fault tolerance in the design phase of a microservice architecture.

To evaluate the proposed approach, we conducted three experiments with the Micro-IDE tool [5][6]. As a case study, four microservices with various number of instances are deployed to servers with limited capacity resources. Several parameters are changed in each experiment and the results of the changes were observed. The properties of the microservices and nodes(processors) were identified as identical in the first experiment. Each of the four microservices were carried out at less than and more than 50% occupancy of nodes. In the first experiment, the communication cost efficiency of the algorithms was compared with identical values. In the second experiment, the memory capacities of the nodes; in the third experiment, the number of microservice instances were changed, and the communication cost rates were compared among five algorithms. The findings emphasize the significance of fault tolerance in microservices and demonstrate how our proposed approach can perform efficient and fault-tolerant deployment while minimizing communication costs.

The rest of this paper is organized as follows: Section 2 includes background information on the concept of microservices, the comparison of monolithic and microservice architectures, and the importance of the design phase in the software development lifecycle. Section 3 discusses related work based on fault-tolerant approaches for MSA. Section 4 introduces a novel fault-tolerant deployment approach for deploying services to limited capacitated servers and introduces the compared algorithms. As a subsection, 4.1 defines the mathimatical model of the proposed approach and 4.2 describes algorithms which are compared with proposed approach. Section 5 first describes the case study of the Micro-IDE tool proposed in our previous work [5], as a subsection. This section basically represents experimental results comparing the proposed fault-tolerant approach and other deployment algorithms. Finally, Section 6 consists of a discussion and concludes the paper.

2. BACKGROUND

In this section, microservice and monolith concept, advantages and disadvantages of these architectures are explained. Besides, the importance of fault tolerance occured during the deployment of microservices to limited capacity cloud servers is mentioned. Then, background information about cloud technologies, virtualization and conteinerization technologies are provided.

2.1. Microservice Concept

Enterprise software systems have hundreds of functions to satisfy a wide range of business needs and they are often built as monolithic applications. As systems and projects grow,they need to scale, collaborate in teams and update applications. In a monolithic application, when an update needs in a part of the system, all system needs to be scaled instead only updated tasks [5]. Because even small changes may require changing the entire application. Tasks like analyzing the entire program and putting it back on the server would be very expensive. Because of these concerns, Service Oriented Architecture is implemented as a solution for deployment and fault tolerant of the systems.

Microservices architecture, which is the advanced form of the Service Oriented Architecture architectural structure, has small, self-contained services with a loosely coupled structure. It has sophisticated methods that build upon the ideas of service-oriented architecture. The microservice architecture is implemented on a single application that combines numerous services that can be independently developed and deployed, each service operates in its own process. The independence of the services from each other makes easier for them to interact over lightweight protocols, distribute, scale and develop. Each microservice should be well coordinated and perform a particular business task or procedure. Because of its modularity, various teams can independently create, test, deploy, and scale specific services, which increases deployment velocity and productivity.

Microservices architecture is a reasonable option for companies looking to increase their operational effectiveness, scalability, and agility. Applications can be divided into modular, independently deployable services to help organizations create a system that is more maintainable, fault-tolerant, and responsive.

2.2. Monolith versus Microservice

2.2.1. Monolithic Architecture

A monolithic architecture is a traditional approach to designing software where an entire application is built as a single and atomic unit. In this architecture, all the different components of the application, such as the user interface, business logic, and data access layer are tightly integrated and deployed together [7]. There are some advantages and disadvantages of monolithic architectures shown in the following:

Advantages of Monolith Architectures:

- Simplicity: All components or services of the application are located in a location when having a monolithic design. This facilitates the comprehension of the inter-relationships between various components of the system [7]. Since developers do not have to concern about how many services connect with one another, this architecture also makes the development process easier.
- Deployment: Since a monolithic application only requires single deployment build, deployment process is not complicated. This feature decreases the possibility of deployment failures and facilitates deployment management. Furthermore, if the system encounters a failure during deployment, it’s simpler to roll back changes because all the code is in a codebase.
- Debugging: Because all functionalities are integrated and in one location, debugging and tracking operations become easier.

Disadvantages of Monolith Architectures:

- Scalability: It can be difficult to scale monolithic applications, particularly when some of their components must manage high traffic volumes. Because the application's components are all closely related, adjusting one of them frequently necessitates adjusting the entire program, which can be expensive and inefficient [8].
- Fault Tolerance: There is no isolation between components in a monolithic architecture. This implies that the application as a whole may crash if one of its parts fails. Because of their lack of fault tolerance, monolithic programs may be more prone to reliability problems and outages [8].

2.2.2. Microservices Architecture

As an alternative to monolith structure, the microservices architecture is constructed as a group of discrete, standalone services, each of which represents a particular business function. These services communicate to each other over a network and they have loosely coupled and high cohesive structure [8]. Each service can be built, deployed, and scaled independently. There are some advantages and disadvantages of microservices architecture shown in the following:

Advantages of Microservices Architecture:

- Scalability: Microservices enable scalability of the components of an application in response to demand. Thus, instead of scaling the entire application, it ensures that only the parts of the application that require more traffic management are scaled.
- Flexibility: From the perspective of work teams, microservices enable the use of different technologies and programming languages for different services. This flexibility enables teams to choose the most appropriate tool for the task, rather than being restricted to a single technology stack.

- Durability: Unlike a single large system, microservices are made up of many smaller, discrete systems, so a failure in one service does not affect the entire system. This property reduces the likelihood of failures in the entire system and increases sustainability of the application.

Disadvantages of Microservices Architecture:

- Complexity: Managing a large number of microservices can be challenging. It can lead to a more complex deployment.
- Cost: Although microservices provide scalability and flexibility, deployment and management of the microservices are challenging and increase cost, especially when it comes to infrastructure and operation.

2.3. Fault Tolerance

Fault tolerance is a fundamental concept that ensures the resilience of systems, particularly in microservice architectures [9]. It addresses various potential sources of failures, offering the system the ability to withstand disruptions and maintain uninterrupted service. It ensures that the system can continue to operate and provide service even when unexpected failures occur. The importance of fault tolerance cannot be ignored, given the various challenges and vulnerabilities of modern systems. These challenges include hardware failures [10][11], software glitches [10][12], network problems [12], and human errors [12]. In microservice architectures, numerous services operate independently but collectively contribute to the overall system functionality. For this reason, applying fault-tolerant approaches in these architectures is significant to ensure system performance. The importance of fault-tolerant systems becomes apparent when we consider that the failure of a single microservice can have a cascading effect, potentially causing the entire system to crash. This issue highlights the critical need for proactive measures to protect the system against these failures. In traditional deployment methods, achieving fault tolerance required in modern and complex systems can be challenging. Innovative solutions are essential to address this challenge.

The real-time nature of today's systems underscores the importance of incorporating fault tolerance from the initial design phase [13]. By embracing fault tolerance from the design phase, we can build systems better equipped to handle unexpected challenges/failures, thus minimizing risks and improving overall reliability [11][14]. Hereby, the minimum level of risk can be achieved in real-time.

2.4. Importance of Design Phase in Software Development Lifecycle

Due to its fundamental role in creating a robust and reliable system, the design phase is important for the fault tolerance of microservices.

- Service Boundaries: Well-defined design prevents other microservices from being affected when one microservice fails.
- Redundancy: Redundancy decisions are made to ensure that failures are handled correctly.
- Load balancing: Load balancing deploys traffic evenly to prevent any service from becoming congested.
- Service Discovery: In case of a failure, scheduling dynamic service registration and inspection enables services to adjust and reroute traffic.

2.5. Cloud Technologies

Cloud computing is a system that maintains data and applications via central remote servers and the internet. Users and companies can access their personal files from any computer with network connectivity and use applications without having to install them. This technology centralizes memory, storage, processing, and bandwidth enabling significantly more efficient computing.

From a microservice deployment perspective, cloud computing offers a stable environment for creating, growing, and overseeing applications as an assembly of loosely coupled,

independent services. Developers are able to create microservices with customized resource requirements by utilizing the cloud's flexible infrastructure models, Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). This allows efficient scaling, fault isolation, and rapid deployment. Through the use of tools such as Kubernetes, Docker and other cloud platforms also facilitate essential functions like monitoring, service discovery which encourage smooth service-to-service communication and guarantee efficient server resource usage. However, this decentralized model requires addressing challenges like data consistency, security, and inter-service communication making cloud-native principles crucial for successful microservice architecture deployment.

2.6. Virtualization

Virtualization is the process of transforming physical computing resources—like servers, operating systems, storage units, or network components into a virtual representation. By altering traditional computing settings, virtualization aims to improve workload management and increase scalability, efficiency, and cost effectiveness. This idea is widely applicable to a variety of contexts, such as server virtualization, hardware-level virtualization, and operating system virtualization.

2.7. Containerization

Containers encapsulate programs within isolated operating environments, including all the necessary files and libraries presenting a lightweight alternative to traditional virtual machines. Although containers share the user space of the host system, each retains its distinct set of system processes, environment variables, and libraries.

The advantages of containerization are substantial, offering rapid deployment, improved portability, and superior scalability. This technology enables developers to create consistent environments that are insulated from other applications, minimizing the risk of system instability and application conflicts. By packaging software along with all required

dependencies, containers can seamlessly operate on any system that runs a container engine, regardless of its specific configuration.

2.7.1. Docker and Kubernetes

Docker is an open-source platform to build, deploy, and manage containerized applications. Docker containers can be deployed almost anywhere because each one is independent of each other in terms of the resources and configurations needed to execute the microservice. Scaling microservices across numerous servers causes cost issues and operational complexity. Then, Kubernetes is used as a complement to Docker to resolve complexities.

Kubernetes is an open-source container orchestration platform that automates most of the manual tasks that are involved in deploying, managing, and scaling containerized applications. Docker and Kubernetes work well together to build a stable and adaptable foundation for microservice-based systems, which can be expanded to meet varying workload demands without introducing additional operational complexity.

3. RELATED WORK

The investigation of fault-tolerant mechanisms in microservices architecture has emerged as a prominent focus in the literature. Especially in the design phase of a microservices architecture, creating the system by adopting a fault-tolerant approach provides great convenience to the developer in the application development and implementation process. Integrating fault-tolerant strategies into the design, development and deployment processes enhances system reliability and resilience [15–18]. Although fault-tolerant frameworks are not commonly proposed during the design phase of the software development lifecycle (SDLC), various state-of-the-art approaches have been proposed to guarantee fault tolerance throughout the application creation and implementation phases.

Power and Kotonya [19] propose a microservices architecture tailored for IoT systems, featuring a plug-and-play framework aimed to enhance fault tolerance. A microservice in this architecture deals with real-time fault detection by leveraging complex event processing and the other microservices conduct online machine learning to detect and prevent errors of predefined fault models. These two complementory microservices help to improve the reliability of IoT systems by enabling real-time error detection and prevention of the errors using machine learning. Hence it facilitates to foster a safer and more dependable ecosystem.

Rasheedh and Saradha [20] focus on the Reactive Microservices (RM) architecture, offering insights into creating flexible and scalable systems. Their study emphasizes individual probabilities within the reference architecture of each microservice component, facilitating independent development, publication, organization, scaling, upgrading, and decommissioning. The study emphasized the benefits of RM architecture such as reduced development time, increased agility, improved fault tolerance, but mentioned that this architecture should be designed differently for each system requirement.

Zilic et al. [21] proposed a method for predicting faults from microservice architectures using a machine learning approach. The method is developed in the framework using Support Vector Regression (SVR) and Markov Decision Process (MDP) to provide a fault monitor

and service availability prediction tool to reduce system failures. Mihai et al. [22] proposed a client-server communication model to tolerate possible failures due to the system's variable workload, emphasizing the integration of dynamic microservices. Huff and Hiltunen [23] propose techniques that use state partitioning, partial replication, and fast re-route with role awareness to enable fault tolerance in xApps within the RIC (Radia Access Network Intelligent Controller) platform while preserving scalability.

Javed and Heljanko [24] propose CEFIoT architecture for IoT applications, leveraging cloud technologies for fault tolerance. To solve the data fault tolerance problem, the authors implemented high-performance data replication in edge and cloud technologies using the Apache/Kafka publish/subscribe platform. In addition, with the use of Kubernetes which is a microservice deployment infrastructure, automatic reconfiguration is performed by handling both hardware and network connection errors. In another study, Flora et al. [25] investigate the effectiveness of Kubernetes in dealing with faults and aging in microservices, and on the possibility of using faults to accelerate aging effects for testing purposes.

Wu et al. [26] conduct a systematic and comprehensive explanation of different fault types, their causes and various fault tolerance approaches are used in cloud. The study presents a broad survey of various fault tolerance frameworks in the context of their basic approaches, fault applicability, and other key features. A comparative analysis of the surveyed frameworks is also included in the study. Besides, the authors propose an extensible fault tolerance testing framework for microservice-based cloud applications based on the non-intrusive fault injection that can be customized and executed to verify the manner and performance of the fault tolerance of the target service.

Li et al. [27] proposes a reliability model of microservice-based cloud application by using predicate Petri net decision-making model [28]. Based on the constructed microservice reliability model, the correctness of predicate Petri net modelling and the effectiveness of the strategies are proven theoretically. Moreover, a formal description language is defined to model microservice, user request, and container accurately, and a reliability analysis is

conducted to measure a critical microservice's fluctuation and vibration attributes within a period, and the related properties of the constructed model are analyzed.

Jia et al.[9] propose a novel fault-tolerant mechanism for IIoT Edge based on the stateful microservices characteristics, using causal logging and distributed checkpoint algorithm, achieves exactly-once guarantees and has less impact on service performance compared to other methods. The proposed method mainly focuses on causal logging and distributed checkpoint algorithm. This fault recovery mechanism utilizes causal logging to record the non-deterministic events, and completes the state recovery of microservices by loading checkpoint and replaying log records.

Richter et al. [29] the general properties of a microservice architecture and its dependability with reference to the legacy system and requirements for an equivalent microservice-based system. Besides the migration process, services and data containerization, communication via message queues are evaluated by achieving fault tolerance and high availability with the help of replication inside the resulting architecture.

According to the our investigation, ensuring fault tolerance in microservices architectures is usually studied on deployment, development and implementation phases. This thesis contributes practical information on fault-tolerant system development and efficient deployment approaches, particularly during the design phase. In this phase, both modeling and deployment of microservices are performed, with provisions for backup servers to handle potential failures, thereby enhancing system resilience. A comparison of related studies and the proposed study in terms of technical features is given in Table 3.1.

Table 3.1 Comparison of fault-tolerant approaches that are used for microservices in the literature and this thesis

Study	Tool/ Framework	Approach	Phase of SDLC	Model-driven architecture
Richter et al. [29] (2017)	✗	Replica of the services	Operation	✗
Power and Kotonya [19] (2018)	✓	Machine learning-based	Implementation	✗
Javed and Heljanko [24] (2018)	✓	High-performance data replication	Deployment	✗
Wu et al. [26] (2018)	✓	Non-intrusive fault injection	Test	✗
Mihai et al. [22] (2019)	✓	Client-server communication	Deployment	✗
Rasheedh and Saradha [20] (2021)	✗	Reactive technique	Development	✗
Li et al. [27] (2021)	✓	Redundancy operation	Deployment	✗
Huff and Hiltunen [23] (2021)	✓	State partitioning, partial replication, and fast re-route	Implementation	✗
Zilic et al. [21] (2022)	✗	Markov Decision Process Support Vector Regression	Test	✗
Flora et al. [25] (2022)	✗	Fault injection	Test	✗
Jia et al.[9] (2023)	✗	Causal logging and distributed checkpoint algorithm	Deployment	✗
This study (2024)	✓	Occupancy limitation, back-up services, allocation algorithms	Design	✓

4. FAULT-TOLERANT DEPLOYMENT APPROACH FOR DESIGN PHASE

Fault-tolerant approach allows the implementation of proactive measures to prevent failures and ensure continuous operation of the sysrem rather than relying solely on reactive measures after failures arise.

As a first principle, general algorithmic approach design needs to be decided whether microservices are deployed to all existing servers or not. If not, some servers are reserved as cold start nodes [30]. The second principle is to try to deploy microservices with high communication costs on the same nodes. Thus, the cost value can be minimized. This proposed algorithmic approach is expressed in Figure 4.1 as a flow diagram for the design phase.

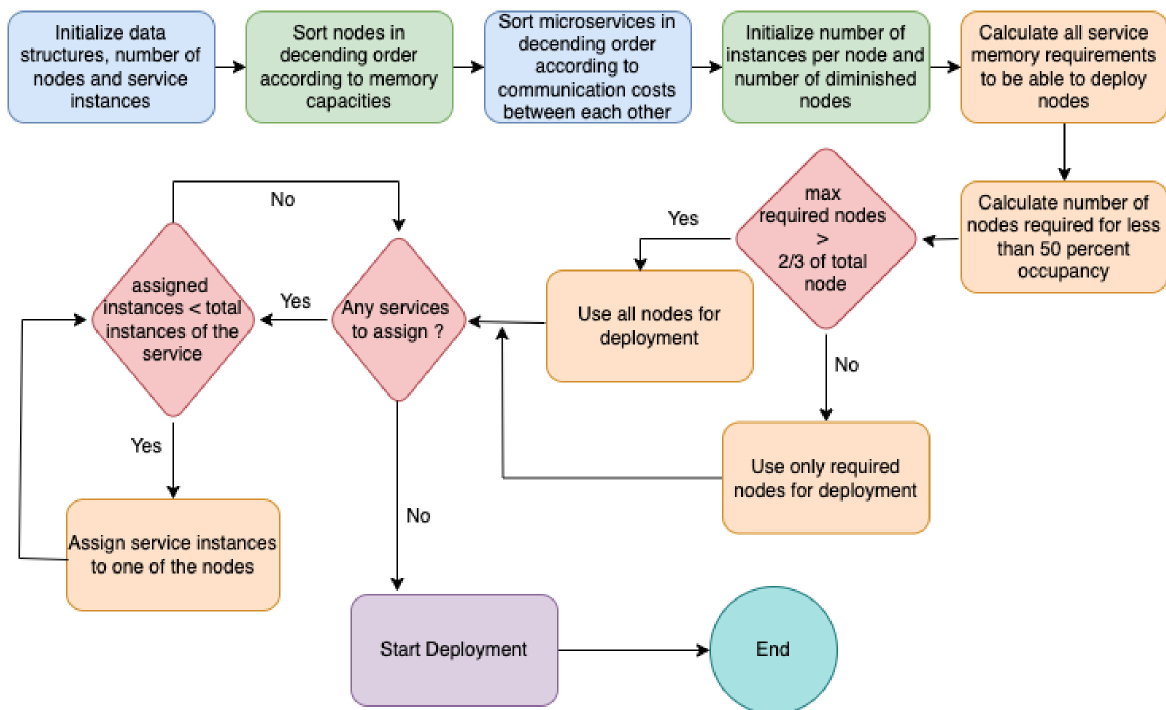

Figure 4.1 Flow diagram of the proposed fault-tolerant approach

The proposed approach initializes the necessary data structure and the relevant variables to define the available nodes (servers) and microservices. The next step optimizes server

Algorithm 1 Fault-Tolerant Microservice Deployment Approach

```
1: function DEPLOYMICROSERVICES
2:     Initialize solutionMap, nodes, serviceInstances
3:     Sort nodes by memoryCapacity descending
4:     Sort microservices by communicationCosts descending
5:     Initialize instancesPerNode and diminishedNodes
6:     Calculate all microservice memory requirements
7:     if ENOUGH_RESOURCES_FOR_ALL_SERVICES then
8:         if (NUMBER_OF_NODES_REQUIRED < 2/3 · totalNodes) then
9:             maxAssignedNodes = numberOfNodesRequired
10:        else
11:            maxAssignedNodes = totalNodes
12:        end if
13:    else
14:        Cancel deployment and return
15:    end if
16:    START_DEPLOYMENT
17: end function
18:
19: function ENOUGH_RESOURCES_FOR_ALL_SERVICES
20:    Calculate the totalRequiredMemory for all microservices
21:    if (totalRequiredMemory > totalMemoryCapacity) then
22:        return false
23:    else
24:        return true
25:    end if
26: end function
27:
28: function NUMBER_OF_NODES_REQUIRED
29:    Calculate the number of nodes needed for 50% memory occupancy
30:    return numberOfNodesRequired
31: end function
32:
33: function START_DEPLOYMENT                          ▷ Actual deployment logic
34:    Calculate the maximum number of instances to be allocated to a node
35:    for all microservices with descending order of communication cost do
36:        for node ← 1 to maxAssignedNodes do
37:            Assign the maximum number of instances possible
38:        end for
39:    end for
40: end function
```

allocation by sorting the nodes in descending order based on their memory capacities and the microservices based on communication costs between each other. The total number of nodes and microservices is calculated to establish the scale of the problem. The maximum amount of microservice instances that can be deployed to each node is also calculated. Besides, the ratio of decreased nodes are defined as 2/3 of the total number of nodes. The memory capacity required to deploy all services to nodes is measured. The memory capacity of the node is checked whether the decreased nodes will be sufficient for all microservice instances with a maximum occupancy of 50%. If the capacity is sufficient, deployment can be performed using only decreased nodes, otherwise all nodes are used. In order to start the deployment process, all microservice instances are assigned to the specified nodes, and finally the deployment process is started. Algorithm 1 shows the pseudo-code that represents the flow diagram.

4.1. Mathematical model of the proposed approach

In this section, a mathematical model of the proposed approach is provided as an enhanced Capacitated Task Assignment Problem (CTAP). The proposed fault-tolerant approach is formulated in six steps. Here is a breakdown of each component and parameter:

1. Definitions and parameters:

    - **S** $= \{s_1, s_2, \ldots, s_m\}$: set of microservices where s_i is i^{th} microservice and m is the number of microservices
    - s_i^j: j^{th} instance of microservice s_i
    - k_i: the number of instances for microservice s_i
    - mem_i: amount of memory needed for microservice s_i
    - **N** $= \{n_1, n_2, \ldots, n_p\}$: set of nodes where n_i is i^{th} node and p is the number of nodes
    - M_n: total amount of memory available on node n

- c_{ij}: the communication cost of s_i and s_j if they are assigned to different nodes. Communication cost is negligible if two microservices are assigned to the same node.
- C: the set of communications between microservices whereby each communicating service combination (i, j) is included in this set.
- th_m: The allocation ratio of the total memory of a node. The proposed algorithm tries to allocate for service instances of microservices to nodes with respect to this threshold. This value is selected as $1/2$ in our current model, which means 50% of memory capacities of nodes will be used.
- th_n: The occupancy ratio of the nodes aimed at the start of allocation of microservices. This value is selected as $2/3$ in our current model, which means that 2/3 of all nodes will be used with th_m allocation ratio to deploy microservices if they have enough memory capacity. In this case, the unused 1/3 of all nodes are used as cold start servers. If the capacity of 2/3 of all nodes is not enough for allocation of microservices, all nodes are used for allocation.

2. Total Minimum Cost: This is the objective function to be minimized. It is a summation over pairs of services, taking into account the communication costs between services when they are not placed on the same node.

3. a^j_{in}: The decision variable representing the assignment of service instances to nodes. a^j_{in} represents the assignment of s^j_i to node n. Each service instance is assigned to exactly one node.

4. The memory requirement for any service instance must be non-negative.

5. The constraint ensures that the sum of the memory requirements of all instances assigned to node n must be less than or equal to M_n.

6. The fault tolerance constraint ensures that the number of service instances assigned to a node can not exceed a certain ratio.

The minimum cost of the fault tolerant deployment system achieved by using these parameters is formulated as a mathematical model as follows:

Total minimum cost=

$$\sum_{(i,x)\,\in\,C}\ \sum_{j=1}^{k_i}\sum_{y=1}^{k_x}\sum_{n=1}^{f(x)} a_{in}^{j}\ (1-a_{xn}^{y})\, c_{ix} \qquad (1)$$

Subject to,

$$a_{in}^{j} = \begin{cases} 1, & if\ \mathrm{s}_i^j \text{ is assigned to node } n \in N \\ 0, & otherwise \end{cases} \qquad (2)$$

$$f(x) = \begin{cases} p \times th_n, & \sum_{i=1}^{m}\sum_{j=1}^{k_i} mem_i \le\ th_m \sum_{n=1}^{p\times th_n} M_n \\ p, & otherwise \end{cases} \qquad (3)$$

$$\sum_{j=1}^{k_i}\sum_{n=1}^{f(x)} a_{in}^{j} = k_i, \text{ for all } s_i \in S \qquad (4)$$

$$\sum_{i=1}^{m}\sum_{j=1}^{k_i} mem_i\ a_{in}^{j} \le M_n, \text{ for all } n \in N, mem_i > 0 \qquad (5)$$

$$\sum_{j=1}^{k_i} a_{in}^{j} \le \frac{k_i}{f(x)}\ , \text{ for all } s_i \in S, n \in N \qquad (6)$$

The provided equation represents a mathematical model for optimizing the deployment of microservices within a fault-tolerant system to minimize costs. The explanations of the various components of the equation and their significance are described as follows:

1. **Objective Function (Total Minimum Cost)**:

$$\sum_{(i,x)\,\in\,C}\ \sum_{j=1}^{k_i}\sum_{y=1}^{k_x}\sum_{n=1}^{f(x)} a_{in}^{j}\ (1-a_{xn}^{y})\, c_{ix} \qquad (1)$$

   Equation 1 calculates the total communication cost between service instances that are deployed on different nodes.

2. **Decision Variable**:

$$a_{in}^{j} = \begin{cases} 1, & if\ s_i^j \text{ is assigned to node } n \in N \\ 0, & otherwise \end{cases} \quad (2)$$

a_{in}^{j} is a variable that indicates whether a particular instance of a service is assigned to a given node. If it is assigned a particular node, the variable is 1; otherwise, it is 0.

3. **Function** $f(x)$:

$$f(x) = \begin{cases} p \times th_n, & \sum_{i=1}^{m} \sum_{j=1}^{k_i} mem_i \leq\ th_m \sum_{n=1}^{p \times th_n} M_n \\ p, & otherwise \end{cases} \quad (3)$$

Equation 3 describes the function $f(x)$ which determines the number of node to be used for the deployment. If th_m ratio (50% percent in our experiments) of total memory of $p \times th_n$ nodes (2/3 of all nodes in our experiments) are enough to deploy all microservices, $p \times th_n$ nodes will used for deployment and the remaining $p \times (1 - th_n)$ nodes will be reserved as cold start nodes. Otherwise, all nodes will be used for the deployment.

4. **Assignment Constraint**:

$$\sum_{j=1}^{k_i} \sum_{n=1}^{f(x)} a_{in}^{j} = k_i, \text{ for all } s_i \in S \quad (4)$$

Equation 4 as a constraint ensures that each service s_i is assigned to nodes in such a way that the number of deployed instances are less than the total number of s_i instances.

5. **Memory Constraint**:

$$\sum_{i=1}^{m} \sum_{j=1}^{k_i} mem_i \, a_{in}^{j} \leq M_n, \text{ for all } n \in N, mem_i > 0 \qquad (5)$$

Equation 5 as a constraint ensures that the total memory requirements of service instances assigned to node n does not exceed the node's memory capacity M_n. This equation also ensures that the memory requirement for any service instance is a non-negative value.

6. **Fault Tolerance Constraint**:

$$\sum_{j=1}^{k_i} a_{in}^{j} \leq \frac{k_i}{f(x)} \text{ , for all } s_i \in S, n \in N \qquad (6)$$

Equation 6 as a constraint ensures that the number of instances of s_i assigned to a node n can not exceed $\frac{k_i}{f(x)}$ value.

Overall, the proposed model is designed to assign service instances to nodes in such a way that the total communication cost is minimized while ensuring that the memory capacity of each node is not exceeded and each service instance is properly assigned according to the defined constraints.

4.2. Comparative Microservice Assignment Algorithms

In this section, algorithms such as Best Fit, Next Fit, Hungarian, and Round Robin among various allocation strategies have been studied for their effectiveness in allocating resources and handling errors. These algorithms are briefly described in the following subsections.

4.2.1. Best Fit

Best fit algorithm basically minimizes servers' (node) resource wastage by allocating the best location to the available memory capacity of the microservice instances to the appropriate partitions in the nodes[31]. It iterates through the available resources and selects a hole that

is equivalent to the capacity of the incoming microservice instance in order to make optimal use of the nodes' memory.

Algorithm Description:

- Initialization: The algorithm starts with an empty set of nodes and services.
- Node Selection: Iterate through the available resources and select the node with the best fit for the incoming microservice instances.
- Service Allocation: Allocate the microservice instances to the selected node.
- Update Node Capacity: Update the remaining capacity of the selected node.
- Repeat: Continue the process until all microservice instances are allocated into nodes.

Application in Microservice Deployments:

In microservice deployments, the Best-Fit Algorithm can be used to allocate microservice instances running on different nodes. By selecting the node with the best fit, the algorithm optimizes node utilization and minimizes the risk of resource consumption.

Fault Tolerance Analysis:

In scenarios where one of the nodes may fail or become unavailable, the algorithm may need enhancements to dynamically redeploy microservice instances and maintain system stability.

4.2.2. Next Fit

Next Fit algorithm is similar to the Best Fit algorithm that simplifies node allocation by selecting the next available node that can accommodate the incoming microservice without consuming the search [31]. Unlike Best Fit algorithm, Next Fit allocates the first hole that is big enough and start search always from the last allocated hole. The algorithm can be applied to efficiently allocate incoming requests to instances of a microservice based on their

availability. By avoiding exhaustive search, the algorithm reduces computational overhead and improves system responsiveness.

Algorithm Description:

- Initialization: The algorithm starts with an empty set of nodes and incoming microservice instances.
- Node Selection: Iterate through the available nodes and select the next available node that can accommodate the incoming microservice instance.
- Service Allocation: Allocate the incoming microservice instance to the selected node.
- Update Node Capacity: Update the remaining capacity of the selected node.
- Repeat: Continue the process until all microservice instances are allocated.

Application in Microservice Deployments:

In microservice deployments, the Next Fit algorithm provides a balance between node utilization and computational complexity. The algorithm optimizes node utilization and simplifies the allocation process by selecting the next from the best hole that fits the capacity of the incoming service instance.

Fault Tolerance Analysis:

Similar to the Best Fit algorithm, the fault tolerance of the Next Fit algorithm relies on the resilience of the underlying infrastructure. Enhancements such as redundancy and failover mechanisms can improve the algorithm's ability to handle faults and failures.

4.2.3. Hungarian

The Hungarian algorithm is an optimization algorithm used for solving assignment problems in bipartite graphs. In the context of microservice deployments, it can be adapted to

optimize node allocation and service assignment [32]. When we consider a scenario where multiple microservices compete for nodes in a shared environment, the system can dynamically allocate nodes to microservices based on their individual needs using this algorithm optimizing node utilization and improving fault tolerance.

Algorithm Description:

- Initialization: The algorithm starts with an empty assignment matrix representing node-service assignments.
- Execution Cost Matrix: Create a cost matrix representing the execution cost of assigning each microservice instance to each node.
- Assignment: The algorithm considers the assignment costs between workers and jobs and iteratively runs to match these pairs at the lowest cost by aiming to find the optimal solution.
- Service Allocation: Allocate microservice instances to nodes based on the optimal assignment.
- Update Node Capacity: Update the remaining capacity of the allocated nodes.
- Repeat: Continue the process until all microservice instances are allocated.

Application in Microservice Deployments:

In microservice deployments, the Hungarian Algorithm can optimize node allocation by considering various factors such as microservice requirements, node capabilities, and network latency. By finding the optimal assignment, the algorithm improves node utilization and system performance.

Fault Tolerance Analysis:

The fault tolerance of the Hungarian Algorithm depends on its ability to adapt to changes in node availability and microservice instance requirements. By continuously reevaluating the

assignment based on dynamic conditions, the algorithm can mitigate the impact of faults and failures on system performance.

4.2.4. Round Robin

The Round Robin Algorithm is a simple scheduling algorithm that allocates nodes in a circular fashion, ensuring fairness and equal deployment of microservice instances among available nodes [33].

Algorithm Description:

- Initialization: The algorithm starts with an empty set of nodes and incoming microservice instances.
- Node Selection: Iterate through the available nodes in a circular fashion.
- Service Allocation: Allocate incoming microservice instances to the selected node.
- Update Node Capacity: Update the remaining capacity of the selected node.
- Repeat: Continue the process until all microservice instances are allocated.

Application in Microservice Deployments:

In microservice deployments, the Round Robin Algorithm provides a straightforward approach to node allocation, ensuring that each node receives an equal share of microservice instances over time. While it may not optimize node utilization as effectively as other algorithms, it promotes fairness and prevents node starvation.

Fault Tolerance Analysis:

The fault tolerance of the Round Robin Algorithm stems from its ability to evenly deploy microservice instances among available nodes. By maintaining a balanced workload, the algorithm reduces the risk of individual nodes becoming overloaded or failing due to excessive load.

4.2.5. Proposed Fault-Tolerant Algorithm

The proposed fault tolerance approach introduces a novel strategy for microservice deployments that prioritizes fault tolerance and communication cost optimization. Considering both occupancy and node availability, the algorithm aims to ensure efficient node utilization while maintaining system stability against the faults. The proposed approach differs from the approaches in the literature in that it minimizes the communication cost between services by reducing the resource utilization of nodes during the design phase of the software development lifecycle.

Algorithm Description:

- Initialization: The proposed algorithm starts with an empty set of nodes and incoming microservice instances.
- Occupancy Check: Calculate the occupancy of each node by dividing the total number of deployed microservice instances by the total capacity of the node.
- Node Availability: Determine the availability of nodes based on the specified threshold (e.g., 50%).
- Deployment Decision:

  If the occupancy of all nodes is less than the threshold (e.g., 50%), deploy all microservice instances to only 2/3 of the total nodes.

  If the occupancy of any node exceeds the threshold, deploy all microservice instances to all nodes in a round-robin fashion.
- Service Allocation: Allocate incoming microservice instances to the selected nodes based on the deployment decision.
- Update Node Capacity: Update the remaining capacity of the selected nodes.
- Repeat: Continue the process until all microservice instances are allocated.

Advantages:

- Fault Tolerance: By dynamically adjusting the deployment strategy based on node occupancy, the algorithm enhances fault tolerance by avoiding overloading individual nodes.
- Communication Cost Optimization: By deploying microservice instances to a subset of nodes when occupancy is low, the algorithm reduces communication overhead and improves network efficiency.

Application in Microservice Deployments:

In microservice deployments, fault-tolerant approach provides a flexible and adaptive solution for managing node allocation and system stability. By considering both occupancy and node availability, the algorithm ensures optimal node utilization while minimizing the risk of node failures and communication bottlenecks.

5. EXPERIMENTAL RESULTS

To evaluate the proposed approach, an online book shopping case study consisting of four services is implemented. A simulation environment is used to analyze for different algorithms and proposed approach. The content of the experiments performed are explained in the following subsections.

5.1. Case Study: Online Book shopping

In this section, a case study is designed to test the proposed fault-tolerant approach. In this case study, different numbers of microservice examples are given for four microservices: Account, Order, Book Inventory and Shipping which were obtained by designing an online book-shopping platform using microservice architectures. In the selected microservice architecture, different types of data, including User, Customer, Seller, CardDetails, Cancellation, CCDetails, PaymentSystem and Transaction are exchanged among microservices. To ensure fault-tolerant deployment of microservices by minimizing total minimum communication cost, an automated fault-tolerant system is crucial to deploy microservices to the limited capacitated servers. The communication cost of an algorithm can be evaluated using the relationships between the data represented in the Microservice Data Exchange Metamodel and the services described in the Microservice Definition and Communication Metamodel using the Micro-IDE tool [5] are presented in Table 5.1. There is a publish-subscribe relationship between microservices, especially between Order and Shipping services with higher communication cost than others.

To deploy microservices, a physical infrastructure should be built. In this case study, we created a physical infrastructure model comprising four or more servers. Each of these servers has varying processors and memory capacities. The servers are interconnected through a local area network connection. Using the Micro-IDE tool developed in our previous work [5][6], users could define a desired set of servers and connections with

Table 5.1 Sample publish/subscribe relation schema for the case study

Message name	Published Microservice	Subscribed Microservice	Publish rate (Hz)
User	Order Service	Shipping Service	5
Payment Service	Account Service	Book Inventory	3
Cancellation	Order Service	Shipping Service	4
Card Details	Book Inventory	Account Service	2
Customer	Order Service	Shipping Service	3
Transaction	Account Service	Order Service	2

different LAN/WAN configurations. Figures 5.1 and 5.2 illustrate the mentioned infrastructure model.

The microservice runtime execution configuration metamodel includes several properties for the MSA design phase such as the number of instances, related microservices, and memory requirements. Additionally, the publication and execution cost classes are attributes of a microservice instance, which determine the frequency of data updates and the cost of executing a microservice instance on each server. Figure 5.1 shows the properties of four services, the execution cost of the services on each server, and the update rate of the services in each second created for the case study.

Each of the four microservices has 700 and 1000 instances. The required memory value was determined as 10 MB, and the cost of deploying it to each node was determined as 10 MB. Initially each of the six nodes was created to have the same 16000 MB memory capacity, then changed unidentical on second experiment.

As seen in Figure 5.1, it is aimed to conduct an experiment through the online book shopping system. This experiment uses four microservices named Account Service, Book Inventory Service, Order Service, and Shipping Service with several instances.

Figure 5.2 shows the physical resource model where the services will be assigned. Initially, a total of six identical nodes are created. The properties of the nodes may vary depending

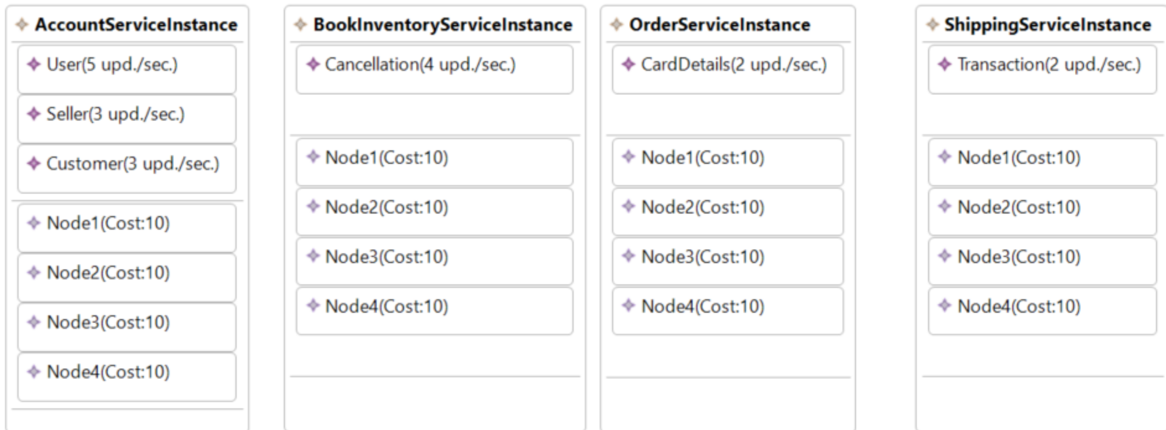

Figure 5.1 Created microservices for the online shopping case study

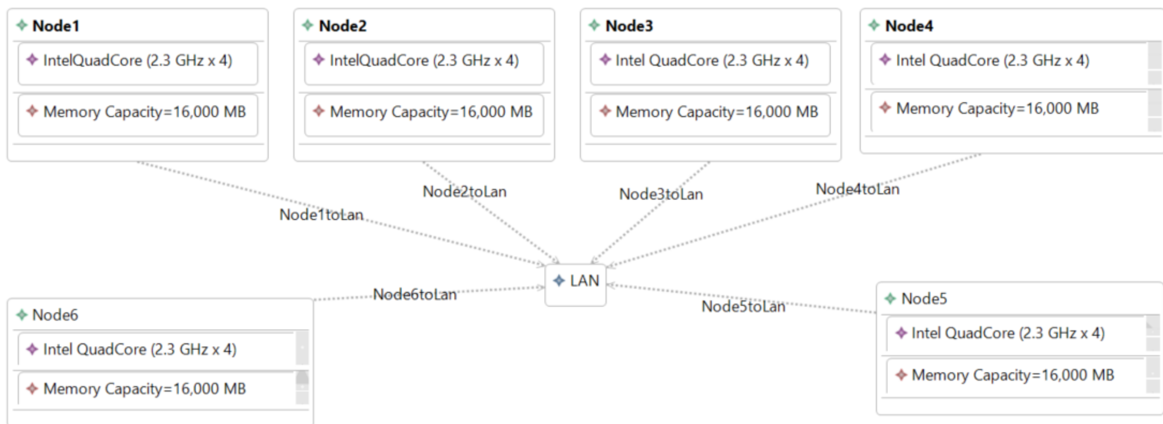

Figure 5.2 Identical processors created for the experiments

on the experiment. Each parameter change is listed in each experiment. In addition, each experiment has two different conditions. One of them is less than 50% occupancy of nodes and the other one is more than 50% occupancy of nodes. When assigning to nodes, we aimed to minimize the total communication cost by assigning frequently interacted microservice instances to the same node in balance. Three experiments are performed, and the results of one parameter change are observed in each experiment. The total communication cost of the proposed fault-tolerant approach is compared to the other four algorithms.

5.2. Simulation Environment

In this thesis, experiments are performed in a simulation environment named the Micro-IDE tool [5] by implementing an online book shopping case study. As explained in the case study section, four microservice types and six servers are created, and the proposed fault-tolerant deployment algorithm is compared with the Best Fit [31], Next Fit [31], Hungarian [32] and Round Robin [33] algorithms by changing certain parameters. A total of three experiments are performed according to identical parameter values, different capacities of nodes and different amounts of service instances respectively. The communication cost efficiency of the created fault-tolerant structure is observed compared to the other task allocation algorithms. The properties of the services and nodes are identified as identical in the first experiment. The following sections explain experiments to evaluate the proposed fault tolerant algorithm efficiency.

5.3. Experiment 1: Identical Microservices and Nodes

In experiment 1, we examined the communication cost improvement rates where values of all parameters are equal for microservices and nodes for five algorithms named Best Fit, Next Fit, Hungarian, Round Robin and proposed the Fault-Tolerant Algorithm.

Table 5.2 Number of microservices and capacities of each instances during deployment for all algorithms

Service Name	Number of Service Instance		Memory Capacities Per Instance(MB)
	Occp. $\leq 50\%$	Occp. $> 50\%$	
Account Service	700	1000	10
Book Inventory	700	1000	10
Order Service	700	1000	10
Shipping Service	700	1000	10

Table 5.2 shows how many instances each service has above and below 50% occupancy and the cost of each instance.

5.3.1. Occupancy Less Than 50%

The deployment of services to nodes using the proposed fault tolerance algorithm with memory occupancy less than 50% is shown in Figure 5.3. Each service is observed to be deployed equally among the four servers. It tends to assign services with high communication costs to the same node.

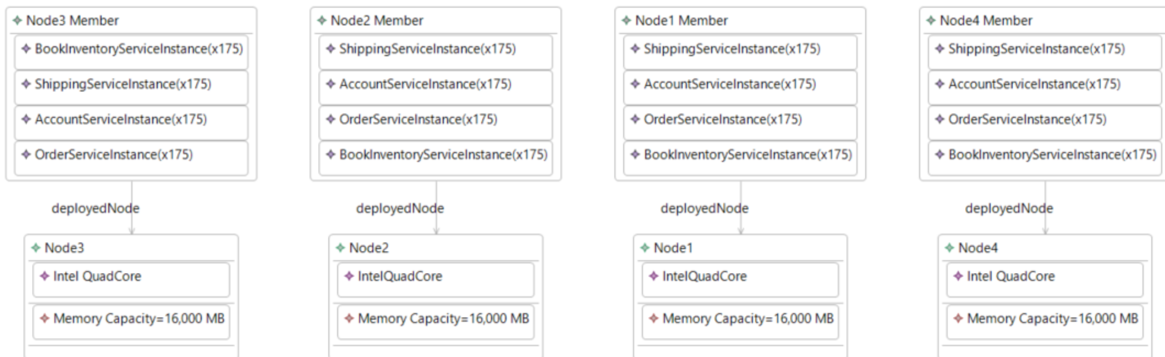

Figure 5.3 Deployment structure using Fault Tolerant algorithm

Figure 5.4 shows model diagram of Round Robin which fault-tolerant approach is resulted as best rates against the other algorithms in the occupancy less than 50% of first experiment. As can be seen in the figure, when the deployment is performed with occupancy below 50%, the proposed approach aims to reduce the communication cost as well as ensuring a fault tolerant deployment by using fewer nodes compared to the Round Robin algorithm. In Table 5.3, an improvement of 10% is achieved.

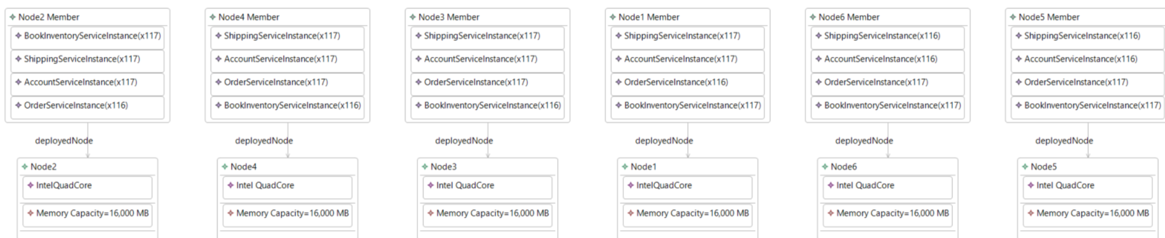

Figure 5.4 Deployment structure using Round Robin

Table 5.3 Improvement rates and total cost values of the algorithms below 50% node occupancy in identical situation

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	2	84.38	3.63
Next Fit	2	84.38	3.63
Hungarian	6	90.28	9.94
Round Robin	6	90.34	10.00
Proposed Approach	4	81.31	

5.3.2. Occupancy More Than 50%

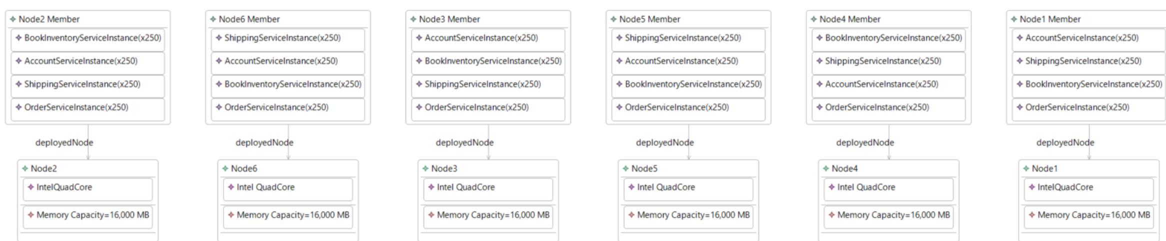

Figure 5.5 Deployment structure using Fault Tolerant algorithm

Figure 5.5 shows the deployment of services to the nodes after deployment using the fault tolerance algorithm with occupancy more than 50%. Services are deployed to six nodes.

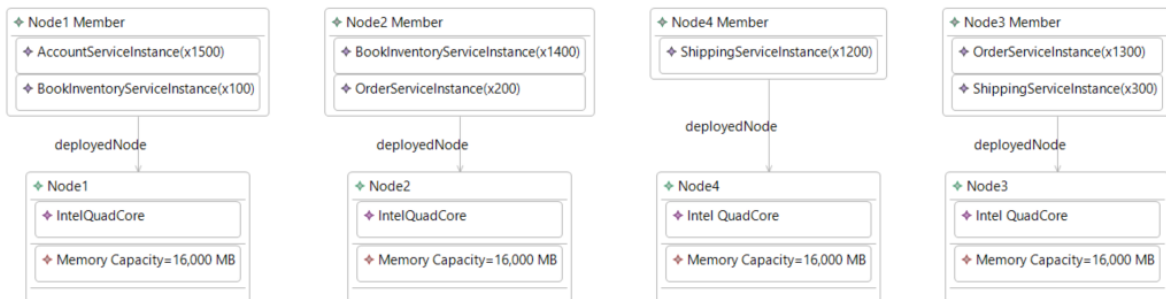

Figure 5.6 Deployment structure using Best Fit

Figure 5.6 shows a deployment model diagram of Best Fit which fault-tolerant approach is resulted as best rates against in the first experiment.

For occupancy above 50%, the method did not provide an advantage over Round Robin, as seen in Table 5.4. On the other hand, in addition to achieving a fault-tolerant deployment by making more sequential deployment to more nodes against the Best Fit and Next Fit algorithms, it achieved a 27.3% improvement in communication cost.

Table 5.4 Improvement rates and total cost values of the algorithms above 50% node occupancy in identical situation

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	4	205.53	27.33
Next Fit	4	205.53	27.33
Hungarian	6	149.11	-0.15
Round Robin	6	149.34	0.00
Proposed Approach	6	149.34	

Tables 5.2, 5.3, and 5.4 demonstrate that the Fault Tolerant algorithm has a relative improvement in communication cost of between 3% and 27% when compared to the four algorithms in the same scenario.

The proposed approach outperforms other traditional deployment algorithms by optimizing the total communication costs in various deployment scenarios. This approach shows a notable advantage against Best Fit algorithm, particularly when node occupancy exceeds the 50% threshold, exhibiting a substantial improvement rate compared to its counterpart. Similarly, in comparison to the Next Fit Algorithm, the fault-tolerant custom approach maintains its superiority, delivering enhanced performance in total communication cost, especially under conditions of high node occupancy. When compared to the Hungarian algorithm, our approach outperforms it significantly, particularly when node occupancy is less than 50%, and demonstrating a notable improvement rate in this setting. Lastly, compared to the Round Robin Algorithm, the fault-tolerant custom approach provides improvement in total communication cost, particularly when node occupancy is below 50%. These consistent improvements emphasizes the efficacy and usability of the fault-tolerant

custom approach in optimizing communication overhead and enhancing network efficiency across diverse deployment environments.

5.4. Experiment 2: The effect of memory capacities of the nodes

In the second experiment, five algorithms are tested with different memory capacities of the nodes. Then, the deployments are compared and the changes in communication cost are observed.

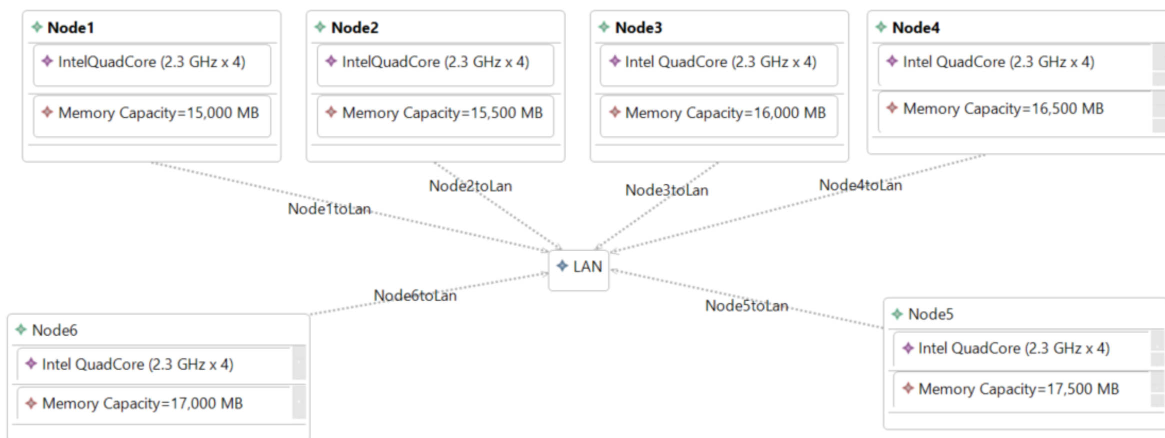

Figure 5.7 The different memory capacities according to the nodes

Figure 5.7 shows a physical resource diagram with different memory capacities of the nodes. The capacities of the nodes have been updated as 15000 MB, 15500 MB, 16000 MB, 16500 MB, 17000 MB, and 17500 MB respectively.

5.4.1. Occupancy Less Than 50%

Figure 5.8 shows how services are deployed to nodes when their capacity changes as well as how the Fault Tolerant algorithm is deployed when less than 50% of the nodes are occupied. Unlike the previous experiment, less server usage is observed thanks to the best combination of servers with sufficient memory capacities.

Figure 5.9 shows a deployment model diagram of Round Robin which fault-tolerant approach is resulted as best rate against in the second experiment.

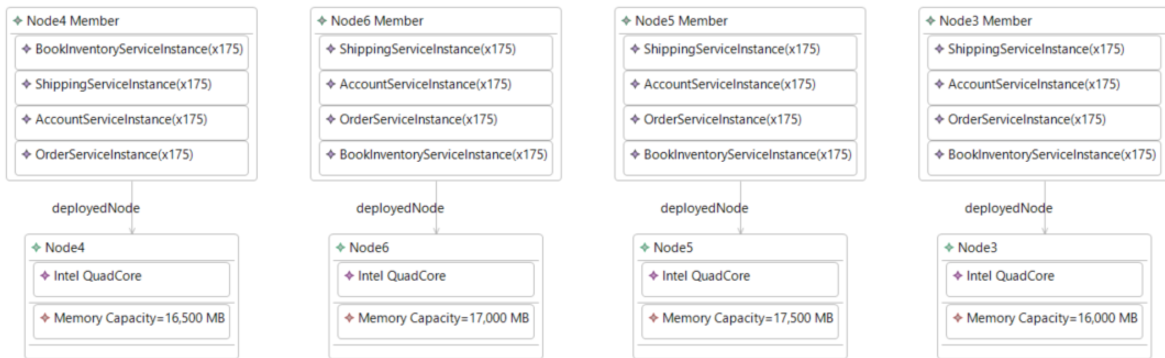

Figure 5.8 Deployment structure using Fault Tolerant algorithm

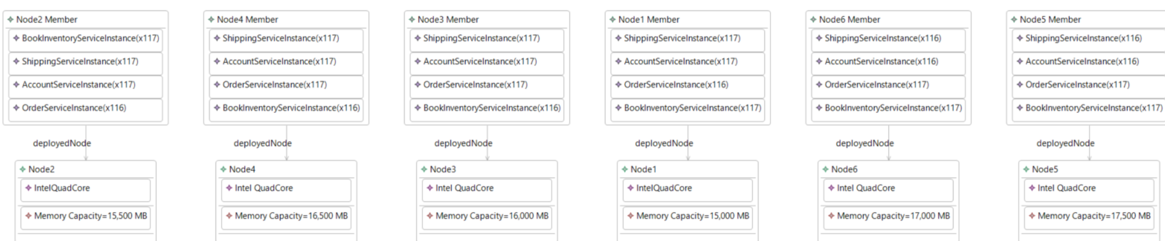

Figure 5.9 Deployment structure using Round Robin algorithm

As can be seen in the figures, when the deployment is performed with less than 50% occupancy, the proposed approach aims to reduce the communication cost as well as providing a fault-tolerant approach, with the same but equal deployment as the Best Fit algorithm and a similar deployment but using fewer nodes than the Round Robin algorithm. As seen in Table 5.5, Best Fit and Next Fit algorithms have less total communication cost compare to the proposed fault tolerant algorithm. However, 9.85% and 10% improvement rates are achieved against Hungarian and Round Robin algorithms respectively.

Table 5.5 Improvement rates and total cost values of the algorithms after the capacity change in the nodes

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	2	79.57	-2.18
Next Fit	2	79.57	-2.18
Hungarian	6	90.20	9.85
Round Robin	6	90.34	10.00
Proposed Approach	4	81.31	

5.4.2. Occupancy More Than 50%

Figure 5.10 shows the deployment of services to the nodes once the capacity changes on the nodes and the deployment using the Fault Tolerance algorithm with more than 50% occupancy. Due to the total capacity of services and to be able to keep node occupancy balanced, all nodes are used.

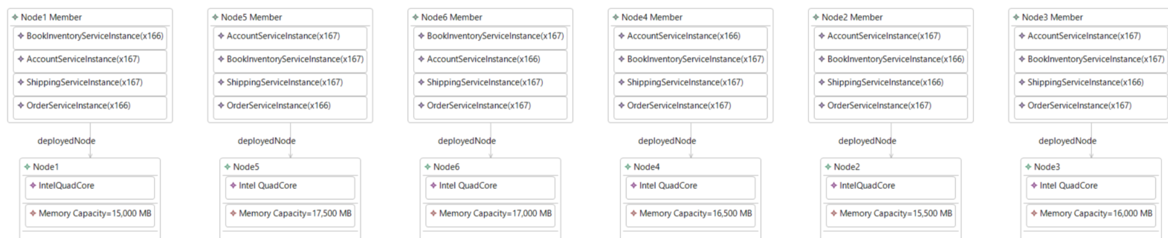

Figure 5.10 Deployment structure using Fault Tolerant algorithm

Figure 5.11 shows model diagram of Best Fit which fault-tolerant approach is resulted as best rate against in the second experiment.

In Table 5.6, for occupancy rates above 50%, the method did not provide an advantage over Round Robin. On the other hand, by making a more sequential deployment to more nodes according to the Next Fit algorithm, it achieved a fault-tolerant deployment as well as a 14% improvement in communication cost.

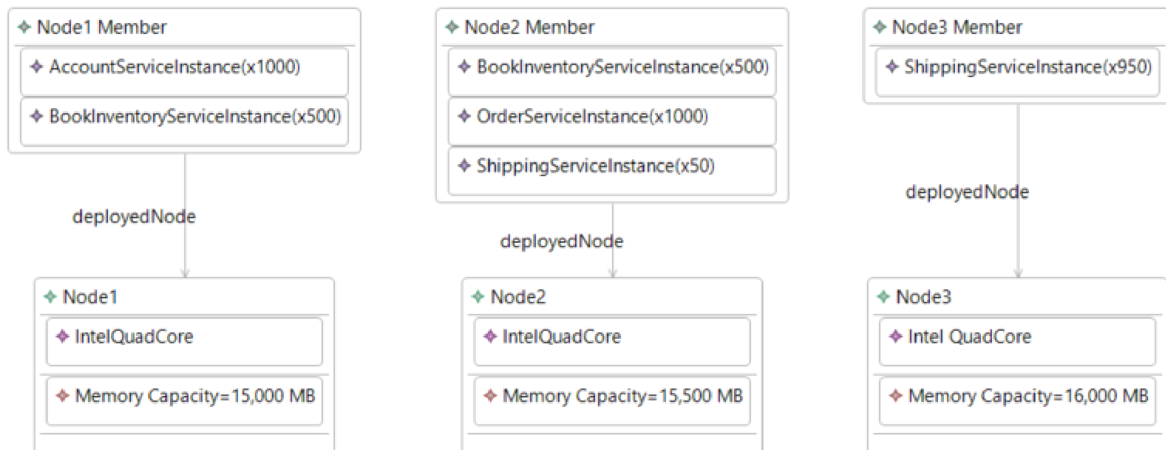

Figure 5.11 Deployment structure using Best Fit algorithm

Table 5.6 Improvement rates and total cost values of the algorithms after the capacity change in the nodes

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	3	215.14	14.30
Next Fit	3	215.14	14.30
Hungarian	6	184.04	-0.17
Round Robin	6	184.37	0.00
Proposed Approach	6	184.37	

Table 5.5 and Table 5.6 show that, compared to the four algorithms after changing the memory capacities of the nodes, there is a relative improvement between 9.85% to 14.3% in terms of communication cost in favor of the fault-tolerant algorithm. Only in the high occupancy experiments, the proposed algorithm has slightly lower performance compared to the Hungarian algorithm.

The specific fault-tolerant approach consistently demonstrates its robustness in optimizing communication costs in various scenarios. It outperforms the Best Fit Algorithm, showing high improvement rates, especially under conditions where node occupancy exceeds 50%. Similarly, compared to the Next Fit Algorithm, the particular fault-tolerant approach

performs well in both total communication cost and improvement rates under similar conditions such as increasing node occupancy. Although it performs well against Hungarian and Round Robin Algorithms when node occupancy is less than 50%, it loses this advantage in higher occupancy scenarios. Here, it achieves a success rate of 14.3% compared to the Best Fit and Next Fit algorithms. These results highlight the robustness and effectiveness of the fault-tolerant custom approach in optimizing communication overhead and improving network efficiency in various deployment environments.

5.5. Experiment 3: The effect of the number of microservice instances

In the third experiment, five algorithms are tested with different number of microservice instances. Then, the deployments are compared, and the changes in communication costs are observed.

Table 5.7 shows how many instances of each service have occupancy rates above and below 50% and the cost of each instance. Unlike the previous experiments, the number of service instances has been determined as unidentical.

Table 5.7 Number of microservice instances and capacities of each instances during deployment for all algorithms

Service Name	Number of Service Instance		Memory Capacities Per Instance(MB)
	Occp. $\leq 50\%$	Occp. $> 50\%$	
Account Service	500	1000	10
Book Inventory	300	1300	10
Order Service	600	1500	10
Shipping Service	1000	2000	10

5.5.1. Occupancy Less Than 50%

After the number of microservice instances differed, Figure 5.12 demonstrates the deployment of services to nodes using the Fault Tolerant algorithm with less and more

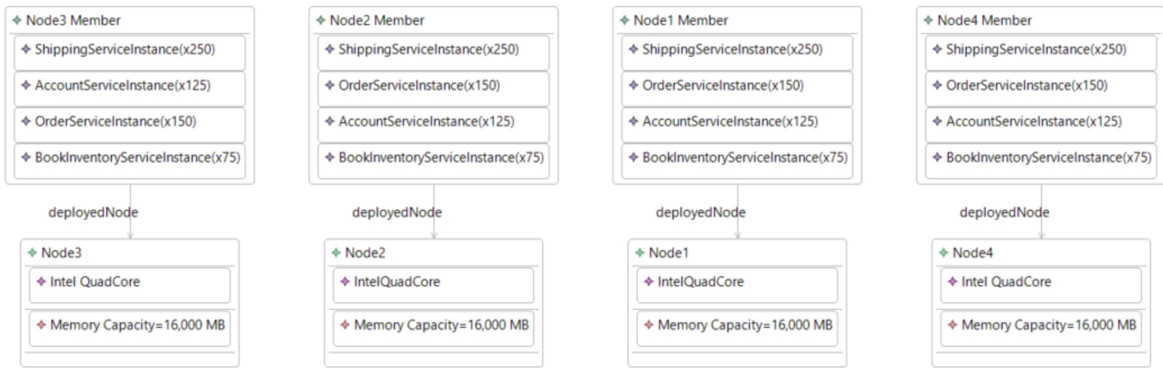

Figure 5.12 Deployment structure using Fault Tolerant algorithm

than 50% occupancy, respectively. Various numbers of service instances are reported to be deployed on each node.

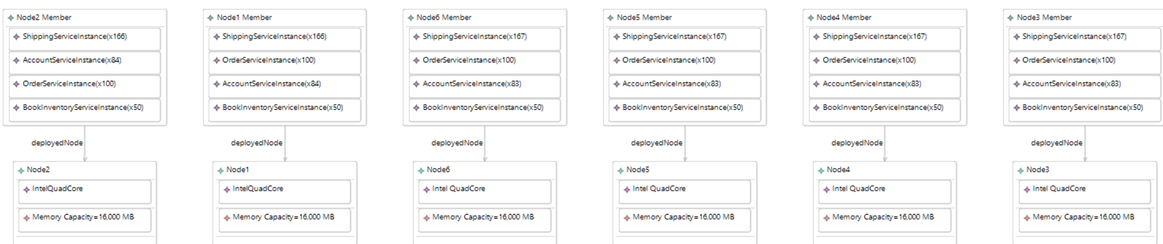

Figure 5.13 Deployment structure using Round Robin algorithm

The Round Robin model diagram, as seen in Figure 5.13, demonstrates how the fault-tolerant approach exceeds the other algorithms when the occupancy is less than 50% in the third experiment.

The figures demonstrate that the suggested methodology uses fewer nodes and the equal deployment method with Round Robin when the deployment is carried out with less than 50% occupancy. It aims to decrease the cost of communication as a result. Table 5.8 demonstrates that a 10% improvement was accomplished.

Table 5.8 Improvement rates and total cost values of the algorithms after number of instances changes

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	2	76.90	6.25
Next Fit	2	76.90	6.25
Hungarian	6	79.84	9.70
Round Robin	6	80.10	10.00
Proposed Approach	4	72.09	

5.5.2. Occupancy More Than 50%

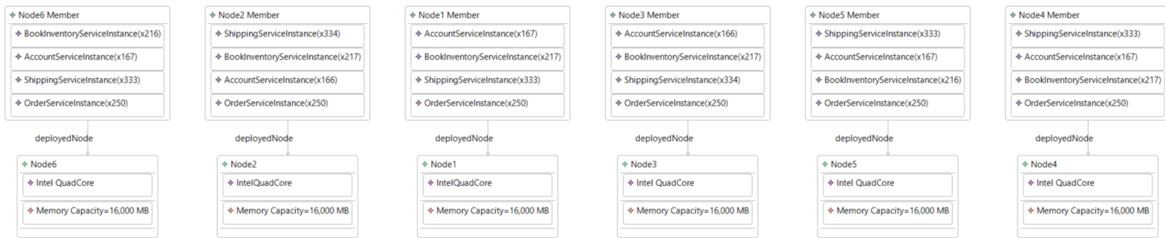

Figure 5.14 Deployment structure using Fault Tolerant algorithm

Figure 5.15 shows a deployment model diagram of Best Fit which fault-tolerant approach is resulted as best rate against in the third experiment.

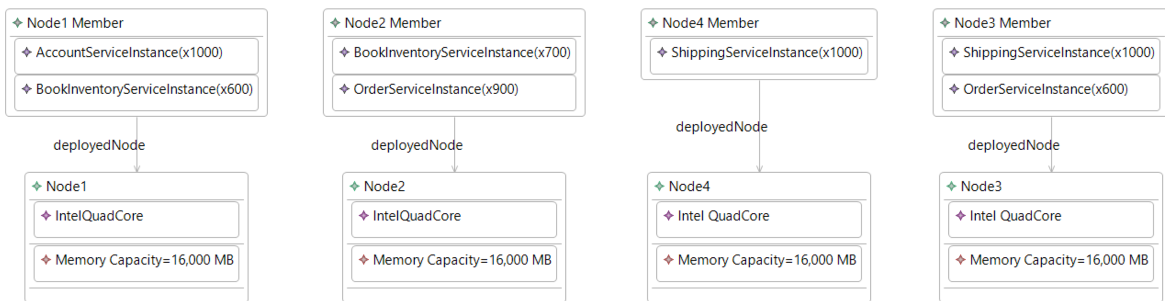

Figure 5.15 Deployment structure using Best Fit algorithm

The comparison of communication costs of the experimental algorithms is shown in Table 5.8 and Table 5.9.

In the occupancy rates above 50% in the Table 5.9, the method could not gain an advantage over Round Robin. On the other hand, by making a more sequential distribution to more nodes compared to the Best Fit and Next Fit algorithms, it achieved both a fault-tolerant deployment and a 10.2% improvement in communication cost.

Table 5.9 Improvement rates and total cost values of the algorithms after number of instances changes

Algorithm name	Number of Nodes	Total Communication Cost (MB/sec)	Improvement Rate of Proposed Approach(%)
Best Fit	4	534.05	10.23
Next Fit	4	534.05	10.23
Hungarian	6	478.32	-0.21
Round Robin	6	479.38	0.00
Proposed Approach	6	479.38	

As shown in Tables 5.8 and Table 5.9, the Fault-Tolerant algorithm performs better than the Round Robin and Hungarian algorithms in terms of 10% communication cost for occupancies below 50% when the number of instances is changed. It provides 10.2% improved results when compared to the Best Fit and Next Fit algorithms. Considering the results of the other two experiments, even though the rates may be different, the successful algorithms show consistency in both experiments.

The fault-tolerant custom approach emerges as a robust solution, surpassing several established algorithms in optimizing communication overhead within microservice deployments. Compared to the Best Fit Algorithm, it consistently has lower total communication costs, notably advantageous when the number of instances exceeds 50%.

Additionally, it achieves a significant improvement rate in Experiment 3 highlighting its efficiency in minimizing communication overhead. Similarly, the fault tolerant approach keeps improving upon the Next Fit Algorithm in terms of communication cost especially when the number of instances increases. Compared to the Round Robin Algorithm, it is

observed that the proposed approach does not provide an advantage for occupancies above 50%, but performs better in total communication cost, especially with a smaller number of instances. These results demonstrate the reliability and effectiveness of the fault-tolerant approach in improving deployment efficiency in microservice architectures at a certain level.

6. DISCUSSION AND CONCLUSION

In this thesis, a fault-tolerant deployment approach for microservices is proposed to minimize communication costs and provide strong fault tolerance during the design phase of the software lifecycle. Our approach has been tested in a number of experiments using the Micro-IDE tool.

Comparative Analysis

In the first experiment, we compared our approach with traditional algorithms such as Best fit, Next fit, Hungarian, and Round Robin under the same conditions. Our comparative findings show an improvement in communication cost reductions ranging from 3.6% to 10% for node occupancy below 50%. The proposed approach shows 27.33% communication cost reduction compared to the Best Fit and Next Fit algorithms for node occupancy above 50%. Apart from these algorithms, it is seen that the fault-tolerant approach did not provide significiant advantage against Round Robin and Hungarian algorithms. The experimental results indicate that our approach may be preferred against Round Robin and preferably other algorithms for occupancies below 50%, while it may be preferred against algorithms such as Best Fit and Next Fit for occupancies above 50%.

In the second experiment, only server memory capacity parameters are changed and the fault-tolerant approach continued to outperform other algorithms. It achieves improvement in communication cost 10% for less than 50% occupancy and 14.3% of improvement for more than 50% occupancy against the Round Robin and Best Fit algorithms, respectively.

The third experiment also examined the impact of varying the number of microservice instances across services. In this experiment, communication costs are optimized by 6.25% to 10% for occupancy below 50%. Although the proposed algorithm achieves a 10.2% improvement over Best Fit and Next Fit for occupancy above 50%, no advantage is obtained against Round Robin and Hungarian algorithms, just like in other experiments.

As a result, besides the proposed fault-tolerant deployment approach is useful for nodes below 50% occupancy, at 50% occupancy and above, it is fault tolerant against certain algorithms and could optimize communication cost better.

Limitations

Despite these promising results, the proposed algorithm and the deployment environment have some limitations. The experiments were conducted in a controlled environment that may not fully capture the complexity of real-world software systems. Factors such as network latency, real-time data traffic, and ironically, unforeseen service failures can affect the performance of our approach.

Additionally, the scope of our experiments was limited to specific algorithms and settings. Future work can expand on this topic by including a wider range of algorithms and more diverse environmental conditions to further validate the robustness and effectiveness of the fault-tolerant deployment strategy.

Future Research Directions

Future research may focus on exploring the integration of security measures within the fault-tolerant deployment framework. Since microservices are often run on distributed networks, security becomes a top concern. It would be valuable to examine how security mechanisms can be incorporated into the deployment strategy without sacrificing performance.

In addition, scalability and resource utilization are critical factors that require a deeper investigation. Using design-time and run-time fault tolerance methods together may be an interesting research and may be more useful to increase the robustness of microservice applications.

Conclusion

Our approach provides an effective solution for designing fault-tolerant, viable microservice deployment architectures that minimize communication costs while ensuring reliability and

efficiency. Incorporating fault tolerance in the design phase is crucial for the success of microservice-based systems to hinder the cascading failures in the remaining phases. Our findings pave the way for the further researchers in areas such as security, scalability, and resource utilization, which are crucial for improving the feasibility and maintainability of fault-tolerant microservices architectures, especially in dynamic and unpredictable environments.

REFERENCES

[1] Abdul Razzaq. A systematic review on software architectures for iot systems and future direction to the adoption of microservices architecture. *SN Computer Science*, 1(6):350, **2020**.

[2] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, **2018**.

[3] Md Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *IEEE communications surveys & tutorials*, 15(2):909–928, **2012**.

[4] Jacopo Soldani, Giuseppe Montesano, and Antonio Brogi. What went wrong? explaining cascading failures in microservice-based applications. In *Service-Oriented Computing: 15th Symposium and Summer School, SummerSOC 2021, Virtual Event, September 13–17, 2021, Proceedings 15*, pages 133–153. Springer, **2021**.

[5] Işıl Karabey Aksakallı, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdoğan. Micro-ide: A tool platform for generating efficient deployment alternatives based on microservices. *Software: Practice and Experience*, 52(7):1756–1782, **2022**.

[6] Isil Karabey Aksakalli, Turgay Celik, Ahmet Burak Can, and Bedir Tekinerdogan. A model-driven architecture for automated deployment of microservices. *Applied Sciences*, 11(20):9617, **2021**.

[7] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, **2022**.

[8] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, **2020**.

[9] Yuke Jia, Tiejun Wang, Tianbo Qiu, Xiaohan Zhang, Rui Wang, and Tianyu Wo. Fault tolerance of stateful microservices for industrial edge scenarios. In *2023 IEEE International Conference on Joint Cloud Computing (JCC)*, pages 50–56. IEEE, **2023**.

[10] Chao Lei and Hongjun Dai. A heuristic services binding algorithm to improve fault-tolerance in microservice based edge computing architecture. In *2020 IEEE World Congress on Services (SERVICES)*, pages 83–88. IEEE, **2020**.

[11] Thomas F Düllmann and André Van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th acm/spec on international conference on performance engineering companion*, pages 171–172. **2017**.

[12] Stephen W Liddle. Model-driven software development. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, pages 17–54. Springer, **2011**.

[13] Tomáš Livora. *Fault tolerance in microservices*. Ph.D. thesis, Masarykova univerzita, Fakulta informatiky, **2017**.

[14] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 165–174. **2016**.

[15] Francisco Silva, Valéria Lelli, Ismayle Santos, and Rossana Andrade. Towards a fault taxonomy for microservices-based applications. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, pages 247–256. **2022**.

[16] Sahar Smaali, Riadh Benbessem, and Hatem Mohamed Nazim Touati. A fault tolerance and recovery formal model for iot systems. *International Journal of Organizational and Collective Intelligence (IJOCI)*, 12(2):1–24, **2022**.

[17] Heberth F Martinez, Oscar H Mondragon, Helmut A Rubio, and Jack Marquez. Computational and communication infrastructure challenges for resilient cloud services. *Computers*, 11(8):118, **2022**.

[18] Swati Goel and Ratneshwer Gupta. Architecture level fault tolerance modeling for soa based systems. In *Smart Systems: Innovations in Computing: Proceedings of SSIC 2021*, pages 1–9. Springer, **2022**.

[19] Alexander Power and Gerald Kotonya. A microservices architecture for reactive and proactive fault tolerance in iot systems. In *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 588–599. IEEE, **2018**.

[20] J Abdul Rasheedh and S Saradha. Reactive microservices architecture using a framework of fault tolerance mechanisms. In *2021 second international conference on electronics and sustainable communication systems (ICESC)*, pages 146–150. IEEE, **2021**.

[21] Josip Zilic, Vincenzo De Maio, Atakan Aral, and Ivona Brandic. Edge offloading for microservice architectures. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, pages 1–6. **2022**.

[22] Mihai Baboi, Adrian Iftene, and Daniela Gîfu. Dynamic microservices to create scalable and fault tolerance architecture. *Procedia Computer Science*, 159:1035–1044, **2019**.

[23] Alexandre Huff, Matti Hiltunen, and Elias P Duarte. Rft: Scalable and fault-tolerant microservices for the o-ran control plane. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 402–409. IEEE, **2021**.

[24] Asad Javed, Keijo Heljanko, Andrea Buda, and Kary Främling. Cefiot: A fault-tolerant iot architecture for edge and cloud. In *2018 IEEE 4th world forum on internet of things (WF-IoT)*, pages 813–818. IEEE, **2018**.

[25] José Flora, Paulo Gonçalves, Miguel Teixeira, and Nuno Antunes. A study on the aging and fault tolerance of microservices in kubernetes. *IEEE Access*, 10:132786–132799, **2022**.

[26] Na Wu, Decheng Zuo, and Zhan Zhang. An extensible fault tolerance testing framework for microservice-based cloud applications. In *Proceedings of the 4th International Conference on Communication and Information Processing*, pages 38–42. **2018**.

[27] Zheng Liu, Guisheng Fan, Huiqun Yu, and Liqiong Chen. An approach to modeling and analyzing reliability for microservice-oriented cloud applications. *Wireless Communications and Mobile Computing*, 2021:1–17, **2021**.

[28] Daniel Tabak and Alexander H Levis. Petri net representation of decision models. *IEEE Transactions on Systems, Man, and Cybernetics*, (6):812–818, **1985**.

[29] Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze. Highly-available applications on unreliable infrastructure: Microservice architectures in practice. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 130–137. IEEE, **2017**.

[30] Nikhil Premanand Bhat. *OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING*. Ph.D. thesis, **2020**.

[31] Carter Bays. A comparison of next-fit, first-fit, and best-fit. *Communications of the ACM*, 20(3):191–192, **1977**.

[32] G Ayorkor Mills-Tettey, Anthony Stentz, and M Bernardine Dias. The dynamic hungarian algorithm for the assignment problem with changing costs. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27*, **2007**.

[33] Dimitrios Tychalas and Helen Karatza. Samw: a probabilistic meta-heuristic algorithm for job scheduling in heterogeneous distributed systems powered by microservices. *Cluster Computing*, 24(3):1735–1759, **2021**.