JOURNAL OF INFORMATION AND COMMUNICATION TECHNOLOGY
https://e-journal.uum.edu.my/index.php/jict

How to cite this article:
Oliha, F. (2022). Guaranteeing performance in a fault tolerant architecture solution using software agent's coordination. *Journal of Information and Communication Technology, 21*(4), 595-625. https://doi.org/10.32890/jict2022.21.4.6

Guaranteeing Performance in a Fault Tolerant Architecture Solution using Software Agent's Coordination

Festus Oliha
Department of Computer Science,
University of Benin, Nigeria

oliha_festus@uniben.edu

Received: 17/4/2022 Revised: 26/7/2022 Accepted: 9/8/2022 Published: 20/10/2022

ABSTRACT

Performance is a critical attribute in evaluating the quality and dependability of service-oriented systems dependent on fault-tolerant architectures. Fault-tolerant architectures have been implemented with redundant techniques to ensure fault-tolerant services. However, replica-related overhead burdens fault-tolerant techniques with associated performance degradation in service delivery, and this consequentially discourages service consumers with discredits for service providers. In this paper, a fault-tolerant approach that adopts replication and diversity was employed on agent-oriented coordination toward guaranteeing the performance of the proposed fault-tolerant architecture solution under a large-scale service request load. In addition, the resultant architecture solution was simulated with Apache JMeter for performance evaluation considering the performability in the absence and presence of a fault load. The simulation experiments and results revealed the architecture's

efficiency in fault tolerance via the timely coordination of logical and replica-related activities by software agents. Noteworthily, the continued service availability and performance were guaranteed for the architecture solution with a significant rate of regularity in the absence and presence of a replica-related fault. Therefore, this study's performance evaluation methods and results could serve as a veritable milestone for building fault-tolerant service systems with appreciable performability and contribute to the service-oriented fields where performance is inevitable.

Keywords: Web services, service-oriented systems, fault-tolerant architecture, fault tolerance, performance, software agents, replication, diversity, computational intelligence.

INTRODUCTION

A fault is a defect that threatens the dependability of service-oriented systems. Suitable fault assessments are paramount to assure the system's performability in terms of fault tolerance and performance. Several studies have emphasized fault-tolerant architectures, their mechanisms, and criticality in ensuring service availability and delivery in service systems (Garcia & Toledo, 2007; Laranjeiro et al., 2008; Aghaei et al., 2011; Ahmed & Wu, 2013; Vargas-Santiago et al., 2017; Li et al., 2018; Pandey et al., 2019; Dahling et al., 2021). Relatively, existing fault-tolerant architectures have been centered around redundant mechanisms in terms of replication (active and passive), diversity, and N-Version Programming (Saha, 2005; Laranjeiro et al., 2008; Hosseini & Arani, 2015; Kumari & Kaur, 2018; Pandey et al., 2019; Abdi & Shahoveisi, 2022). Therefore, the effect of fault-tolerant mechanisms on the overall system's performance is critical to the service-oriented system communities and open to research.

Furthermore, redundancy and diversity have been highlighted as judicious approaches to implementing fault-tolerant architectures for web service-based systems. In view of the above, the cost of guaranteeing fault tolerance in these architectures is evaluated by the rate of regularity in the system's responsiveness upon service requests or consumption at a huge scale. Specifically, the architecture identifies the minimal effect of fault tolerance on the responsiveness or performance of a fault-tolerant service system under a service

replica failure or replica-related overheads over an interval of time. Performance is relatively measured with the attributes of response time (latency), throughput, and response stability – guaranteed responsiveness (Dobson & Sommerville, 2005; Garcia & Toledo, 2007; Laranjeiro et al., 2008; Lau et al., 2008; Carzaniga et al., 2009; Aghaei et al., 2011; Ladan, 2011; Ahmed & Wu, 2013; Hosseini & Arani, 2015; Sari & Akkaya, 2015; Kumari & Kaur, 2018; Li et al., 2018; Pandey et al., 2019).

Response time connotes the time it takes to respond to a service request. Throughput is the unit processing time for service requests, whereas guaranteed responsiveness is the rate and regularity of service response on a time interval. A defect in one or more attributes affects the performability of the service solution (Rickard & Oskar, 2017). Rickard and Oskar (2017) also asserted that approaches to fault tolerance in service systems may be consequential on their performance. This assertion connotes that performance overhead is presumed to have a high correlation with replica-related overheads due to: the fault-tolerant mechanisms on service replicas selection or switching in the presence of faults; and their dependencies on web services technologies – which are unavoidably prone to failure at invocation or runtime in an unpredictable network on the Internet (Saha, 2005; Lyu, 2011; Reddy et al., 2017; Pandey et al., 2018). Relatively, software agents are capable of logical service coordination at invocation time and have been described as fundamental for developing fault-tolerant service systems via triplicate redundancy (Saha, 2009).

It is unbearable to ensure fault tolerance with disorienting performance in service systems. As a result, this assertion motivated the researcher to evaluate the behavior of a fault-tolerant architecture solution in the context of guaranteeing performance over an interval of time. The researcher further proposed a fault-tolerant architecture and implemented a service solution using software agents' coordination capabilities; evaluated the performance of the resultant architecture solution via the metrics of response time, throughput, and guaranteed responsiveness in the presence and absence of a fault; and then reported the behavioral effects of its fault-tolerance on the performance of the architecture solution. The rest of the paper is organized as follows: Section Two describes related studies on building fault-tolerant service-oriented systems. Section Three projects the proposed

methodology: the architectural design, components, workability, and fault-tolerant approach. The experiments for performance evaluation are simulated with a load stressing tool, and the results are reported with analysis and discussion in Section Four. The paper ends with a conclusion and future direction.

RELATED STUDIES

A fault-tolerant system continues its functionality toward achieving a specified goal in the presence of faults. However, the absence of a fault-tolerant capability is a consequential loss, particularly to critical service systems dependent on web services for service provisions (Laranjeiro et al., 2008; Peng & Huang, 2014; Chimmanee & Jantavongso, 2016; Dahling et al., 2021). Succinctly, related studies on fault-tolerant approaches and mechanisms, performance of fault-tolerant mechanisms, and their reports were explored to guide the study's focus. Aghdaie and Tamir (2002) were motivated by the limitations of web services for fault tolerance and developed a client-transparent architecture with a replication approach. Their proposal was a modification of the Apache webserver to handle client requests in the presence of server failure. Their evaluation of latency, throughput, and processing cycles was reported with significant overheads via the replication schemes for supporting request failure. Therefore, the overhead could be associated with a significant loss in the system's performance under fault loads.

In view of the replication issues related to the developments of fault-tolerant web systems, Hong et al. (2005) proposed a replication (active and passive) approach for fault tolerance in distributed web systems. Performance was evaluated via request generator and LoadCube using the response time metric in the absence and presence of a fault. Their evaluation reported a better performance in the active scenarios than in the passive approach. The parameter of throughput and regularity rate in response time for service requests for an interval of time was a limitation in their study. The Universal Description, Discovery, and Integration (UDDI) failures were bridged in Oliveira et al. (2014) via a set of selection algorithms for new services when a component failure occurs. Their proposal depended on a novel selection mechanism employed in module logging and data mining to ensure the efficient selection of guaranteed web services. In the current study, a similar

concept was adopted in the selection of replica service solutions to ensure service availability for requests during service replica crashes.

Fault-tolerant designs were considered by Peng and Huang (2014) for the reliability of service-oriented architecture (SOA). Common fault-tolerant strategies were introduced to a reliability model for evaluating the overall reliability of service-oriented systems. A sensitivity analysis was employed to describe the results on the behavioral effect of the SOA-based systems. It was observed in their study that design faults could be propagated across service replicas due to replication and cause the system to perform unreliably. Furthermore, Rickard and Oskar (2017) evaluated the performance of a fault-tolerant system. Their study proposed a fault-tolerant architecture solution on replication and load balancing techniques for service availability against a fail-stop fault. The performance was evaluated under the fault load via response time. Their experiments monitored the replication effect on response time with the system's consistency. Nevertheless, the result revealed a performance overhead in terms of slower response time under different network loads and requests. The fault tolerance mechanism of replication and load balancing traded off performance for reliability.

The need for reliability and robustness was stressed in Pandey et al. (2019). They proposed a reliable mechanism for fault-tolerant web services via a novel framework using a replication manager approach. An algorithm was designed for fault prevention, and a pseudo-procedure (restart, retry, or reboot) was implemented for faulty service replacements. Fault loads included business faults, entry point failures, and network faults to evaluate the reliability of the proposed architecture. The evaluation showed a higher failure rate without the redundancy (retry or restart) mechanism, but reliability was improved under the resource fault load using temporal redundancy mechanisms. However, the performability of the architecture solution was associated with overheads in service response even though reliability was achieved with fault tolerance. The rate of regularity or stability of service responses tended to drop over an interval of time under a large-scale request.

Emphasizing replication across several nodes, Zhao (2007) rendered fault-tolerant web services by designing a fault-tolerant framework that is lightly weighted to ensure a consensus-based algorithm for

total ordering and consistency in replica membership. The study results revealed that the architecture incurred overhead in runtime, which moderately degraded the performability of the architecture solution. In summary, the review of related studies has been steered with motivations in fault-tolerant approaches, evaluation metrics, results, and limitations to garner insights on how to proffer tangible solutions to the performance overheads associated with fault-tolerant architecture for service systems. Previous literary works concisely exposed the challenges and opportunities to bridge the divides between fault tolerance and performance in service systems. In other words, the relevance in making service systems adequate on fault-tolerant capability with efficient service delivery demands features such as transparent self-healing, scalability, and guaranteed service responses. These attributes are not peculiar to software agents since fault tolerance is fundamental for building agent-based applications (Saha, 2009).

Software Agent Technologies in Fault-Tolerant Computing

Literary works revealed trends of software agents in fault tolerance and their adoption for computational intelligence as a result of their autonomous, proactive, adaptable nature, and inherent coordination capabilities for logical computations (Alhosban, 2013; Oliha, 2018; Alvi et al., 2019). Moreover, the popularity over the years has been impactful in distributed systems' implementation, and details of their approaches were documented in correlated literature (Saha, 2009; Alvi et al., 2019). However, Saha (2005) expressed that the use of triplicate redundancy has been prevalent for fault tolerance by software agents regardless of their minimal support for design diversity during execution time for interactive applications. Another study added that their humanlike nature makes them autonomous and suitable for handling fault tolerance in critical systems (Dahling et al., 2021).

Unarguably, software agents are highly capable of coordinating service solutions because of their self-healing attributes. Besides self-healing capability, they are efficient in activities involving execution before and at runtime – logical activities (Saha, 2005; Alhosban, 2013; Erlank & Bridges, 2018) and managing byzantine or arbitrary faults (Alvi et al., 2019). With such intelligent nature, a collection of two or more agents collaborating to achieve a common task comprises

a Multi-Agent System (MAS). However, service systems depend on web services that must be synchronized as agent services to enable seamless integration via web services integration gateway (WSIG). WSIG facilitates software agents to communicate with web services by synchronizing Simple Object Access Protocol (SOAP) messages as Agent Communication Language (ACL) messages (Bellifemine & Greenwood, 2007; Calisti et al., 2010). The tools and platforms supporting seamless integration and communication between both technologies have been described in Bellifemine and Greenwood (2007).

According to Saha (2009), the fundamental for developing an agent-based service system is highly dependent on fault tolerance. Their capabilities included efficiently coordinating logical activities associated with replication at runtime for web services executed as agent services to avoid failure at invocation time (Lyu, 2011; Kumar, 2015; Li et al., 2018). Therefore, a feasible panacea to replica-related overheads is the adoption of software agents for redundant logical activities due to their support for fault tolerance through triplicate replication.

THE PROPOSED METHOD

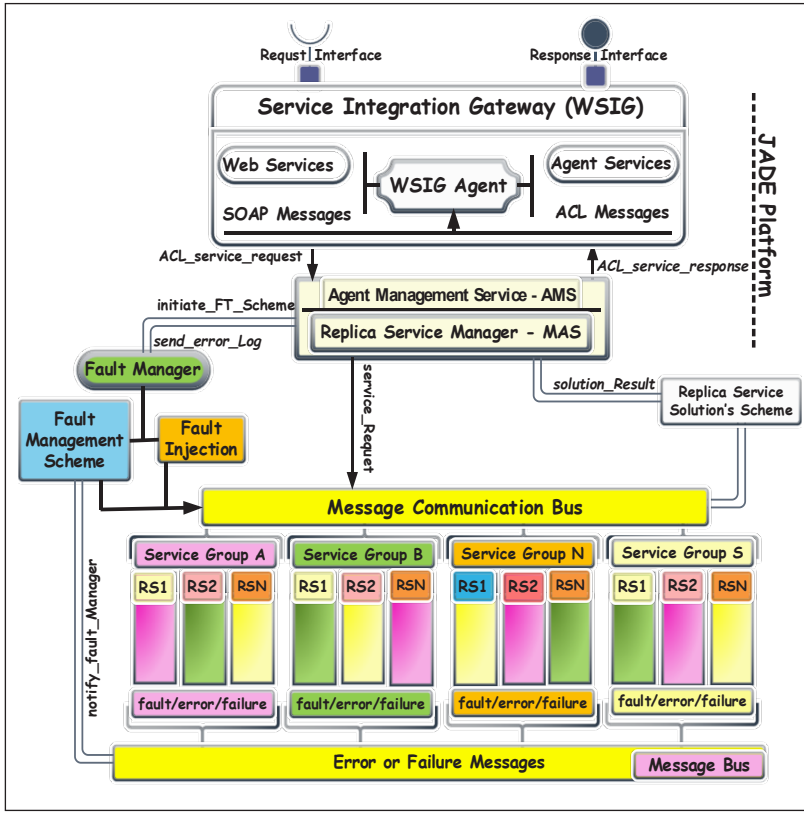
The Proposed Architecture

The theoretical framework in this study was initiated with an introduction of a fault-tolerant architecture that adopted an agent-oriented approach for implementing fault-tolerant mechanisms. Figure 1 shows the proposed architecture with software agents as the major component for integrating existing time-based fault-tolerant techniques for logical activities. The architecture was component-based, where each component had unique services to attain a common goal of guaranteeing efficiency in service availability, reliability, and delivery. The architecture was implemented on a series of activities that were activated with a service request (and response) component via an interface for the client system to consume a targeted computational service. This request was originally sent as a SOAP message, parsed in an extensible markup language (XML) format for web services. A conversion was executed via the WSIG gateway to exchange this message format with the service provider. WSIG converted SOAP

messages to ACL messages to enable software agents to consume web services as agent services and vice versa.

Figure 1

The Proposed Fault-Tolerant Architecture with Agent Coordination Services



Software agents were hosted in the Java Agent DEvelopment (JADE) environment and their lifecycles were under the agency of Agent Management Services (AMS) coordination. JADE was one of the most suitable platforms for multi-agent systems (Leitao et al., 2016; Dahling et al., 2021). The service replica solutions were components with N-Versioned solutions built by different service vendors transparent to each other while providing computational services. Each replica solution was a collection of at least four N-Version sets of replica groups with n-1 of them that were active and running concurrently for

each group alongside a passive standby. The proposed architecture was anticipated with the capability of checking design and code level faults, byzantine (incoherent result), and crash faults under large-scale user requests. The replica-crash fault was uniquely emphasized to test the fault-tolerant mechanism via a fault injection scheme. The impact of the fault was communicated to the replica manager, and appropriate action was initiated for continuity.

It is important to note that AMS was the major strength and heart of the architecture because it was a component highly responsible for multi-agent services, including creation, registration, behavior, communication, deletion, and agency under the guidelines of the Foundations for Intelligent Physical Agents specification. It is also saddled with the coordination of all logical activities, as depicted in Figure 2 via the replica manager involving replica group creation, replica addition, and replica process management for every replica solution. In this way, service requests, service response selection, fault management, service replicas, and service response were all managed by MAS, which was unique to the proposed architecture for guaranteeing performance in service delivery. As considered in literary works (Shafiq et al., 2006; Zhao, 2007), a lightweight client-transparent service known as a grade point average (GPA) solution was emphasized to test the fault tolerance mechanism and examine the solution's behavior under large-scale simultaneous requests. Consequently, latency was increased on service responses intentionally to crash the system or degrade its performance.

The drawback of this architecture was capped at its capability to only tolerate replica crash faults at the application layer. Replica crash faults at the network layer may yield varying outcomes for fault tolerance and performance. However, application layer fault (replica crash) is a threat to the availability of services in fault-tolerant systems (Qian et al., 2018; Pandey et al., 2019) and threatens the survivability, performability, and dependability of service systems dependent on web services in general.

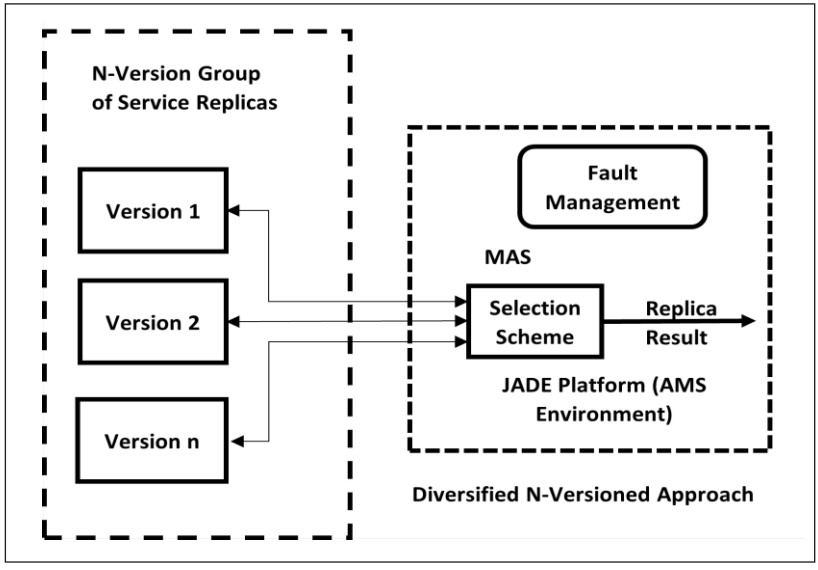
The Fault Tolerant Approach and Replica Management

The fault-tolerant approach adopted in the current study subsumed replication and diversity, considering software agents' capability of fault tolerance by triplicate replication (Saha, 2009), though not in design diversity. Replication was adopted to ensure service

availability by building service replicas. Diversity ensured that each service replica group was built by different service vendors with exact functionalities, involving at least four N-Version sets of replica groups with n-1 of them that were active and running concurrently for each group and a passive standby, as captured in Figure 1. The coordination of all replica-related logical activities was governed by software agents – MAS as seen in Figure 2. However, fault propagation is a defect of the replication approach (Aghaei et al., 2011). A design and code diversity approach was implemented to handle service defects and faults propagation associated with the replication approach. For this reason, faults could not be propagated to other replicas since these replicas were of different service vendors.

Figure 2

Service Replica Coordination



At the occurrence of a replica fault, a transparent service provision was continued, while a retry or restart protocol was initiated for the failed replica by the fault manager under the coordination of MAS. Nevertheless, the fault-tolerant mechanism process of replica switching in some studies revealed overhead in latency with downtime in service response – performance. The current study implemented a selection algorithm for solution activities associated with service

replicas, where selection was done to avoid delay in response time. Selection algorithms have been noted in similar concepts in literary works (Zhao, 2007; Oliveira et al., 2014; Pandey et al., 2019; Santish et al., 2022), though in fault-tolerant mechanism selection or primary available service selection and not in replica result selection. Consequently, replica result selection was handled with the pseudocodes in Algorithm 1, with subsets in Algorithms 1a and 1b to ensure the optimal selection of appropriate service replica responses.

Algorithm 1: ***Selection Algorithm***

Replica Result Selection Algorithm

Output: *S* – coherent replicas' solution result
Inputs: Array of replica solution results
Initialization: *Let S be a set of replica solutions;*
Let n be the size of the adjacency matrix for S solutions;
Import ArrayList properties;

Begin

```
1.    public class selectionAlgorithm {
2.        private final String[] solutions; //solution set
3.        private int [ ] [ ] SolMatrix; //matrix solution
4.        private int n ← 0;
5.        private ArrayList<integer> finalSol ← new ArrayList<>( );
6.        public SelectionAlgorithm (String[ ] set) {
7.            this.solutions ← set;
8.            n ←solutions.length;
9.        //call selection process phases
10.       phase0( ); //form matrix from n by n replica set of solutions
11.       phase1( ); //scanning phase - rows and columns
12.       finalSol ← phase3(phase2());
13.       }
14.       public ArrayList<Integer> getFinalSol() {
15.           return finalSol;
16.       }
17.   } //end selectionAlgorithm
```

Algorithm 1a: Selection Process Phases

```
//phase0 – matric formation
1.  private void phase0( ) {
2.        SolMatrix ← new int[n][n];
3.        //initialize all to -1 and and where i=j to -2
4.        for(int i ← 0;i<n;i++)
5.           for(int j ← 0;j<n;j++)
6.              if(i←j)
7.                 SolMatrix[i][j] ← -2;
8.              else
9.                 SolMatrix[i][j] ← -1;     }
   // phase 1: scanning phase
10. private void phase1( ){
11.       for (int i ← 0;i<n;i++)
12.          for(int j ← i+1; j<n;j++)
13.             if(SolMatrix[i][j] ← -1)
14.                if(solutions[i].equals(solutions[j])){
15.                   SolMatrix[i][j]←1; SolMatrix[j][i] ← 1;
16.                }else{
17.                   SolMatrix[i][j]←0; SolMatrix[j][i] ← 0;
18.                }
19.   }
```

Algorithm 1b: Selection Process Phases Cont'd

```
//phase 2: row counts
20.        private int[] phase2(){
21.           int[] C ← new int[n];
22.           ArrayList<Integer> S_stack ← new ArrayList<>();
23.           for(int i ← 0;i<n;i++){
24.              C[i] ← 0;
25.              if(!S_stack.contains(i))
26.                 for(int j ← i+1;j<n;j++){
27.                    C[i] +← SolMatrix[i][j];
28.                    if(SolMatrix[i][j]←1) //push to S_stack
29.                       S_stack.add(j);
30.                 }
31.           }
32.        return C;
33.        }
   //phase 3: row selection for most correct result
34.        private  ArrayList<Integer> phase3(int[] C){
35.           int p ←0;//p for position
36.           int max ← C[p];
37.           for(int i ← 1;i<n;i++)
38.              if(C[i] > max){
39.                 max ← C[i];
40.                 p ← i;
41.              }
42.         ArrayList<Integer> F_stack ← new ArrayList<>();
43.           for(int i←0;i<n;i++)
44.              if(C[i]←max) //push to F_stack - result with same
Max count
45.                 F_stack.add(i);
46.           return F_stack;
47.        } //end of phases
```

For phase0(), consider the sub-computations for the algorithm:

Let R = result set for solution S, where Ri = n, i = {2.0, 2.2, 3.0, 3.5, 2.0, 3.08} and n = 6

The following n x n matrix is created for the set of solution results R,

The matrix is generated using the loop from lines 4 to 9, where the intersection is replaced by -2 and others with -1 values.

Thus, the matrix generated is represented in Table 1.

Table 1

Generated Matrix

	R_1	R_2	R_3	R_4	R_5	R_6
R_1	-2	-1	-1	-1	-1	-1
R_2	-1	-2	-1	-1	-1	-1
R_3	-1	-1	-2	-1	-1	-1
R_4	-1	-1	-1	-2	-1	-1
R_5	-1	-1	-1	-1	-2	-1
R_6	-1	-1	-1	-1	-1	-2

Since n = 6, the loop will require 36 iterations for an n x n matrix.
To optimize these iterations, **phase1()** scans through the matrix to check if $R_i = R_j$ and replaces the content with 1's else 0's if and only if $R_i \neq R_j$ for all -1.
In this phase, the rows R_i are checked against the columns R_j for all -1 entries given that:

n = set of replica results R_i for all -1;

$R_i = \{R_1 = 2.0, R_2 = 2.2, R_3 = 3.0, R_4 = 3.5, R_5 = 2.0, R_6 = 3.08\}$; $R_i=R_j=1$; and $R_i \neq R_j$.

Thus, the number of iterations can be optimized with Equation 1:

$$R_{ij} = \sum_{i=1}^{n}(n-1) \quad (1)$$

Where R_{ij} is the number of iterations for n number of replica solutions.

$\forall Ri = -1$; if i = 1 then $R_i = R_1$ and R_1 is checked against all columns of all R_j except for the intersection of the row against the column. Therefore, the optimization occurs from the row-wise comparison for the 1's and the following pair of computations creates a new matrix via the reduction in Equation 2:

for $i = 1, j = i+1,$ *and* $R_i \neq R_1$, then $R_1 = \{R_1{:}R_2, R_1{:}R_3, R_1{:}R_4, R_1{:}R_5, R_1{:}R_6\}$ … 5 iterations
for $i = 2, j = i+1,$ *and* $R_i \neq R_1$, then $R_2 = \{R_2{:}R_3, R_2{:}R_4, R_2{:}R_5, R_2{:}R_6\}$ $R_2 = R_5=1$ … 4 iterations
for $i = 3, j = i+1,$ *and* $R_i \neq R_1$, then $R_3 = \{R_3{:}R_4, R_3{:}R_5, R_3{:}R_6\}$ … … 3 iterations
for $i = 4, j = i+1,$ *and* $R_i \neq R_1$, then $R_4 = \{R_4{:}R_5, R_4{:}R_6\}$ … … 2 iterations
for $i = 5, j = i+1,$ *and* $R_i \neq R_1$, then $R_5 = \{R_5{:}R_6\}$ $R_5=R_2=1$ … … 1 iteration
for $i = 6, j = i+1,$ *and* $R_i \neq R_1$, then $R_6 = \{\text{ignored}\}$

Following this, Equation 1 becomes Equation 2:

$$Rij = [(n2 - n)/2] \quad (2)$$

Equation 2 represents the optimized iteration of n set of replica solution results, R. The number of iterations is reduced to 15 as a result of the

stepwise decrement by (n - 1) in Equation 1. The matrix pairs are represented in Table 2 with 0's except for the pair of {R2:R5} = 1.

Table 2

Matrix Pair

	R_1	R_2	R_3	R_4	R_5	R_6
R_1	-2	0	0	0	0	0
R_2	0	-2	0	0	1	0
R_3	0	0	-2	0	0	0
R_4	0	0	0	-2	0	0
R_5	0	1	0	0	-2	0
R_6	0	0	0	0	0	-2

The vitality of this algorithm was to ensure rapid and optimal response of solution results to service requests irrespective of failed replicas, and this optimization was noticeable in phase1() in Algorithm 1a. n x n matrix of size 6 gave 36 iterations, but the algorithm executed it in $[(n^2-n)/2]$ iterations (where n must be ≥ 3) and stored the outcome unto a stack in phase2() while popping the highest occurrence in phase3(). Algorithm 1 implemented the different phases to achieve the desired objective for n set of replica solutions via the capability of software agents, ensuring service availability, regularity, and stability in the presence of a failed replica service(s). This not only ensured fault tolerance but also guaranteed the availability of services and reliability of the results. With these alignments, the architecture solution would be unburdened by the replica result selection issues. Furthermore, the efficiency of service responsiveness over an interval of time could be guaranteed for a fault-tolerant service system during a failed replica or switching among services for a response.

Fault Scenarios and Performance Measurement

Faults were injected into the architecture solution to assess fault tolerance and its solution behavior in different scenarios or conditions. The fault types and scenarios involved the following:

1. Propagation faults: this is a replica type of fault that is propagated via replication of service functionalities to other

replicas. In this scenario, a replica solution code is mutated to see if it affects the replica results during the replication of services by the replica manager.
2. Replica crash fault: a scenario where a replica service is killed from an N-Versioned set of the same replica group. The computational solution of such a replica is never returned, but service responses are returned for others.
3. Group crash fault: another similar scenario where the entire replica group of N-Versioned set is crashed, and computational service response is denied from that group.

Furthermore, the given fault conditions were expectant of performance overheads over time – that is, replica-related faults were injected to cause some service replica overheads. However, the impact on performance was evaluated during runtime via the metrics of response time and throughput by the service agents. Therefore, evaluating the architecture solution over an interval of time was paramount to examining the rate of responsiveness or solution behavior under the fault conditions.

EXPERIMENTS AND RESULTS

Experiment Setup and Configurations

Apposite configurations were vital to assess the fault-tolerant approach's impact on the performance of the architecture solution over time. Necessary files, addons, and plugins were extracted into relevant directories and the multi-agent systems were configured using the JADE platform as the agency for housing running agent services. The experimental setup involved configuring parameters, directories, libraries, system environment, and user variables in preparation for the evaluation. The simulation environment was mimicked with Apache JMeter for virtual service requests users, web services consumption, and report generations for each user group on a Lenovo G470 system with 4GB RAM and a Processor of Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz (4 CPUs), ~2.6Ghz. Apache JMeter was selected as the evaluation tool in this study. It was feasible for load-stressing web applications in simulating their behavior using corresponding performance indicators, such as response time, throughput, error percentage of response to request rate, and standard deviation, for the rate of regularity or deviation in response time.

The performance experiment was configured for two versions of the architecture solution, i.e., the absence and presence of fault with a configuration of 25,000 requests per unit time at a large scale from ten random service request channels. Software testing and quality assurance revealed that the requests were processed, and responses were delivered successfully within a stable state. The behavior of the architecture solution in the presence of faults needed to be assessed to determine its guaranteed performability.

Fault Injection

This study adopted the compile-time technique, whereby the replica crash fault was injected into the architecture solution to ascertain its fault-tolerant capability and performance in the presence of faults. This fault injection technique has been used in literary works to test the dependability of software systems (Looker et al., 2004; Zaide et al., 2004; Hossain, 2006; Rychly & Zouzelka, 2012; Ramakrishnan et al., 2014; Umadevi & Rajakumari, 2015). The fault injection mechanism employs code mutation, which modifies or refactors (Almogahed & Omar, 2021) a subsection of the architecture solution code at compile time to activate replica-related faults conditions at runtime such that:

1. a fault propagated to service replica solution produces arbitrary and incoherent responses (byzantine);
2. if it crashes some replica solution processes, then an error of request timeout, connection lost, or service unavailable is returned from the affected group processes;
3. if it crashes the entire group of service replicas, then delay or latency overheads are experienced, causing inconsistent regularity in service delivery.

Subsequently, the experiment results for the architecture solution were extracted, as represented in Figure 3.

Figure 3

Evaluation Summary Report for the Architecture Solution in the Presence and Absence of Fault

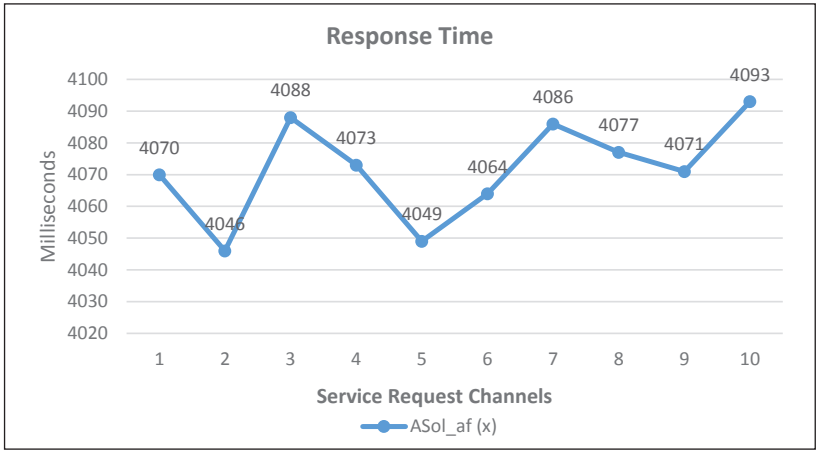
	Data Capture of Summary Reports for both Versions of the Experiments: ASol_af (X) & Asol_pf (Y)													
Service	Sample Size		Response Time AVG		Guaranteed Resp		Throughput		Response Time Max		Error%		Average Bytes	
Channels	ASol_af	Asol_pf	ASol_af	Asol_pf	ASol_af	Asol_pf	ASol_af	Asol_pf	ASol_af	Asol_pf	ASol_af	Asol_pf	ASol_af	Asol_pf
1	2492	2491	4070	1174	653.08	169.52	0.2	0.8	7615	2565	0.00%	0.00%	2241.2	2247.4
2	2462	2434	4046	1180	679.04	173.83	0.2	0.8	7258	2343	0.00%	0.00%	2243.1	2247.5
3	2455	2443	4088	1163	651.07	165.89	0.2	0.8	7071	2257	0.00%	0.00%	2243	2153.7
4	2536	2545	4073	1174	666.68	164.5	0.2	0.9	10885	2270	0.00%	0.00%	2150.2	2248
5	2493	2534	4049	1176	633.67	166.83	0.2	0.9	6690	2330	0.00%	0.00%	2150.6	2430.1
6	2506	2555	4064	1172	656.22	168.58	0.2	0.9	6864	2469	0.00%	0.00%	2242.3	2336
7	2495	2509	4086	1175	672.33	166.85	0.2	0.9	7414	2178	0.00%	0.00%	2243.8	2247.7
8	2514	2524	4077	1170	630.02	160.77	0.2	0.9	7347	2111	0.00%	0.00%	2334.2	2154.3
9	2551	2500	4071	1167	648.29	160.65	0.2	0.8	7001	2114	0.00%	0.00%	2151	2246.5
10	2496	2465	4093	1177	657.26	159.49	0.2	0.8	6937	2410	0.00%	0.00%	2425	2155.1
Total	**25000**	**25000**	**4072**	**1173**	**655.06**	**165.8**	**2.4**	**8.5**	**10885**	**2565**	**0.00%**	**0.00%**	**2242.1**	**2247.3**

The extracted result data contained a sample request size of 25,000 on ten different service channels, response time with average intervals, etc. The experiments in the absence (ASol_af) and presence (ASol_pf) of a replica fault revealed the architecture solution's performability with an error response rate of 0.00 percent (100% error-free) to all service requests from the designated channels. On average, the architecture solution in the absence of a replica fault responded with about 4,072 ms to a sample request of 25,000 being processed at 2.4 s for every 2.19 Kbytes of requests. Moreover, 2.19 Kbytes were processed with a throughput of 8.5 s for 25,000 sample requests at an average response time of 1,173 ms for ASol_pf. The deviation from the regularity of expected response time was capped at 655.6 for the architecture solution in the absence of a fault and 165.8 in the presence of a fault.

The graphical implications were represented to visualize the behavior of the architecture solution for both scenarios. Based on the experimental result data, the response time and the responsiveness rate were indicated by standard deviation from the summary reports (guaranteed responsiveness), which were graphically represented for both architecture solutions in Figures 4–7. ASol_af represented the architecture solution in the absence of replica fault with the legend – X, and Asol_pf signified the architecture solution in the presence of fault with the legend – Y. The average throughput for both scenarios was greater than 5 s for processing requests/responses for the experiments. The higher the throughput, the better the outcome (Ladan, 2011).

Figure 4

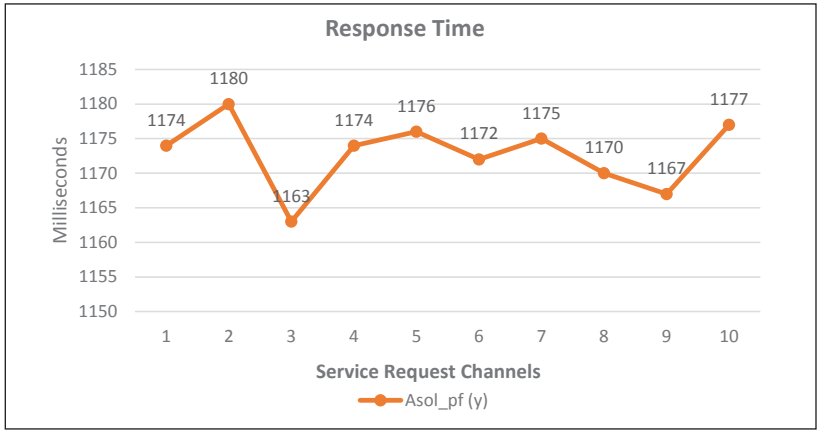
Line Graph for Response Time – Asol_af (X)



The average response time for Asol_af from the ten different request channels was 4,072 ms. This finding depicted the average number of responses processed per unit time from each service response channel for ASol_af. Similarly, the highest and lowest points for ASol_pf were 1,180 ms and 1,163 ms response per millisecond. Meanwhile, the average response time was capped at 1,173 ms for the ASol_pf architecture solution.

Figure 5

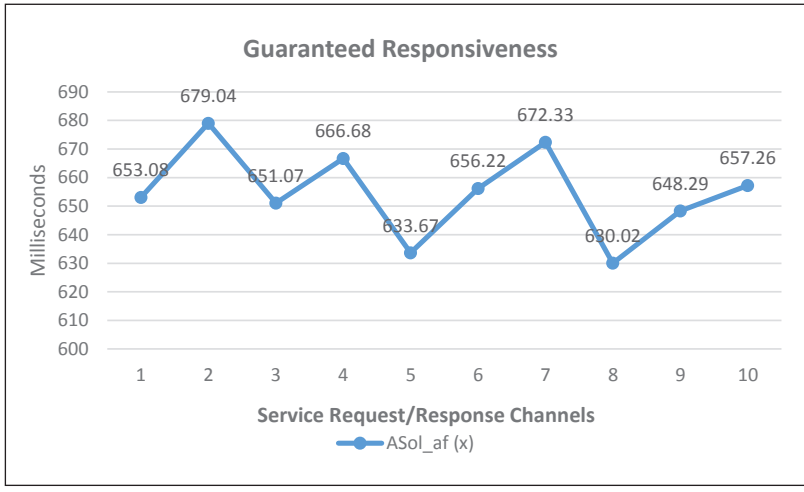
Line Graph for Response Time – Asol_pf (Y)



From the graphs in Figures 4 and 5, the average difference in response time for ASol_af was summated as 47 ms from the peak and lowest response times. This result signified the range at which the service responses were guaranteed or slightly distorted for service provision upon requests. Furthermore, the average difference in response time for ASol_pf was summed as 17 ms, indicating a better time frame for guaranteeing service responses to requests. The line graphs in Figures 6 and 7 captured the graphical representations for the performance attribute of guaranteed responsiveness for the architecture solution. Guaranteed responsiveness is a product of the standard deviation result data, denoting the rate of regularity at which service responses are guaranteed over an interval of time with or without a fault presence. This regularity depicted the fault tolerance ability and responsiveness of service systems.

Figure 6

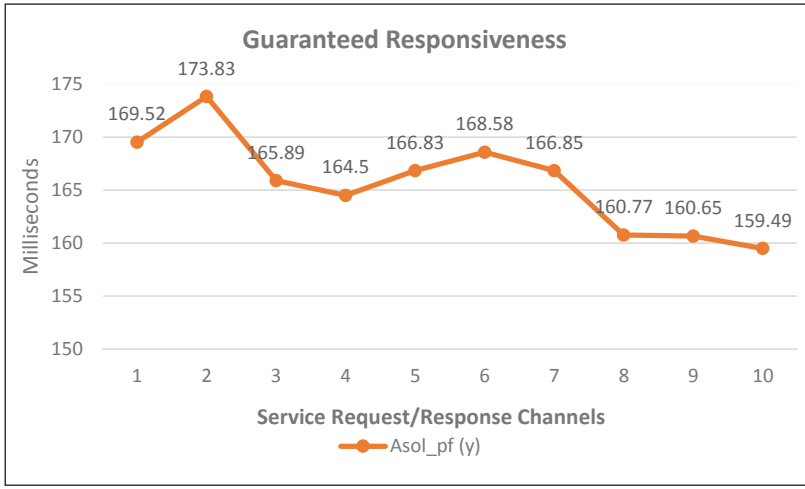
Guaranteed Responsiveness Line Graph for ASol_af (X)



The line graph for ASol_af in Figure 6 concerning regularity in responses displayed a zigzag outcome. It is hard to tell which service response channel maintained regularity as the peak was capped at channel 2 and the lowest at channel 8. This result indicated that service responses were guaranteed within a time frame of 630.02 ms and 679.04 ms as the lowest and peak times.

Figure 7

Guaranteed Responsiveness Line Graph for ASol_pf (Y)



The line graph of ASol_pf in Figure 7 showed a maintained regularity from the service response channel for client 6, from which elevation was decreased below the point of initiation at channel 1. The peak guaranteed response time was capped at an interval within 173.83 ms, with a low response time of 159.49 ms at channel 10. This finding indicated that the response time rate was more guaranteed for ASol_pf than ASol_af. From Figures 4–7, ASol_pf projected better in both response time and guaranteed responsiveness than ASol_af. The ASol_pf vs ASol_af line graphs were plotted against each other for response time and guaranteed responsiveness, respectively, in Figures 8 and 9.

Figure 8

ASol_af (X) vs ASol_pf (Y) Line Graph for Response Time

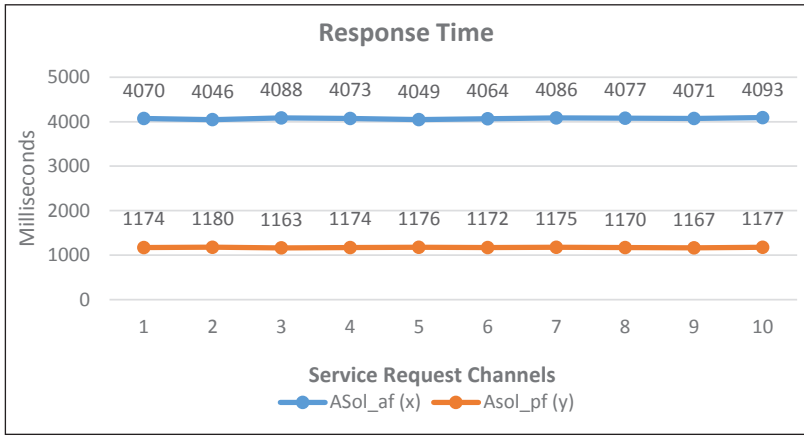
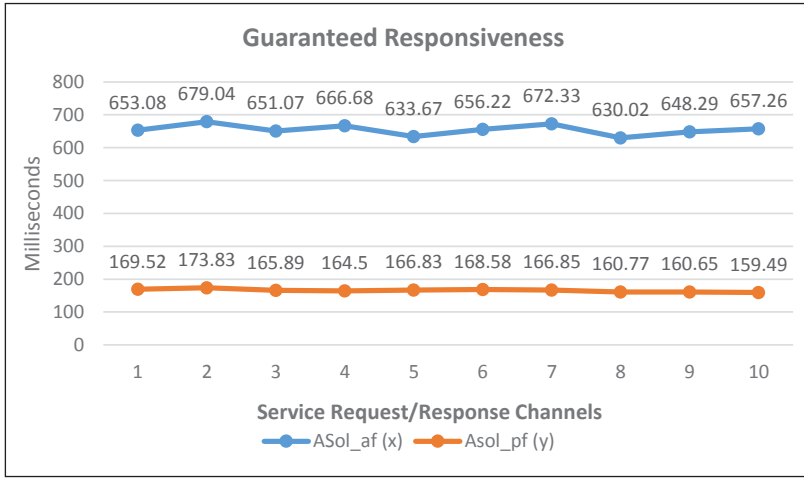


Figure 9

ASol_af (X) vs ASol_pf (Y) Line Graph for Guaranteed Responsiveness



Figures 8 and 9 showed that the performance difference was parallel for both attributes, i.e., there was a huge performability difference for both architecture solutions. Nevertheless, ASol_pf's performance was more appreciated as it reflected regularity in guaranteeing service

response even under a replica fault. Bar graphs were also plotted in Figures 10 and11 to confirm and buttress the architecture's strength in guaranteeing performance in service systems.

Figure 10

Average Performance Bar Graph for Response Time – ASol_af (X) vs ASol_pf (Y)

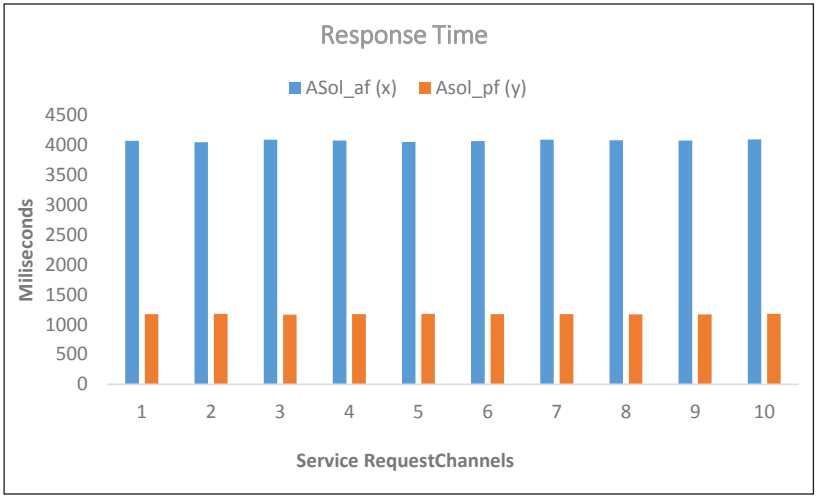
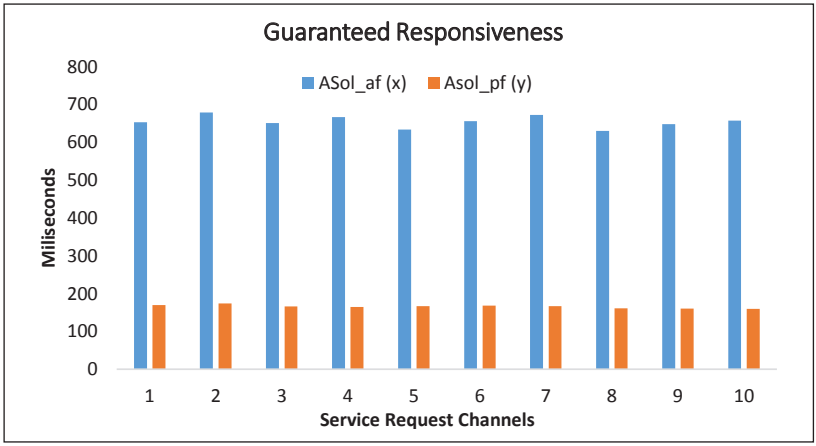


Figure 11

Average Performance Bar Graph for Guaranteed Responsiveness – ASol_af (X) vs ASol_pf (Y)



For clarity, the rate of regularity in response time was better comprehended in Figures 10 and 11. The average performance bar graph for response time and guaranteed responsiveness depicted the range of regularities. The service response for ASol_af was only possible for an insignificant deviation range of 49.02 ms for the targeted request size but with consistency in the regularity range. Moreover, ASol_pf recorded a service consistency response regularity with a deviation range of about 14.34 ms per request size. Better service responsiveness with an insignificant variation in service delivery was noted in ASol_pf. The performability of the architecture solution was good for both versions; however, better performance was guaranteed for the ASol_pf version of the deployed architecture solution for service delivery. The resultant efficient service delivery credited the performability of the architecture solution with an insignificant variation in guaranteeing the system's behavior – performance. The result findings were also synonymous with literary works in performance computing (Hong et al., 2005; Peng & Huang, 2014; Pandey et al., 2019).

DISCUSSIONS

Saha (2009) discoursed that "software-based fault tolerance is the use of technologies to enable the continued delivery of services at an acceptable level of performance after a design fault becomes active". This study realized assertion via the use of software agents' coordinating capabilities to manage replica-related overheads. A fault-tolerant architecture solution was implemented, the resultant solution was simulated with Apache JMeter for performance experiments, and the evaluation results were documented and analyzed.

From the experiments, the architecture solution demonstrated capability in handling replica-related crash faults with support for accurate responses to service requests at a 0.00 percent error return rate on an average response time of 4,072 ms, and a throughput of 2.4 s captured for ASol_af. At the same time, an improved average response time of 1,173 ms was capped for ASol_pf. Additionally, a better processing time of 8.5 s as ramp-up time was captured for ASol_pf in handling services provision at a large scale of 25,000 sample requests, as depicted in Figure 4. The performance of most service-related literary works was adjudged with the attributes of response time and high throughput (Ladan, 2011; Bora & Bezborual,

2015; Kumar, 2015). A high throughput indicated good performance. ASol_pf had a throughput of 8.5 s, while ASol_af had 2.4 s. A better performance was found for the architecture solution when under a fault load.

The experimental results from the graphs in Figures 4 and 5 showed that the average difference in response time for ASol_af and ASol_pf was summated as 47 ms and 17 ms, respectively, indicating a better time frame for guaranteeing service responses to client requests. Likewise, the average rate at which the response regularities were in variation for the architecture solutions was within a time frame difference of 49.02 ms for ASol_af and 14.34 ms for ASol_pf. It is indicative that the architecture solution in the presence of fault guaranteed more performability in response time and guaranteed responsiveness by about 36.2 percent and 29.3 percent over ASol_af. The outcomes of the simulated experiments and evaluation were in harmony with literary works in service-oriented system communities (Hong et al., 2005; Calisti et al., 2010; Alvi et al., 2019; Pandey et al., 2019). Convincingly, the architecture solution's performance was contrary to the latency and replica-related overheads observed in some other studies (Aghdaie & Tamir, 2002; Li et al., 2005; Zhao, 2007; Rickard & Oskar, 2017; Li et al., 2018; Dahling et al., 2021).

The study findings are worth noting that:

1. The architecture solution demonstrated robustness in fault tolerance with efficiency in the system's performance at a very large-scale service request and response provision.
2. The architecture solution's behavior in the presence and absence of faults unveiled a matching uniformity of regularity in service delivery.
3. There is a strong indication that the performance of the fault-tolerant service-oriented architecture solution is guaranteed and even better in the presence of a replica crash fault because of the computational intelligence of multi-agent technology in coordinating the logical-replica-related services.

CONCLUSION

In this study, fault was guaranteed by building service replicas with replication and a diversified N-Versioned redundancy approach

to ensure service availability and regularity in provision/delivery. Replica-related overheads were managed via MAS services by coordinating all logical time-related activities (replica creation, service registry facilitators, service registration, service request, binding, distribution and communication, replica solution services, fault injection, and replica solution results) to ensure efficient service delivery in the presence and absence of replica faults. The fault tolerance and performance of the architecture solution were evaluated with attributes of response time, throughput, and guaranteed responsiveness. Therefore, the study affirms that the architecture solution is efficient in guaranteeing regularity in service delivery for service systems deployed on fault-tolerant architecture. This efficiency is thereby credited to software agents' coordination intelligence. Software agents are emphasized as one of the credible solutions for implementing logical activities associated with service replicas for fault-tolerant service systems dependent on web services.

The approach, results, and findings of this study are in no attempt to condemn service systems but to contribute to fault-tolerant computing research where guaranteeing performance is inevitable. Nonetheless, for the advancement of knowledge, the need to further subject the results to statistical interpretations is vital to assess the confidence rate at which service responses are guaranteed, particularly in the presence of faults.

ACKNOWLEDGMENT

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

Abdi, A., & Shahoveisi, S. (2022). FT-EALU: Fault tolerant arithmetic and logic unit for critical embedded and real time systems. *Hardware Architecture.* 1–15. https://doi.org/10.48550/arXiv.2-204.01262

Aghaei, S., Khayyambashi, M. R., & Nematbakhsh, M. A. (2011). A fault-tolerant architecture for web services. In *2011 International Conference on Invocations in Technology (IIT)* (pp. 53–56). IEEE. https://doi.org/10.1109/INNOVATIONS.2011.5893867

Aghdaie, N., & Tamir, Y. (2002). Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proceedings of the IEEE International Conference on Computer Communications and Networks, Miami, Florida* (pp. 63–68). https://doi.org/10.1109/ICCCN.2002.1043047

Ahmed, W., & Wu, Y. W. (2013). A survey on reliability in distributed systems. *Journal of Computer and System Sciences*, *79*, 1243–1255. https://doi.org/10.1016/j.jcss.2013.02.006

Alhosban, A. A. (2013). *Fault management for service-oriented systems* (Doctoral dissertation, Wayne State University, Detroit, Michigan). https://digitalcommons.wayne.edu/oa_dissertations/745

Almogahed, A., & Omar, M. (2021). Refactoring techniques for improving software quality: Practitioners' perspectives. *Journal of Information and Communication Technology*, *20*(4), 511–539. https://doi.org/10.32890/jict2021.20.4.3

Alvi, A. B., Hashmi, M. A., Chuban, Z. H., Atif, M., & Ahmed, I. (2019). Adaptive byzantine fault tolerance support for agent oriented systems: The BDARX. *International Journal of Advanced and Applied Sciences*, *6*(2), 57–64. https://doi.org/10.21833/ijaas.2019.02.009

Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE*. John Wiley & Sons Ltd: West Sussex.

Bora, A., & Bezboruah, T. (2015). A comparative investigation on implementation of restful versus soap-based web services. *International Journal of Database Theory and Application*, *8*(3), 297–312. https://doi.org/10.14257/IJDTA.2015.8.3.26

Calisti, M., Dignum, F., Kowalczyk, R., Leymann, F., & Unland R. (2010). Service-oriented architecture and (multi-)agent systems technology. In *Dagstuhl Seminar Proceedings 10021, 2010*. Volltexte.

Carzaniga, A., Gorla, A., & Pezze, M. (2009). Handling software faults with redundancy. In *Architecting dependable systems VI: Lecture notes in Computer Science* (5835) (pp. 148–171). https://doi.org/10.1007/978-3-642-10248-6_7

Chimmanee, S., & Jantavongso, S. (2016). The performance comparison of third generation (3G) technologies for internet services in Bangkok. *Journal of Information and Communication Technology*, *15*(1), 1–31.

Dahling, S., Razik, L., & Monti, A. (2021). Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native

computing. *Autonomous Agents and Multi-Agent Systems*, *35*(10), 1–27. https://doi.org/10.1007/s10458-020-09489-0

Dobson, G., Hall, S., & Sommerville, I. (2005). A container-based approach to fault tolerance in service-oriented architectures. In *Proceedings of the 27th International Conference of Software Engineering 2005 (ICSE '05)*. Saint Louis, USA.

Erlank, A. O., & Bridges, C. P. (2018). A hybrid real-time agent platform for fault-tolerant, embedded applications. *Autonomous Agents and Multi-Agent Systems*, *32*, 252–274. https://doi.org/10.1007/s10458-017-9378-4

Gadgil, H., Fox, G., Pallickara, S., & Pierce, M. (2007). Scalable fault-tolerant management in a service-oriented architecture. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, *HPDC 2007* (pp. 235–236). http://dx.doi.org/10.1145/1272366.1272407

Garcia, D. Z. G., & Toledo, M. B. F. D. (2007). An architecture for fault-tolerant and service-based business processes. In *Brazilian Workshop on Business Process Management, in Conjunction with IEEE 11th International Conference on Computational Science and Engineering 2007*. Gramado, Brazil. https://doi.org/10.1.1.126.4098

Hong, Y. S., No, J. H., & Han, I. (2005) Evaluation of fault-tolerant distributed web systems. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '05)* (pp. 148–151). IEEE. https://doi.org/10.1109/WORDS.2005.35

Hossain, M. S. (2006). Web service-based software implemented fault injection. *Information Technology Journal*, *05*(01), 138– 43. https://dx.doi.org/10.3923/itj.2006.138.143

Hosseini, S. M., & Arani, M. G. (2015). Fault-tolerance techniques in cloud storage: A survey. *International Journal of Database Theory and Application, 8*(4), 183–190. http://dx.doi.org/10.14257/ijdta.2015.8.4.19

Kumar, D., Jaglan, V., & Srinivasan, S. (2013). An efficient and reliable parametric approach for web service composition. *Asian Journal of Computer Science and Information Technology*, *2*(7), 226–229.

Kumar, M. (2015). Various factors affecting performances of web services. *International Journal of Sensor and Its Applications for Control Systems*, *3*(2), 1–20. http://dx.doi.org/10.14257/ijsacs.2015.3.2.02

Kumari, P., & Kaur, P. (2018). A survey of fault tolerance in cloud computing. *Journal of King Saud University – Computer and Information Sciences*, *33*(10), 1159–1176. https://doi.org/10.1016/j.jksuci.2018.09.021

Ladan, M. I. (2011). Web services metrics: A survey and a classification. In *2011 International Conference on Network and Electronics Engineering* (Vol. 11, pp. 93–98). IACSIT Press, Singapore. https://doi.org/10.1.1.1038.1043

Laranjeiro, N., & Viera, M. (2008). Deploying fault-tolerant web service compositions. *International Journal of Computer Systems Science & Engineering*, *0*, 23–34.

Lau, J., Lung, L. C., Fraga, J. S., & Santos, G. (2008). Designing fault-tolerant web services using BPEL. In *Seventh International Conference on Computer and Information Science (ICIS)* (pp. 618–623). IEEE. http://doi.org/10.1109/ICIS.2008.65

Leitao, P., Karnouskos, S., Ribeiro, L., Lee, J., Strasser, T., & Colombo, A. W. (2016). Smart agents in industrial cyber-physical systems. In *Proceedings of the IEEE* (Vol. 104, No. 5, pp. 1086–1101). https://doi.org/10.1109/JPROC.2016.2521931.2016

Li, B., Weichbrodt, N., Dehl, J., Aublin, P., Distler, T., & Kapitza. P. (2018). Troxy: transparent access to byzantine fault-tolerant systems. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* 2018 (pp. 59–70). IEEE. https://doi.org/10.1109/DSN.2018.00019

Li, C., Cheng, B., Chen, J., Gu, P., Deng, N., & Li, D. (2011). A web service performance evaluation approach based on users experience. In *2011 IEEE International Conference on Web Services*. (pp. 734–735). IEEE. https://doi.org/10.1109/ICWS.2011.29

Liu, L., Meng, Y., Zhou, B., & Wu, Q. (2006). A fault-tolerant web services architecture. In *Advanced Web and Network Technologies, and Applications* (*APWeb 2006): Lecture Notes in Computer Science* (pp. 664–671). Springer. https://doi.org/10.1007/11610496_89

Looker, N., Munro, M., & Xu, J. (2004). Testing web services. In *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)* (pp. 1–5). Oxford.

Lyu, M. R. (2011). Service reliability engineering: Performance evaluation, fault tolerance, and reliability prediction. In

International Symposium on High Confidence Software (ISHCS 2011). https://doi.org/10.1.1.231.4928

Oliha, F. O. (2018). *A fault tolerant architecture for web services solutions* (Doctoral dissertation, University of Benin, Nigeria).

Oliveira, E. M., Estrella, J. C., Kuehne, B. T., Filho, D. M. L., Adami, L. J., Nunes, L. H., Nakamura, L. H., Libardi, R. M., Souza, P. S. L., & Reiff-Marganiec, G. (2014). Design and implementation of fault tolerance techniques to improve QoS in SOA. In *10th International Conference on Network and Service Management (CNSM) and Workshop* (pp. 37–45). IEEE. https://doi.org/10.1109/CNSM.2014.7014139

Pandey, A. K., Kumar, A., & Shukla, S. (2019). A novel framework for reliable and fault-tolerant web services. *International Journal of Recent Technology and Engineering, 07*, 67–73.

Peng, K., & Huang, C. (2014). Reliability evaluation of service-oriented architecture systems considering fault-tolerance designs. *Journal of Applied Mathematics*, *2014*, 160608. http://dx.doi.org/10.1155/2014/160608

Potok, T., Phillips, L., Pollock, R., Loebi, A., & Sheldon, F. (2003). Suitability of agent-based systems for command and control in fault-tolerant, safety-critical responsive decision networks. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems 2003* (pp. 13–15), Nevada, USA.

Ramakrishnan, R., Anbarasi, J., & Kavitha, V. (2014). SoapUI and soap sonar testing tool using vulnerability detection of web service. *International Journal of Innovative Research in Computer and Communication Engineering*, *2*(11), 6995–7002.

Reddy, C. R. M., Geetha, D. E., & Kumar, T. V. S. (2017). An appraisal of web applications vs. web services with respect to performance engineering using software performance engineering approach. *International Journal of Computer Applications*, *158*(4), 20–31. https://doi.org/10.5120/ijca2017912779

Rickard, H., & Oskar, G. (2017). Evaluating performance of a fault-tolerant system that implements replication and load balancing (Bachelor's thesis, Linköping University, Sweden). Diva Portal. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1107496&dswid=-7462

Rychly, M., & Zouzelka, M. (2012). Fault injection for web services. In *Proceedings of the 14th International Conference on Enterprise Information Systems 2012* (Vol. 2, pp. 337–383). SCITEPRESS. https://doi.org/10.5220/0004153003770383

Saha, G. K. (2005). Approaches to software-based fault tolerance – A review. *Computer Science Journal of Moldova*, *13*(3), 193–231.

Saha, G. K. (2005). Transient fault tolerance in mobile agent-based computing. *INFOCOMP Journal of Computer Science*, *4(*4), 1–11.

Saha, G. K. (2009). Software based fault tolerant computing using redundancy. *International Journal of the Computer, the Internet and Management*, *17*(3), 41–46.

Satish, K. T., Madhusudhan, H. S., Syed, S. M. F. D. M., Punit, G., & Rajan, P. T. (2022). Intelligent fault-tolerant mechanism for data centers of cloud infrastructure. *Mathematical Problems in Engineering*, *2022*, 1–12. https://doi.org/10.1155/2022/2379643

Sari, A., & Akkaya, M. (2015). Fault tolerance mechanisms in distributed systems. *International Journal of Communications, Network and System Sciences*, *08*, 471–482. https://doi.org/10.4236/ijcns.2015.812042

Shafiq, M., Ding, Y., & Fensel, D. (2006). Bridging multi-agent systems and web services: Towards interoperability between Software Agents and Semantic Web Services. In *Proceedings of the 10th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2006) (*pp. 85–96), IEEE Computer. https://doi.org/10.1109/EDOC.2006.18

Umadevi, K., & Rajakumari, S. B. (2015). A review on software fault injection methods and tools. *International Journal of Innovative Research in Computer and Communication Engineering*, *3*(3), 1582–1587. https://doi.org/10.15680/IJIRCCE.2015.0303027

Vargas-Santiago, M., Pomares-Hernandez, S.E., Morales, L. A. R., & Hadj-Kacem, H. (2017). Survey on web services fault tolerance approaches based on checkpointing mechanisms. *Journal of Software*, *12*(7), 507–525. http://doi.org/10.17706/jsw.12.7.507-525

Zaide, H., Ayoubi, R., & Velazco, R. (2004). A survey on fault injection techniques. *The International Arab Journal of Information Technology*, *01*(02), 171–186.

Zhao, W. (2007, October). A lightweight fault tolerance framework for web services. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'07), Nov. 2007, Fremont, CA, USA* (pp. 542–548). https://doi.org/10.1109/WI.2007.18