International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V71I11P206

Volume 71 Issue 11, 56-68, November 2023

Original Article

Component-Based Software Development through DCFTM in Software Engineering

Lalu Banothu[1], M. Chandra Mohan[2], C. Sunil Kumar[3]

[1,2]*Department of Computer Science Engineering, JNTUH College of Engineering, Hyderabad, India.*
[3]*Department of Computer Science Engineering, The Apollo University, Chittoor, Andhra Pradesh, India.*

[1]*Corresponding Author : lalunb.csegnitc@gniindia.org*

Received: 14 May 2023 Revised: 25 August 2023 Accepted: 04 October 2023 Published: 04 November 2023

Abstract *- The evolution of software systems has witnessed tremendous changes in the last decade, moving from simple web-based applications to enterprise-level distributed applications built on top of interoperable components. When systems are realized with the integration of heterogeneous components, they should evolve to accommodate changes gracefully. Moreover, systems need to be resilient against runtime faults that can occur for different reasons. Component-based software engineering has been phenomenal in producing such systems that drive the home chain of businesses in the real world. Building an enterprise application based on reusable components, instead of reinventing the wheel, is the main approach in the contemporary era. The reusable components are platform-independent and interoperable in nature. There is every possibility to have certain faults as the components are heterogeneous in nature, and they are location transparent as well. Several approaches were found in the literature to have fault-tolerant architectures in this context. However, there is still a need for leveraging fault tolerance architecture by addressing the problem of dynamic configuration of fault tolerance mechanisms at runtime. Towards this end, in this paper, we proposed a novel fault-tolerant architecture for component-based software development in the domain of software engineering. We proposed an algorithm known as Dynamic Configuration of Fault Tolerance Mechanisms (DCFTM) to ensure the system can withstand different kinds of faults at runtime and be resilient against faults. A case study enterprise application with distributed component-based architecture is built to evaluate the proposed fault-tolerant architecture and underlying DCFTM algorithm to prove the concept. The empirical study revealed that the DCFTM algorithm outperforms state of the art.*

Keywords *- Software engineering, Fault tolerant architecture, Component-based software development.*

1. Introduction

Component-based software systems are essentially distributed in nature, where server components can be geographically located anywhere in the world. Distributed computing is a completely server-side phenomenon where different server programs or components work together. Reusing server components in related applications is also possible instead of reinventing the wheel. Moreover, the components with different configurations involved in the system are to be used appropriately. Especially there is a need for fault tolerance among critical configurations [1]. As presented in Figure 1, software components can be constructed and reused to form different applications. Each application can have critical configurations to be maintained in a fault-tolerant approach. Fault tolerance should be one of the salient features of distributed applications based on Service Oriented Architecture (SOA). Faire of critical configurations can cause severe problems to the underlying system. The rationale behind this is that a distributed application must be made available around the clock and should have scalability besides resiliency against possible faults at runtime.

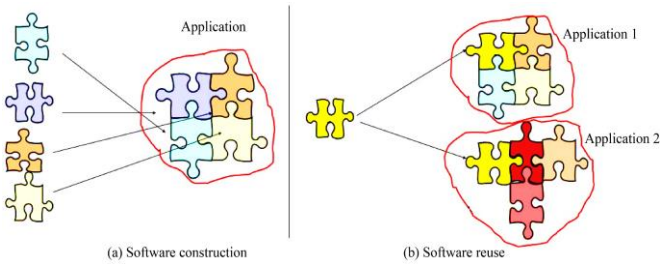

Fig. 1 Shows how to construct software using components and reuse them in different applications

Millions of interactions are possible in a live distributed application as it has users across the globe. It is necessary to maintain configuration interactions correctly, and there are characteristics of different interactions. An interaction is the communication between the components in a given configuration of a distributed application. There is a need for fault tolerance, which reflects the health of the distributed application that gives Quality of Service (QoS) despite faults occurring. As discussed in [2], it is also known as the resiliency of the distributed system. Moreover, component

reliability in configurations while serving the purpose against client calls is another important consideration [7]. There might be faults propagating from the source to other components at runtime, and such faults need to be handled appropriately [14]. In distributed applications, different technologies like RMI are used. Therefore, there is a need to be aware of technology-specific configurations and methods of invocations, if any.

There are many existing approaches found in the literature for fault-tolerant architectures. Chinnaiah and Niranjan [1] proposed a methodology for fault-tolerant software based on configurations for the cloud. Bajunaid and Menasce [2] used the concepts of checkpointing and rollback to improve availability in component-based software systems. Schirmeier et al. [3] proposed a framework for fault injection to determine the ability of fault tolerance of hardware implemented using software. Their framework has pre-injection and post-injection analysis besides the fault injection method. Himabindu et al. [4] focused on resilient embedded systems and discussed software-based fault-tolerant approaches, while Smara et al. [5], on the other hand, focused on different aspects pertaining to fault tolerance. Liu et al. [6] proposed a software rejuvenation-based fault tolerance approach. Pham et al. [7] focused on component-based reliability prediction. Chemashkin and Zhilenkov [8] proposed an active fault-tolerant control system. Mukwevho and Celik [9] made a review of fault-tolerant methods towards the smart cloud. Fiondella [10] focused on the reliability of component systems with positive and negative correlations. The work found in the literature can be categorized into configuration approaches ([1, 7, 8, 18]), checkpoint approaches ([2, 6, 9, 10]), software-based approaches ([3, 4, 5, 11, 12]) and others.

From the literature review, it is understood that there is a need for improving fault tolerance dynamics in component-based distributed applications. As fault tolerance dynamics are associated with configurations of different reusable artefacts, this paper proposes an algorithm for ensuring fault tolerance, which is evaluated using a Distributed Reservation System (SOA) implemented using Java's RMI based on SOA architecture. Our contributions to this paper are as follows.

- A fault-tolerant architecture is proposed for achieving fault tolerance in component-based software engineering.
- An algorithm named Dynamic Configuration of Fault Tolerance Mechanisms (DCFTM) for ensuring fault tolerance in component-based systems.
- A case study application is implemented using Java programming language to prove the concept.

The remainder of the paper is structured as follows. Section 2 reviews the literature on the state of the art pertaining to fault-tolerant architectures for component-based software engineering. Section 3 presents the proposed architecture and algorithm for fault tolerance. Section 4 presents the cast study application. Section 5 presents experimental results, while section 6 concludes the paper and gives directions for future work.

2. Related work

This section reviews the literature on fault-tolerant approaches existing for component-based software systems.

2.1. Configuration Approaches

Chinnaiah and Niranjan [1] proposed a methodology for fault-tolerant software based on configurations for the cloud. Their method is based on the frequency of configuration and interactions and their characteristics. It has a provision to create a failure log for future revisions. They considered both proactive and reactive fault-tolerant schemes. Pham et al. [7] focused on component-based reliability prediction. Their prediction model involves different layers such as component developers, service architecture and a reliability prediction tool. Transformation and Markov models are used in the process of reliability prediction. They intend to improve it in future with error propagation to develop more approaches for fault tolerance. Chemashkin and Zhilenkov[8] proposed an active fault-tolerant control system that includes the reconfigurable feed-forward controller and reconfiguration mechanism for fault detection and diagnosis. It considers dynamic faults and system configuration faults leading to problems in reliability.

Wienke and Wrede [18] proposed an autonomous fault detection framework for component-based systems considering performance bugs. Their method makes use of a regression approach to identify faults. Their algorithm is assisted by feature generation that reduces the complexity of the proposed method. It is found to be stable with robotic systems. Peng and Huang [19] used stochastic modelling to tolerate faults in distributed applications. A web service-based system is used for empirical study. It has server-side restful functions and client-side applications where composite services are consumed by the application.

2.2. Checkpointing Approaches

Bajunaid and Menasce [2] used checkpointing and rollback to improve availability in component-based software systems. They used a queuing network model and checkpointing for heterogeneous software components. The components are represented as Markov chains for better performance. Liu et al. [6] proposed a software rejuvenation-based fault tolerance approach. Their scheme was meant for cloud applications. The system depends on a checkpointing technique with a rejuvenation agent that works between the original Virtual Machine (VM) and the interim node. However, they intend to improve it for better failure detection accuracy. Mukwevho and Celik [9] made a review of fault-tolerant methods towards the smart cloud. The methods include proactive, reactive and resilience-based methods. They intend to use machine learning towards fault tolerance methods in future. Jafary and Fiondella [10] focused on the

reliability of component systems with positive and negative correlations. They explored methods like bivariate Bernoulli and component reliability expressions. In future, they intended to generalize their approach to be suitable for different series of components.

Song et al. [13] proposed an Interface Definition Language (IDL) for system-level fault tolerance in embedded systems. Their tool is known as SuperGlue, which generates code for an interface-driven approach towards finding faults. The tool also has provisions to optimize and decrease the code needed for fault recovery. Though it causes non-prohibitive slowdown, it does not lead to system failure. Shu et al. [14] considered fault propagation and architecture-based reliability of distributed systems. The fault tolerance analysis approach makes use of the fault pervasion intensity matrix, input and output state matrix and fault tolerance of the architecture. The reliability analysis model is made using transition probability among the distributed components. Stoicescu et al. [15] proposed an adaptive fault tolerance approach for realizing resilient computing systems. Their method includes both offline and online agile management of fault tolerance mechanisms. Fault tolerance design patterns are used for the effective implementation of their method. The main advantage is agility, which needs further investigation with different case studies.

Nagaraju et al. [20] focused on correlated failures of server systems in distributed applications. A fault-tolerant server is built using a software rejuvenation approach. The system has different states and transitions between them. Fault recovery, available and rejuvenation are the states available. However, they intend to include the impact of correlation in their future endeavours.

2.3. Software-based Fault Tolerance and Other Methods

Schirmeier et al. [3] proposed a framework for fault injection to determine the ability of fault tolerance of hardware implemented using software. Their framework has pre-injection and post-injection analysis besides the fault injection method. It makes use of meta-information in order to have the framework realized. With fault injection complaints, their architecture finds the capability of fault tolerance approaches. Holler [4] focused on resilient embedded systems and discussed software-based fault-tolerant approaches. They reviewed about fault injection methods and fault tolerance methods based on software diversity.

On the other hand, Smara et al. [5] focused on different aspects of fault tolerance. First, it focused on fault detection using acceptance tests. Second, they investigated different kinds of faults, such as response-time failures, software faults and transient hardware faults. With a fire control system case study and model checker, they investigated and evaluated their method. However, they did not focus on the core work of fault recovery and tolerance.

Aponte-Moreno et al. [11] proposed a software-based fault tolerance method supported by Approximate Computing (AC). Their approach could improve energy efficiency and reduce computational overhead in fault detection and fault tolerance mechanisms. Their method includes different phases such as approximation, fault tolerance, fault injection and fault detection.

In future, they intend to implement other fault tolerance methods on top of AC. Hellhake et al. [12] proposed a data flow-based approach and black box integration testing in distributed applications for resiliency. Their scheme considers Electronic Control Units (ECUs) as part of distributed software systems. Data flow-based convergence conditions are used to find faults with integration testing.

Zheng et al. [16] focused on software reliability issues associated with component-based systems. They illustrated software challenges considering autonomous vehicle functionalities. A model-based approach is followed, and it has an application layer, functional model, software model, hardware platform and design metrics. Dubey and Jasra [17] focused on software reliability in distributed component-based systems. They combined ANFIS and fuzzy approaches towards it.

ANFIS is the neural network model that is coupled with fuzzy logic in order to have better reliability of software. In the future, they intend to use different factors to improve their hybrid approach. Chiang et al. [21] proposed a framework for fault-tolerant reliability prediction in distributed systems. Maskura et al. [23] focused on maintainability and reliability issues of software architecture for fault tolerance. From the literature review, it is understood that there is a need for improving fault tolerance dynamics in component-based distributed applications.

3. System Model and Proposed Algorithm

The system model involves a distributed application in the real world. It is meant for travel reservations. The application is elaborated and discussed in Section 4. However, this section focuses on the system model and the algorithm proposed. The system model includes RMI client and RMI server components. The distributed server components include a car server (C), flight server (F), room server (R) and middleware server (M) as presented in Figure 2. Out of the server components reused with different configurations, M is the component that helps access other components. The SOA-based reservation system is essentially encapsulated by C, F and R.

3.1. Problem Definition

Provided the components such as C, F and R, building a component-based system and its implementation with an underlying algorithm for fault tolerance in the travel reservation system is the problem considered.

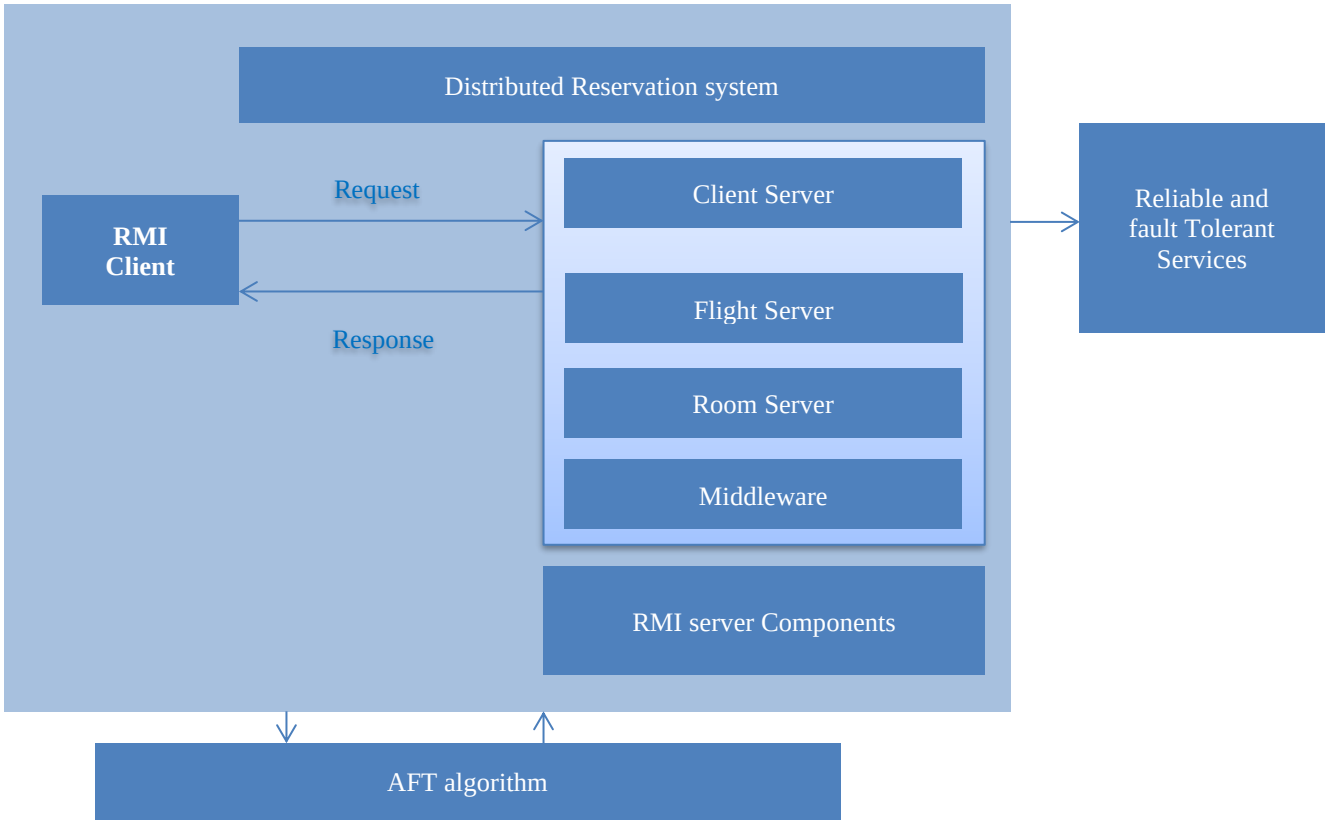

Fig. 2 The system model

In the case of SOA-based architecture with Single Sign On (SSO), it is essential to have different configured services that involve required server-side components. It is also important that the components work in a reliable fashion. In case of any fault, there needs to be fault tolerance, and the client's work is to be carried out without failing the job to be done. The critical configurations are made fault-tolerant.

Table 1. Shows notations used in the proposed system

Notation	Description
F_{jk}	the number of times configuration
C_k	the number of configurations present in a software system
C_i	Configuration
N	the number of configurations
V_i	significance of i^{th}configuration
$S(C_i)$	set of configurations
ß	parameter
X	program/job processing requirement
C	random variable
R	repair time

As presented in Table 1, different notations are used to represent the proposed system.

3.2. Mathematical Model

Consider G as a graph representing the system model; Ci, C and Ck containing components of DRS are different critical configurations. Let Ijk denote interactions between specific components such as Cj and Ck. Each reservation activity is a transaction that is made up of many operations that are executed as units conforming to ACID properties of transactions. By the end of different interactions, it is possible to determine how many successful interactions there are between pairs of components such as Cj and Ck using equation 1.

$$M(I_{jk})=\frac{F_{jk}}{\sum_{k=1}^{N} F_{jk}} \quad (1)$$

Where N denotes the number of configurations, Cj and Ck are two components between which successful interactions are made, and Fjk represents the number of times Cj invoked Ck. With every successful interaction, the value of Ijk increases. When the jth configuration never interacts with the kth configuration in its lifetime, the interaction value Ijk is set to zero. In the case of recursive configuration invocation, such as a component invoking itself, there is also increment Ijk. If the jth configuration interacts with only the kth configuration, then Ijk=1/N for all k=1 except j. Finally, a stochastic matrix

M is realized with all successful interactions. Fault tolerance can be based on the frequency of interactions. Initially, each component's individual interaction value is considered zero. As the critical configurations are used frequently in the system's interactions, it is important to keep track of failed and successful interactions. It is possible to determine frequently used configurations based on interactions.

The interaction value of Ci (ith configuration), which is represented as P(Ci), is computed as in equation 2.

$$P(C_i)= \frac{1-\alpha}{N} + \alpha \sum_{j \epsilon S(C_i)} P(C_j) M(I_{ji}) \quad (2)$$

$$P(C\_1)=1-\alpha/N +\alpha \sum$$

Where a set of configurations is denoted as S(Ci) that involve interactions with the configuration such as Ci, finally, the performance of critical configuration is computed as in Eq. 3.

$$P(C_i)=(1 - \alpha)\frac{\beta}{|C|} + \alpha \sum_{j \epsilon S(C_i)} P(C_j) M(I_{ji}) \quad (3)$$

Moreover, with respect to a non-critical configuration, the performance of such configuration is computed as in Eq. 4.

$$P(C_k)=(1 - \alpha)\frac{1-\beta}{|NC|} + \alpha \sum_{j \epsilon S(C_k)} P(C_j) M(I_{jk}) \quad (4)$$

With the recovery block approach where redundant modules are used in programming, the probability of recovery, denoted as F, can be computed as in Eq. 5.

$$F=\prod_{i=1}^{n} f_i \quad (5)$$

3.3. Algorithm Design

The algorithm considers different components in the DRS, such as C, F, R, and M. There needs to be an array of fault tolerance candidates with values and a configuration value that requires the algorithm's fault tolerance candidate as input. The algorithm's outcome is finding the best fault tolerance candidate for the given configuration.

Algorithm 1: Dynamic configuration of fault tolerance mechanisms algorithm

Algorithm: Dynamic Configuration of Fault Tolerance Mechanisms (DCFTM)
Inputs: Fault tolerance candidate values vector t, configuration C

1. Initialize x to zero
2. Initialize i to one

Finding Eligible Candidates

3. For each i in 1 to n
4. If C>=t[i] Then
5. S[x]=t[i]
6. Increment x
7. End If
8. End For

Finding Candidate that has Minimum Failure Probability

9. Assign S[1] to min
10. For each j from 2 to x
11. IF S[j]<min Then
12. min=S[j]
13. FT(C) =j
14. End If
15. End For

As presented in Algorithm 1, it is evident that the algorithm takes two inputs. The first input is a set of fault-tolerance candidate values, and the second parameter is a fault-tolerant configuration. The algorithm's output is to find the best fault tolerance candidate for execution at runtime. Based on the interaction value of the given configuration, the suitable fault-tolerant candidates are identified, and finally, the best one is determined.

The algorithm takes an array of FT candidates and returns the best candidate for a given configuration. The best candidate is the one which exhibits minimum failure probability. Step 1 and Step 2 x and i are initialized to zero and one, respectively. Both act as index values for vectors such as S and T, where S is a vector to hold eligible candidates. Step 3 through Step 8 is an iterative process that finds all the suitable candidates for given C, and they are assigned to vector S. Step 4 has a condition to check whether the given FT candidate is suitable for given configuration requirement C.

Step 5 adds a suitable configuration to S. Step 6 increments the index x. By the end of the iterative process, S holds all suitable FT candidates that can be used for given configuration C. However, it is important to choose the best candidate from S. As mentioned earlier, the best candidate is the one which exhibits minimum failure probability. This process is done in the algorithm from Step 9 through Step 15. In Step 9, S[1] is the FT candidate assigned to the min variable, which is going to keep track of the best FT candidate in the ensuring iterative process takes place from Step 10 through Step 15. Step 11 iteratively checks whether the given FT candidate has a failure probability less than min. Thus, by the end of the process, the min holds the best FT candidate.

4. Case Study Application

This section presents different aspects of the case study application built to prove the concept. It covers implementation details.

4.1. Implementation Details

A distributed application is built to demonstrate proof of the concept using Java's RMI technology and API. The application is built on component architecture, where each component has self-contained functionality and can be

integrated into an application to serve a specific purpose. The application is based on broker architecture, where components are looked up by the RMI client from the RMI registry, and then calls are made to appropriate server components.

The server components implement java.rmi—remote interface to get features of a remotely callable component. Different servers are implemented for different kinds of reservations. For instance, car server implementation takes care of car reservations. Similarly, flight server implementation takes care of flight reservations, while room server implementation helps with room reservations. A single sign-on helps the user to have complete planning of a trip that involves travel and staying in hotel rooms.

As presented in Figure 3, the implementation is made with plenty of Java classes organized into several packages. The main packages include common, object, resinterface, servers and two-phase commit. These packages, in turn, can have sub-packages for better and meaningful organization of the code. The common package contains different classes that are common to various server components. The server package contains sub-packages and corresponding classes to encapsulate specific server implementations. A ResourceManager interface is defined in the resinterface package, which is the basis for methods that can be invoked. This interface extends java.rmi.Remote to support remote method invocations.

The ResourceManager interface is implemented by different server implementations such as CarServerImpl, FlightServerImpl and RoomServerImpl. Since the DRS application supports distributed monetary transactions, a two-phase commit protocol is used to achieve a transaction's Atomicity, Consistency, Isolation and Durability (ACID) properties.

4.2. Class Hierarchy

The hierarchy of classes defined in DRS includes abstract classes, interfaces and implementations. The whole application can be split into client-side and server-side functionalities. The server-side classes represent remote objects to be invoked by client applications. The overview of different classes used to realize DRS is shown in the form of a class diagram shown in Figure 4. It shows inheritance and dependency relationships among classes.

The DRS application supports distributed architecture; the server components can run on any machine geographically. Car, Flight and Hotel are reservable items. The classes such as Car, Flight and Hotel are sub-classes of a ReservableItem class. The Middleware Server Impl acts as a middleware server component to create and manage all server objects. Middleware Server Impl implements Resource Manager and uses it appropriately. Other server components are encapsulated in the implementation classes, such as Car Server Impl, Flight Server Impl and Room Server Impl.

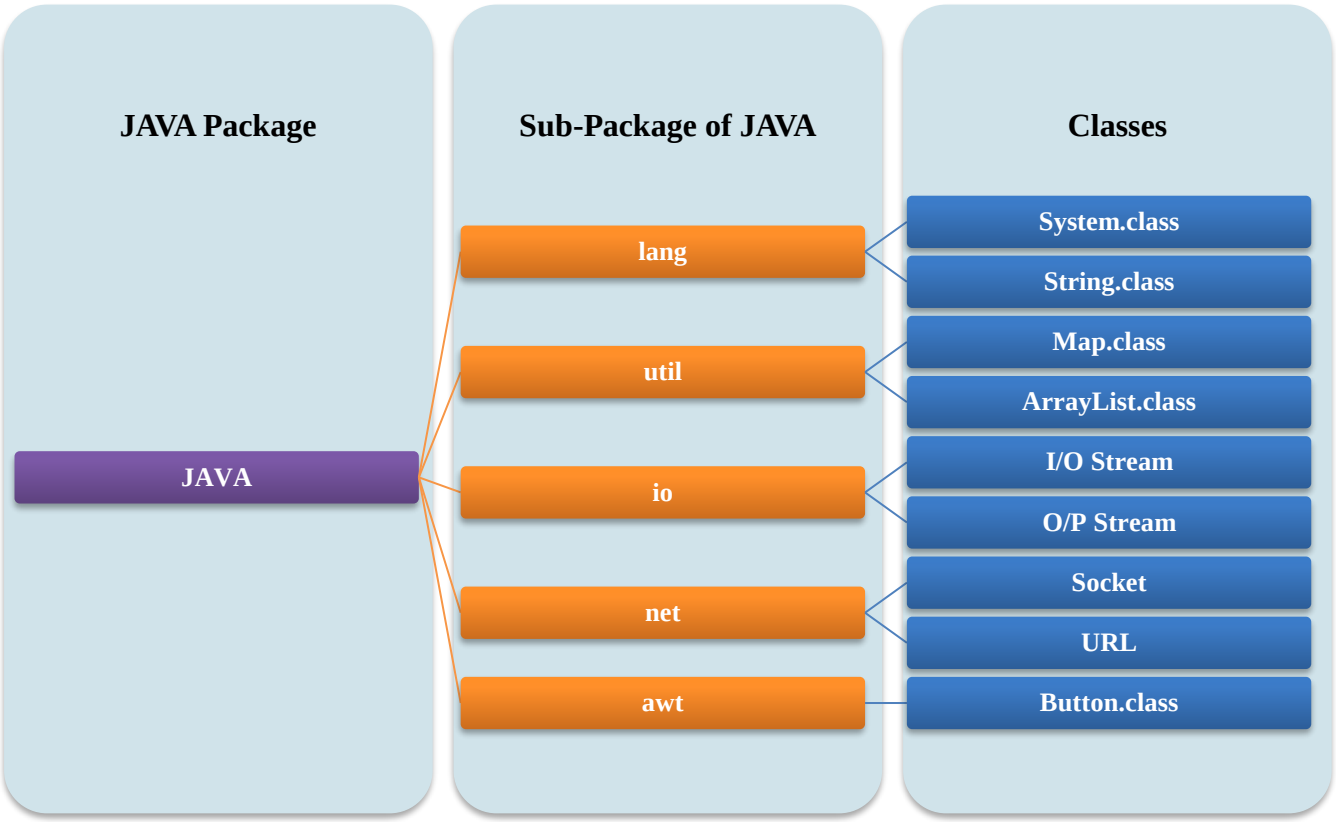

Fig. 3 Java classes packages

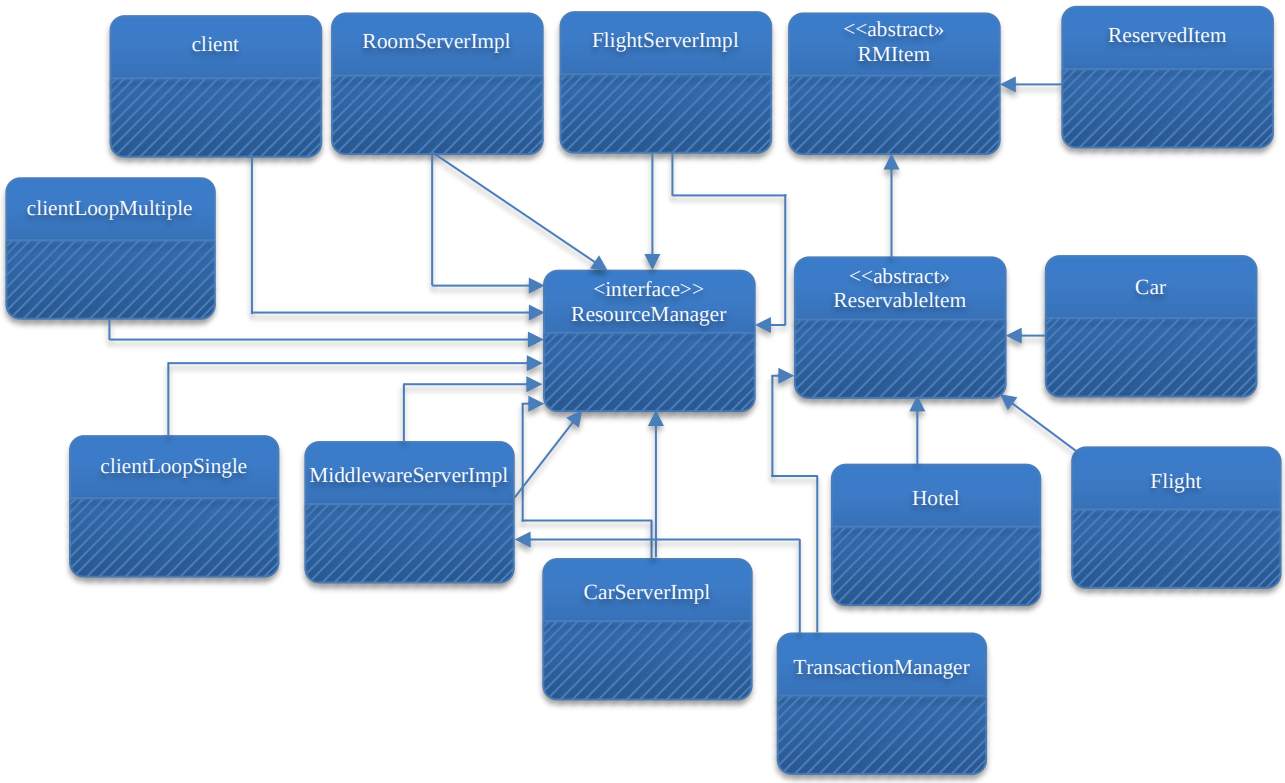

Fig. 4 Overview of class hierarchy in DRS

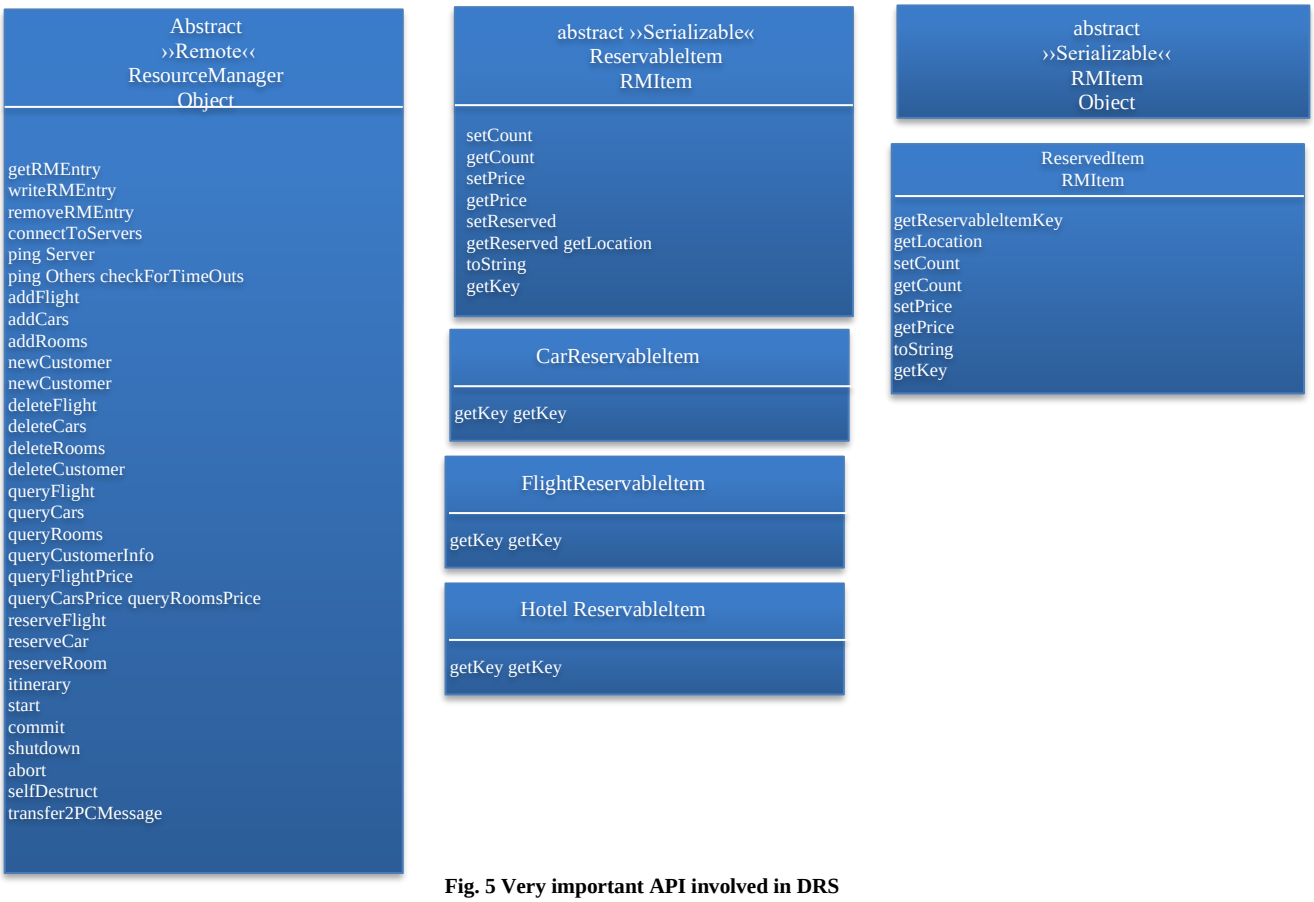

Fig. 5 Very important API involved in DRS

Fig. 6 Client-side API involved in DRS

All these servers are implementing Resource Manager, which is a remote interface[24]. The TransactionManager class uses a middleware server component and reservable items to complete transactions pertaining to travel reservations. Different client-side applications encapsulated by the classes client, clientLoopSingle and clientLoopMultiple interact with server components through ResourceManager. RMI technology is in place to support the invocation of objects running in the Java Virtual Machine (JVM) of a remote machine. All server objects are registered in the RMI registry, which holds the references of server objects. These references are looked up by the client applications in order to interact with server components.

4.3. Important API

As the DRS is based on a distributed architecture, using technology that supports remote object calling is indispensable. ResourceManager is the class that encapsulates remotely accessible functions accessed through various server objects. Very important classes of DRS are presented in Figure 5. They include four abstract classes like: ResourceManager, ReservableItem, RMItem and ReservedItem. The ResourceManager class is abstract in nature, and it implements the Remote interface of java.rmi package. Hence, it is a very important class that encapsulates all remotely invoked methods. The ResourceManager class has all the methods required by the DRS for achieving various kinds of reservations. It has methods to deal with car reservations, flight reservations and room reservations. It has methods to deal with customer requests to coordinate with different remote server objects. It also has methods required by distributed transactions and a two-phase commit protocol for consistency. The API presented has different reservable items such as Car, Flight and Hotel. It also has a class known as ReservedItem to keep track of reserved items.

4.4. Client-Side API

RMI client applications are the ones that know how to look up the RMI registry in order to obtain references of server objects whose functions can be run from a remote location. This is essential for invoking components that run in geographically located machines. Moreover, clients can be of many types. Different client applications are built to serve different users making travel reservations using single sign-on. The client-side applications are encapsulated in three different classes: client, clientLoopSingle and clientLoopMultiple, as presented in Figure 6. The first application is not interactive, while the remaining two are interactive in nature.

When compared with the second, the third model supports multiple client threads to run and get services. Thus, it can serve multiple users to have their reservations done. The three applications have a main() method to run them as client applications. They can be started and stopped as and when needed. However, RMI server components run around the clock and provide client services.

4.5. RMI Architecture and its Usage in DRS

RMI technology is one of the distributed technologies supported by Java. It is widely used in the real world to build applications that need service orientation. The implemented classes (API) can be categorized into server-side and client-side API.

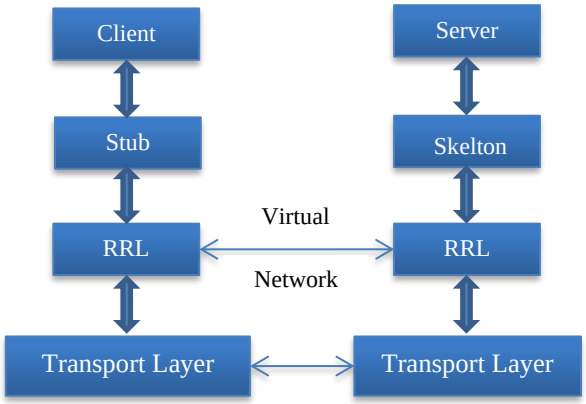

Fig. 7 Overview of RMI architecture on top of which DRS is built

There is communication between the client side and server side API through RMI technology specification. Figure 7 shows the RMI architecture in which client and server components interact.

Both RMI server and RMI client applications are essentially software components that achieve remote method invocation, thus leading to a distributed architecture. In other words, the components or servers involved in the system can run any machine located across the globe. Still, such components can work together and realize an application that serves the purpose per business rules. The client and server cannot have direct communication. Instead, they will interact through proxies known as Stub and Skeleton. The client-side proxy is known as Stub, and the server-side proxy is known as Skeleton. In other words, Stub represents a remote object (server) at the client while Skeleton resides at the server to take a request from Stub and pass it to the real server object.

The transport layer encapsulates the connectivity between client and server machines. On the other hand, the Remote Reference Layer (RRL) manages references made by the client to the remote object. The client call first goes to Stub, which passes it to the remote reference layer. RRL makes a virtual connection to server-side RRL. Then, server-side RRL sends a request to Skeleton. The Skeleton actually invokes the method in the remote object (car server/flight server/room server). Between the RMI client and server, there are procedures taking place, such as marshalling and unmarshalling. The client passes parameters to a remote method, and the parameters and calls are bundled for proper serialization. This process at the client side is known as marshalling. When the server receives the request, the process of unbundling parameters to known actual arguments and method calls is known as unmarshalling. In the process, the RMI registry plays a crucial role as the references of different RMI servers are registered with the RMI registry. The entire process is illustrated in Figure 8. The three RMI clients used in the DRS application are known as client, clientLoopSingle and clientLoopMultiple.

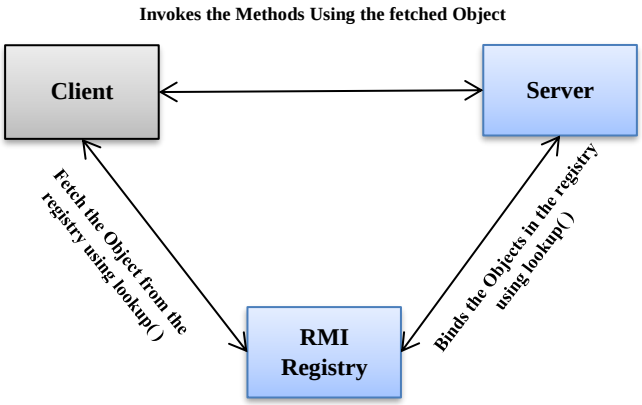

Fig. 8 Illustrates the flow of interactions between the DRS client and DRS servers through the RMI registry

These three kinds of clients essentially represent an RMI client (remote client). They cannot invoke methods on RMI servers (remote objects) directly. Instead, RMI clients need to get the reference of remote objects from the RMI registry, as the server objects are registered in the registry using the rebind() method. The client invokes the lookup() method on the registry to obtain server objects' references. Once the remote reference is available, the client can make calls on the remote objects as needed. This architecture resembles broker architecture used in RMI technology to support the worldwide reuse of distributed components.

5. Experimental Results

Experiments are made using the prototype application, which is nothing but a DRS case study. The results are observed in terms of the percentage of successful interactions versus the number of interactions with different critical configurations considered. The proposed method is compared with existing fault-tolerant methods found in [1].

As presented in Table 2, the percentage of successful interactions is provided against the number of interactions when 1% of critical configurations are used in the empirical study.

Table 2. Shows the percentage of successful interactions with 1% critical configurations

Percentage of successful interactions			
NoFT	**FCI-FT**	**FI-FS**	**DCFTM**
92.5	97.5	98	100
92	97	97.8	100
91	96.5	97.7	100
90	96	97.6	100
89	95.5	97.5	100
88	95	97.3	100
86.9	94.5	97	100
86	94	96.9	100
84.9	93.5	96.8	99
83.9	93	96.6	98.8
82.5	92.5	96.5	98.7

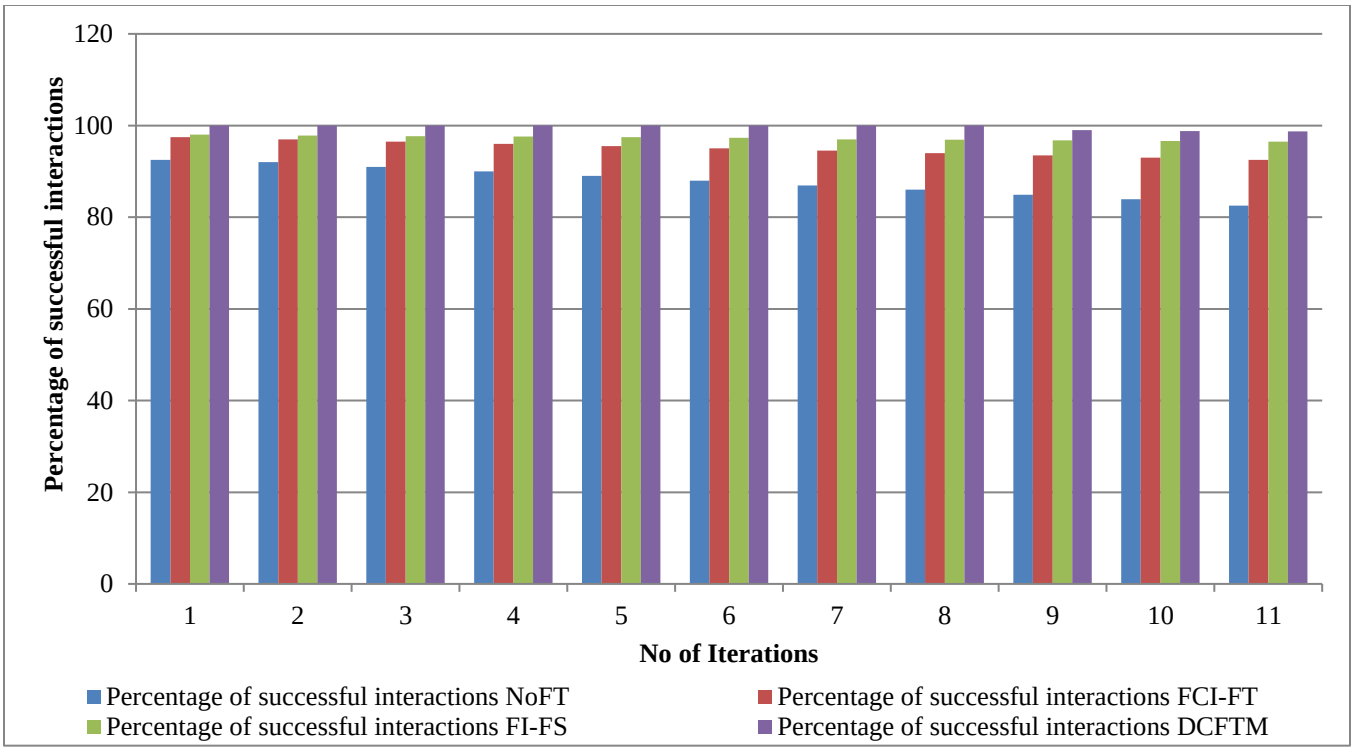

Fig. 9 Performance with 1% of critical configurations

As presented in Figure 9, the number of interactions is provided in the horizontal axis, and vertical access shows the percentage of successful interactions. The observations are made when 1% of critical configurations are used for empirical study. When no fault tolerance is used, the performance deteriorates. When the proposed algorithm DCFTM is used for fault-tolerant execution, it could achieve the best results by identifying the ideal candidate configuration.

As presented in Table 3, the percentage of successful interactions is provided against the number of interactions when 5% critical configurations are used in the empirical study.

Table 3. Shows the percentage of successful interactions with 5% critical configurations

Percentage of successful interactions			
NoFT	**FCI-FT**	**FI-FS**	**DCFTM**
86.5	94	97	100
86	93.8	96.7	100
85.8	93.7	96.5	100
85.5	93.6	96.3	100
84.9	93.5	96.1	100
83.9	93.4	95.8	99.7
83	92.9	95	99
82	91.9	94.9	98.7
80.5	91.8	93.7	97.9
79	91.7	93.5	97.7
76	91.6	93	97.4

As presented in Figure 10, the number of interactions is provided in the horizontal axis, and vertical access shows the percentage of successful interactions. The observations are made when 5% critical configurations are used for empirical study. When no fault tolerance is used, the performance deteriorates. When the proposed algorithm DCFTM is used for fault-tolerant execution, it could achieve the best results by identifying the ideal candidate configuration.

As presented in Table 4, the percentage of successful interactions is provided against the number of interactions when 10% of critical configurations are used in the empirical study.

Table 4. Shows the percentage of successful interactions with 10% critical configurations

Percentage of successful interactions			
NoFT	**FCI-FT**	**FI-FS**	**DCFTM**
69.9	79	82	100
69.5	78.9	85	100
69	78.5	84.9	100
68	77	84.7	99.9
65	76	84.5	99.8
63	75	83.9	99.6
60	74	83.8	99
57	73	83.4	98.7
50	72.5	82	97.9
41	71	81.9	97.7
31	70	75	97.4

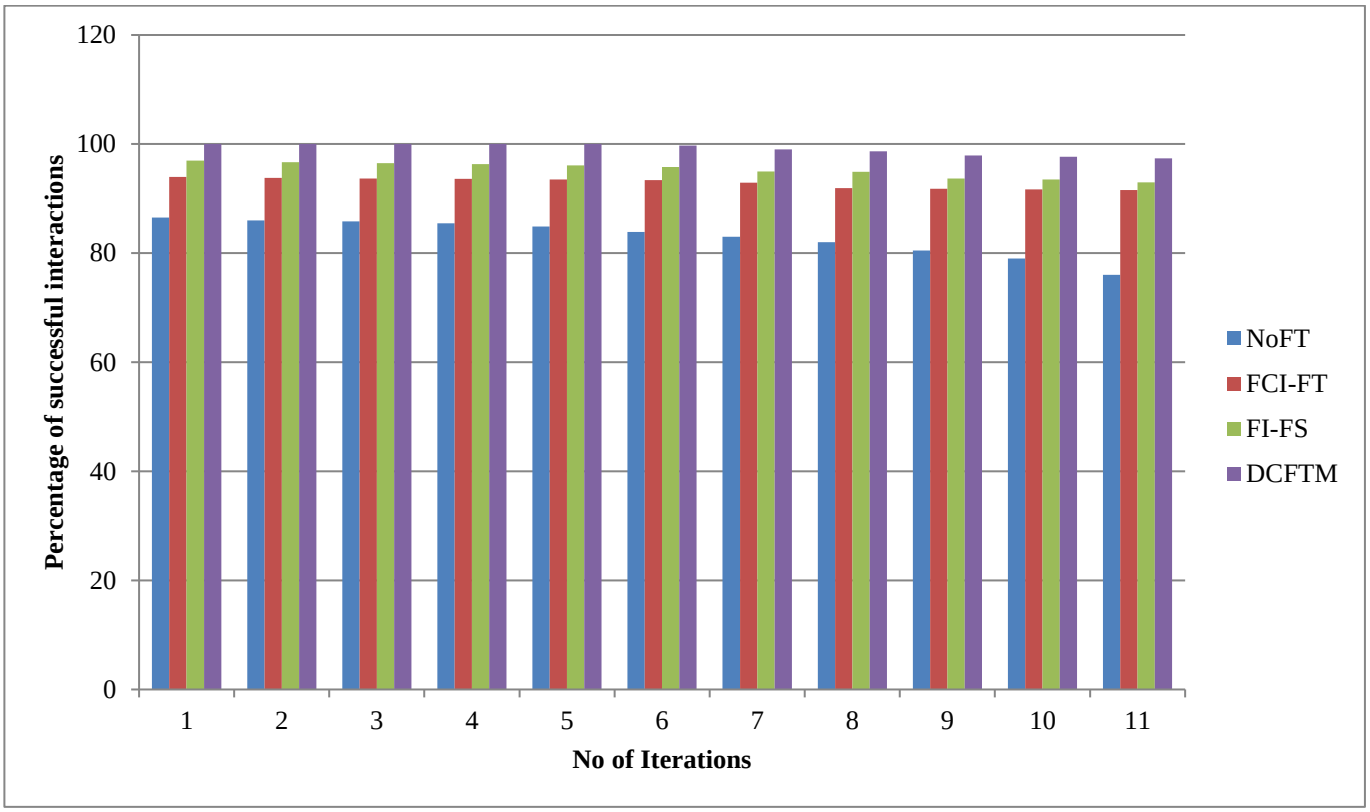

Fig. 10 Performance with 5% of critical configurations

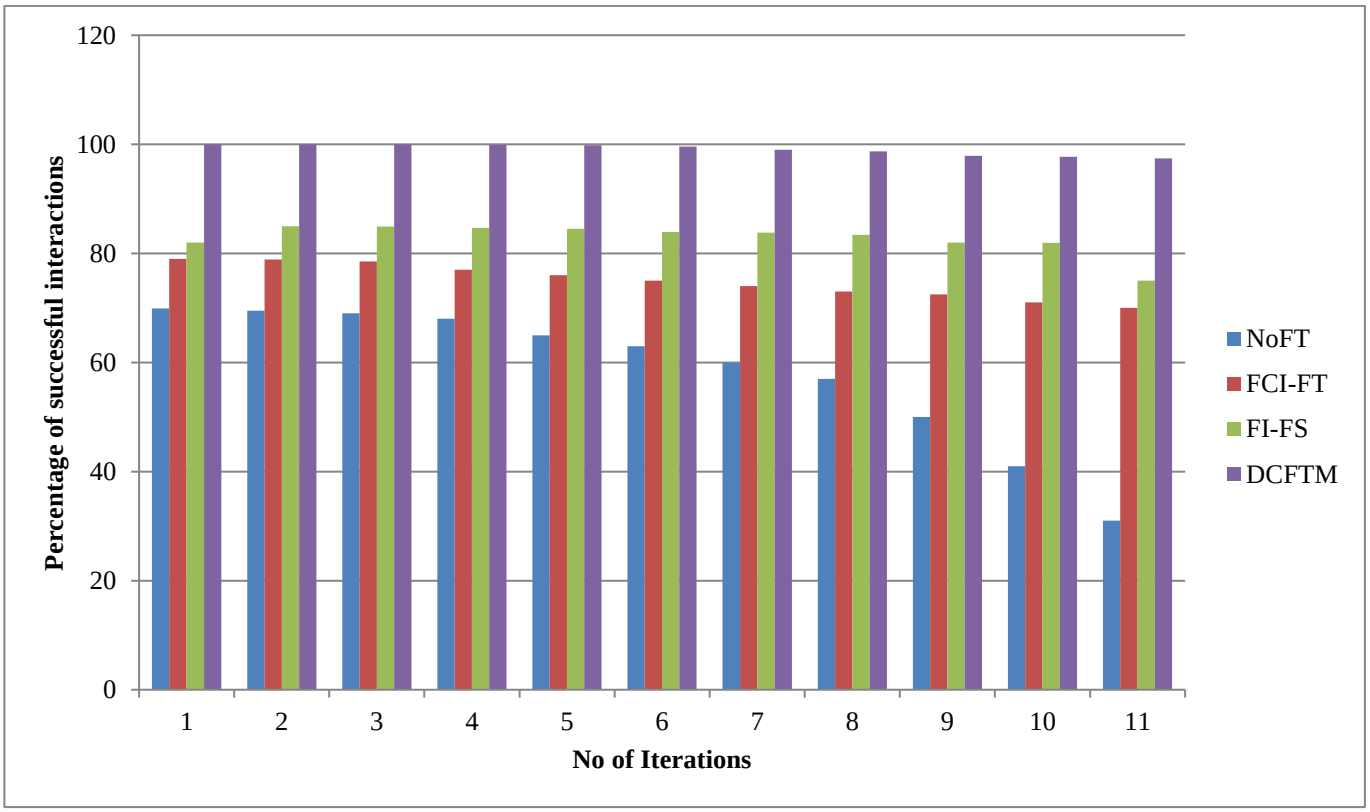

Fig. 11 Performance with 10% of critical configurations

As presented in Figure 11, the number of interactions is provided in the horizontal axis, and vertical access shows the percentage of successful interactions. The observations are made when 10% of critical configurations are used for empirical study. When no fault tolerance is used, the performance deteriorates. When the proposed algorithm DCFTM is used for fault-tolerant execution, it could achieve the best results by identifying the ideal candidate configuration.

6. Conclusion and Future Work

In this paper, we proposed an algorithm known as Dynamic Configuration of Fault Tolerance Mechanisms (DCFTM) to ensure that the system can withstand different kinds of faults at runtime and be resilient against faults. A case study enterprise application with distributed component-based architecture is built to evaluate the proposed fault-tolerant architecture and underlying DCFTM algorithm to prove the concept. The distributed reservation system supports various components that work together. They facilitate reservations pertaining to travel, including car, flight and hotel rooms. The implementation is made using Java and its Remote Method Invocation (RMI) technology. It has provisions for consistent transactions and fault tolerance. Experiments are made with the case study application. The empirical study revealed that the DCFTM algorithm outperforms state of the art. Though this paper proposes an algorithm for fault tolerance on top of distributed architecture, we believe it can be improved further. Therefore, in our future work, we intend to define an algorithm that will have not only fault tolerance but also suggestions for performance improvement.

References

[1] Mylara Reddy Chinnaiah, and Nalini Niranjan, "Fault Tolerant Software Systems Using Software Configurations for Cloud Computing," *Journal of Cloud Computing*, vol. 7, no. 3, pp. 1-17, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[2] Noor Bajunaid, and Daniel A. Menascé, "Efficient Modeling and Optimizing of Checkpointing in Concurrent Component-Based Software Systems," *Journal of Systems and Software*, vol. 139, pp. 1-13, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[3] Horst Schirmeier et al., "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," *2015 11th European Dependable Computing Conference (EDCC)*, pp. 245-255, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[4] D. Himabindu, and K. Pranitha Kumari, "Software Fault Prediction Using Machine Learning Algorithms," *International Journal of Computer Engineering in Research Trends*, vol. 9, no. 9, pp. 170-174, 2022. [Publisher Link]

[5] Mounya Smara et al., "Acceptance Test for Fault Detection in Component-Based Cloud Computing and Systems," *Future Generation Computer Systems*, vol. 70, pp. 74-93, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[6] Jing Liu, Jiantao Zhou, and Rajkumar Buyya, "Software Rejuvenation Based Fault Tolerance Scheme for Cloud Applications," *2015 IEEE 8th International Conference on Cloud Computing*, pp. 1115-1118, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[7] Thanh-Trung Pham, Xavier Défago, and Quyet-Thang Huynh, "Reliability Prediction for Component-Based Software Systems: Dealing with Concurrent and Propagating Errors," *Science of Computer Programming*, vol. 97, pp. 426-457, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[8] Fedor Y. Chemashkin, and Andrei A. Zhilenkov, "Fault Tolerance Control in Cyber-Physical Systems," *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pp. 1169-1171, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[9] Mukosi Abraham Mukwevho, and Turgay Celik, "Toward a Smart Cloud: A Review of Fault-Tolerance Methods in Cloud Systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589-605, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[10] Bentolhoda Jafary, and Lance Fiondella, "Component-Based System Reliability Considering Positive and Negative Correlation," *2018 Annual Reliability and Maintainability Symposium (RAMS)*, pp. 1-5, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[11] Alexander Aponte-Moreno, Cesar Pedraza, and Felipe Restrepo-Calle, "Reducing Overheads in Software-Based Fault Tolerant Systems Using Approximate Computing," *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1-6, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[12] Divya Rohatgi, and Gurmit Singh, "Improving Web Services Maintenance through Regression Testing," *International Journal of Computer Engineering in Research Trends*, vol. 3, no. 5, pp. 261–265, 2016. [Publisher Link]

[13] Jiguo Song, Gedare Bloom, and Gabriel Parmer, "SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems," *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 227-238, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[14] Shaoguang Shu, Yichen Wang, and Yikun Wang, "A Research of Architecture-Based Reliability with Fault Propagation for Software-Intensive Systems," *2016 Annual Reliability and Maintainability Symposium (RAMS)*, pp. 1-6, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[15] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy, "Architecting Resilient Computing Systems: A Component-Based Approach for Adaptive Fault Tolerance," *Journal of Systems Architecture*, vol. 73, pp. 6-16, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[16] Bowen Zheng et al., "Model-Based Software Synthesis for Safety-Critical Cyber-Physical Systems," *Safe, Autonomous and Intelligent Vehicles*, pp. 163-186, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[17] Sanjay Kumar Dubey, and Bhat Jasra, "Reliability Assessment of Component Based Software Systems Using Fuzzy and ANFIS techniques," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 1319-1326, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[18] Govind Prasad Arya, and Devendra Prasad, "Design of a System to Import Common Information of an Applicant from a Centralized Database While Filling Online Recruitment Application Form," *International Journal of Computer Engineering in Research Trends*, vol. 4, no. 1, pp. 30-32, 2017. [Publisher Link]

[19] Kuan-Li Peng, and Chin-Yu Huang, "Stochastic Modelling and Simulation Approaches to Analysing Enhanced Fault Tolerance on Service-Based Software Systems," *Software Testing, Verification and Reliability*, vol. 26, no. 4, pp. 276-293, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[20] Vidhyashree Nagaraju, Veeresh Varad Basavaraj, and Lance Fiondella, "Software Rejuvenation of A Fault-Tolerant Server Subject to Correlated Failure," *2016 Annual Reliability and Maintainability Symposium (RAMS*), pp. 1-6, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[21] Meng-Chu Chiang et al., "Analysis of a Fault-Tolerant Framework for Reliability Prediction of Service-Oriented Architecture Systems," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 13-48, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[22] R. Surendiran, "Secure Software Framework for Process Improvement," *SSRG International Journal of Computer Science and Engineering*, vol. 3, no. 12, pp. 19-25, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[23] Maskura Nafreen, Saikath Bhattacharya, and Lance Fiondella, "Architecture-Based Software Reliability Incorporating Fault Tolerant Machine Learning," *2020 Annual Reliability and Maintainability Symposium (RAMS),* pp. 1-6, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[24] J. VijiPriya, S. Suppiah, and Adeela Ashraf, "A Study on Development of Multilingual Dictionary Software," *International Journal of Computer Engineering in Research Trends*, vol. 4, no. 9, pp. 367-372, Sep. 2017. [Publisher Link]