A Resilient Framework for Fault Handling in Web Service Oriented Systems

Weidong Wang[1,2],Liqiang Wang[1],Wei Lu[2]
[1]*Dept. of Computer Science,University of Wyoming,Laramie, WY, USA. Email: {wwang8,lwang7}@uwyo.edu*
[2]*School of Software Engineering,Beijing Jiaotong University, Beijing, China. Email: {11112094,luwei}@bjtu.edu.cn.*

***Abstract*—Resilience is an important factor in designing web service oriented systems due to frequent failures arising in runtime. These failures derive from the stochastic and uncertainty nature of a composite web service. Service providers need to rapidly address issue when a fault occurs in system running. But it is not easy to locate and fix the faults only using the log generated by the system. In this paper, we propose a resilient framework to automatically generate a fault handling strategy for each failed service to improve the efficiency of fault handling. In the framework, we design and implement three components including exception analyzer, decision maker, and strategy selector. First, The exception analyzer builds a record, derived from the system log generated by an application, for each failed service. Next, the decision maker adopts a k-means clustering approach to construct a decision including the fault handling to each failed service in a scope. Then, the strategy selector uses an integer program solver to generate the solution to strategy selection problem that is boiled down to the optimization problem. The experiment shows that the framework can improve resilience of Web service-oriented systems under acceptable overheads, and meanwhile the accuracy of fault handling strategy is over 95% .**

***Keywords*-Web service, Resilience, Fault handling.**

I. Introduction

In Web service-oriented systems, fault is a common phenomenon, for example, services might be down, programs may return wrong results, and process might throw exceptions and so on. Once a fault occurs, a composite service provider often faces a situation that he/she takes a lot of time to analyze the system log, and then locate faults and further give a fault handling strategy. Meanwhile, it is not easy to rapidly locate the faults in a big composite service, because it has so many different atomic services. Next, once the failure is located, a decision that how to handle these failed atomic services should be made within a limited time. Moreover, it becomes impossible to deploy a fault handler for each atomic service especially in a complex workflow, because such a fault handling method will spent huge expenses, which seriously affects the system's performance. In reality, the general method is to identify the key path and pivot atomic services in this key path. For the pivot atomic services, fault handler can be manually added into the pivot services according to the types of faults and experience of programmer. However, once a fault occurs in a non-critical path, the above method becomes unuseful.

In order to solve the above problem, we adopt a two-step solution. First, we need to locate the all failed services in both non-critical paths and critical paths. Second, we explore an automatic approach to automatically generate a fault handling strategy to reduce the workload of programmer when a fault occurs. In the first step, the system log can be viewed as the data source for fault analysis. By retrieving the log, we identify the locations of faults and fault types. In the second step, we analyze some samples of fault handling strategy to the above failed services in a small scale, and then construct the fault handling strategy based on the learning results for each type of failed service.

Based on the above analysis, we call the above strategy as resilient fault handling, which is an ability to automatically make a system strong, healthy, and successful after faults happened. The resilient fault handing implementation in the Web service-oriented systems is usually designed using business process tools such as Business Process Execution Language for Web service (BPEL4WS) and WS-BPEL[1]. Furthermore, the resilient fault handling should be efficiently achieved in the manner of automation. The achieved capability of fault resilience is important and essential for a variety of critical services (e.g. E-commerce), which are attracting attentions.

In this paper, we combine the two-step solution and propose a framework for resilient fault handling. In the framework, three components are designed and implemented including exception analyzer, decision maker, and strategy selector. First, The exception analyzer builds a record for each failed service by tracking system log. Next, the decision maker adopts a k-means clustering approach to generate a decision including the fault handling to each failed service in a scope. Then, the strategy selector uses the integer program algorithm to generate a solution for strategy selection optimization problem.

A. Contributions

The contributions of this paper can be categorized as follows.

–Compared with traditional approaches, the proposed framework achieves higher fault recognition rate and less messages. Long duration and asynchronous communication make tracing fault more different. In the past, a runtime fault detector based on the method of message

[1]http://docs.oasis-open.org/wsbpel/2.0/

listening [1] generates lots of messages during its execution in such an environment. In this paper, we introduce an approach by analyzing system log to acquire the location of failed path and services to improve the fault recognition rate under less messages.

–We adopt the similarity-based clustering approach to automatically classify these failed services. Then, we design a decision algorithm that can make a decision to generate fault handling strategies for a set of failed services.

The rest of this paper is organized as follows. Section II provides the design of a resilience approach. Section III gives a framework details about implementation. Section IV presents the experiment results. Section V provides an overview of the related work. Section VI concludes this paper and outlines the future work.

II. Resilience Strategy

In this section, we explain the resilience methodology in service oriented systems.

A. Fault Model

The types of malfunctions to service oriented systems includes errors, faults, failures, and exceptions. An error, as part of system state, indicates a serious system problem, such as machine crash or system shutdown. A fault is a cause of an error, which can be repaired by some strategies including manual operations and automatic repaired methods. A failure is considered as the result of an error or fault. An exception is an event occurring during the execution of a program, and disrupts the normal flow of the program's instructions[2]. The classification about failures is shown in Table I.

Class 1. Network and system errors or faults. These errors or exceptions are raised by system hardware faults, middleware platform faults, or communication faults.

Table I
Failures in Web service oriented systems.

Error or fault	No.	Type of error or fault
Class 1. Network and system errors or faults	1	Network connection failure or breakoff
	2	Servers shut down
	3	Reply or receive packets drop
	4	Partner computer unavailable
	5	System runtime error
Class 2. Application faults	6	Process logic faults
	7	Web service failure during transaction
	8	User-defined faults
Class 3. Interface matching and Web service binding faults	9	Exception in Web service name or port name
	10	Input or output parameter mismatching including name, type, and number
	11	Web service style mismatching
	12	Target name space fault
Class 4. Contract violation faults	13	QoS dissatisfaction
	14	SLA violation
	15	User-defined rules violation

[2]http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Class 2. Application faults. These faults can be classified into three types. The first type of fault is raised by Web service itself during system running time. There are so many causes to raise such kind of faults. The fault mainly derives from the design of a Web service itself, and is associated to different running environments. The second kind of exception is process logic fault. The cause could be that the designer may design the business process incorrectly. The third kind of fault is user-defined fault. To increase the system's robustness, fault handling mechanisms are added into the program.

Class 3. Interface matching and Web service binding faults. These faults often occur at Web service binding time. Interface mismatching results in Web service binding failure, which is often caused by mismatching of the name, type, and number of a Web service.

Class 4. Contract violation faults. These faults can be classified into three types, QoS dissatisfaction,SLA violations, and violations of rules defined by users.

B. Resilience Strategy

According to the characteristics of faults mentioned in Section II-A, we classify resilience mechanisms into six categories [2].

–Abort. In order to ensure the completion of a process, we have to give up the failed service, which can be used in deadlock or other runtime fatal failures.

–Hang. The service fails to complete execution due to unsatisfied execution conditions. If the waiting time exceeds the plan's requirement, the main process automatically gives up this service.

–Notify. The main process invokes a service, but fails to complete execution due to out-of-service problems such as data access failure or network connection failure. In order to keep the execution going on, the main process should give up this service and write logs or notice users.

–Retry. This strategy repeats the execution of a service upon its successful completion. Specifically, we may set the retry times as a special condition. This handler can be used in a subprocess, which requires more data exchange, especially when a network environment is relatively poor.

–Skip. This strategy means that the process omits the execution of optional services. This strategy focuses on uncritical services, which can be skipped. SLA faults could be usually handled by the skip strategy.

–Substitute. The main process invokes a service, but fails to complete execution due to some failures, where the invoked service is necessary for the completion in a composite service. In order to complete the process, the main process provides more service candidates to ensure the success of this process.

III. Design and Implementation

In this section, we design and implement a framework with a resilient fault handling using the mechanisms mentioned in Section II-B. The framework consists of three components, *i.e.,* exception analyzer, decision maker, and strategy selector, as shown in Figure 1.

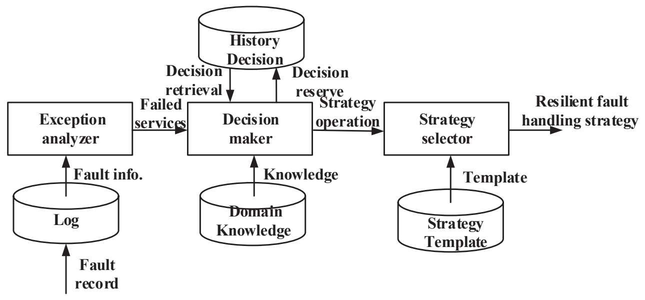
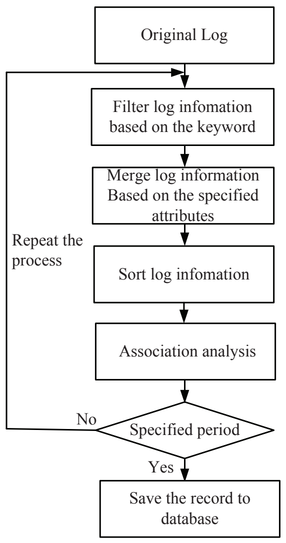


Figure 1. Framework of a resilient fault handling.

A. Exception Analyzer

The exception analyzer builds records for all failed service. The records come from the system log generated by an application. The flowchart is shown in Figure 2. First, the original log is filtered to ignore irrelevant information using the BM [3] algorithm based on the keywords (*e.g.,* time and type of faults). Next, we merge the results with the same time, type, and location. Then, we re-sort the merged log using the quick-sort algorithm[3] based on the main attribute (*e.g.,* time). Finally, we group faults from the same scope but different services together by association analysis. If the faults occur within the given period, it will be saved into the database.



Original Log
Filter log infomation based on the keyword
Merge log information Based on the specified attributes
Repeat the process
Sort log infomation
Association analysis
No
Specified period
Yes
Save the record to database

Figure 2. Flowchart of exception analyzer.

An example is shown in Table II, the exception in the first row indicates the current failed service including the fault type and the time of fault occurrence.

[3]http://en.wikipedia.org/wiki/Quicksort

B. Decision Maker

As the core of three components, the decision maker decides what fault handling strategy to use. According to the fault location and ID of failed services provided by the exception analyzer, the decision maker firstly looks for the matching fault handling strategy by retrieving the history decision database. If a decision is available, it will be provided to the strategy selector. Otherwise, the decision maker will combine domain knowledge and use a decision algorithm to generate a new fault handling decision, which is sent to the strategy selector and meanwhile is also saved into the history decision database.

In order to match fault handling strategy for each type of fault, we divide fault records into two parts: training sample and test data. For training sample, we record every fault handling strategy for each different type of fault. Let $P_{type}^{strategy}$ be the occurrence frequency for a type of the strategy appearing in the same type of fault. For instance, P_{SLAV}^{skip} means the occurrence frequency for the *skip* handling in SLAV (SLA violation) fault. Let $U_{type}^{strategy}$ be the probability of usability with an fault handling strategy (*e.g.,* NCF, Network connection failure) upon an fault handling strategy (*e.g.,* retry). Let T_{type} be the occurrence number of a type of fault, for instance, T_{NCF} indicates that the number of network connection failures. The equation is shown as follows.

$$U_{type}^{stategy} = \frac{P_{type}^{strategy}}{T_{type}} \tag{1}$$

According to Equation 1, we compute the probability for each handling strategy upon a type of fault. As Table III shows, the column denotes the types of faults, the row represents the handling strategy for each type of fault. In order to find rules related to the usability of each fault handling strategy, we use training sample derived from system log. Then, using Equation 1, we obtain the value for each item in Table III, which indicates the probability for the corresponding fault handling strategy applied to the same type of fault. Based on these results, we know what handling strategy can be used. In reality, we try each probable strategy and then choose the most appropriate one. However, there are three handling strategies at least for each type of fault. We need to know which one is the best solution for the given failed service. In order to decrease the complexity of this processing, we design a metric to measure similarity of type of fault among fault handling strategies. Here, Euclidean Distance-based Similarity Algorithm [4], [5] shown in Equations 2 and 3 is adopted by our approach.

$$Distance(E_{\alpha j}, E_{\beta j}) = \sqrt{\sum\nolimits_{j=1}^{N} (E_\alpha - E_\beta)^2},\ \forall \alpha \neq \beta,\ 1 \leq j \leq N \tag{2}$$

$$Similarity(E_{\alpha j}, E_{\beta j}) = \frac{1}{1 + Distance(E_{\alpha j}, E_{\beta j})}, \forall \alpha \neq \beta,\ 1 \leq j \leq N \tag{3}$$

Table II
SYSTEM LOG RELEVANT TO FAULTS IN DATABASE

(Note: fault type is denoted by ET; the failed service is denoted by FS; the time of the fault thrown is denoted by TT; the number of services in process sequence is denoted by SS; associated services in the same scope numbers are denoted by SN; the severity of the fault is denoted by SE; and the scope that the fault belongs to is denoted by SC.)

ID	ET	FS	TT	SS	SN	SE	SC
20001	Style mismatching	Hotel reservation	05-09-2014, 17:04:30	1	C101	Medium	SC03
20002	SLA exception	Searching tourist	05-09-2014, 18:01:42	4	C101:104	Low	SC05
20003	Network failure	Car reservation	05-09-2014, 19:03:50	1	C102	High	SC11

According to the k-means cluster algorithm [6], [7], we obtain four clusters ($k = 4$) as follows. The first cluster includes SRE, PLF, SSD, and WSF, the second cluster includes PCU, EWP, IOPM, WSSM, and TSF, the third cluster includes UDE, QOSD, SLAV, and UDRV, and the fourth cluster includes NCF and RPD (The above abbreviative notations are defined in Table III). Function $Distance(E_{\alpha j}, E_{\beta j})$ indicates the distance computation between the fault handling strategy α and fault handling strategy β at the same fault j. $Similarity(E_{\alpha j}, E_{\beta j})$ denotes the similarity between the fault handling strategy α and fault handling strategy β at the same fault j. N denotes the number of types of faults.

Let R_{skip} be the strategy for the first cluster using skip handling, R_{retry} be the strategy for second cluster using retry handling, $R_{substitute}$ be the strategy for the third cluster using substitute handling, and $R_{rollback}$ be the strategy for the fourth cluster using rollback handling.

When handling a failed service, we should consider all executed services within its scope. Here, we assume that all services can be compensated in a scope, and there is no conflicts between the compensating activities. Algorithm 1 shows how to choose a concrete fault handling strategy to deal with the fault. Let d_{ij} be the decision probability of the j^{th} fault handling strategy for the i^{th} service.

C. Strategy selection

A fault handling decision has a number of variations based on different strategies for the failed services in a scope. For example, if there are n failed service to be handled, and each one may generate m fault handling strategies, where n is the maximal number of failed services and m is maximal number of fault handling strategies. Here, we consider three user constraints: execution time, response time, and cost. The selection problem of fault handling strategy with user constraints can be formulated mathematically as

Problem 1. **Minimize:** $\sum_{i=1}^{n}\sum_{j=1}^{m} d_{ij} \times x_{ij}$

Subject to:

- $\sum_{i=1}^{n}\sum_{j=1}^{m} e_{ij} \times x_{ij} \leq u_1$
- $\sum_{i=1}^{n}\sum_{j=1}^{m} r_{ij} \times x_{ij} \leq u_2$
- $\sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij} \times x_{ij} \leq u_3$

Algorithm 1 Decision algorithm.

```
Input:
    Failed path and service,"$P_{ij}$" and"$S_{ij}$"
Output:
    Decision set for the given faults in a scope,"$d_{ij}$".
 1: For $(i = 0; i \leq n; i++)$ do {
 2: j=0;
 3: if $P_{ij}$ and $S_{ij}$ happened before then
 4:     Retrieve the previous decision for $d_{ij}$
 5: else if $S_{ij}.FaultType \in R_{skip}$ then
 6:     // Skip the failed service.
 7:     $d_{ij} \leftarrow Skip(S_{ij})$;
 8: else if $S_{ij}.FaultType \in R_{retry}$ then
 9:     // Retry the service that was failing.
10:     $d_{ij} \leftarrow Retry(S_{ij})$;
11:     j=j+1;
12:     if $P_{ij}$ is not in key path then
13:         Skip the failed service.
14:         $d_{ij} \leftarrow Abort(S_{ij})$;
15:     end if
16: else if $S_{ij}.FaultType \in R_{substitute}$ then
17:     // Find a service that substitutes a failed one.
18:     $d_{ij} \leftarrow Substitute(S_{new}, S_{ij})$;
19:     j=j+1;
20:     if $P_{ij}$ is not in key path then
21:         $d_{ij} \leftarrow Reminder(S_{ij})$;
22:     end if
23: else if $S_{ij}.FaultType \in R_{rollback}$; then
24:     $d_{ij} \leftarrow Rollback(Scope, S_{ij})$;
25: else if $S_{ij}.FaultType \in Others$ then
26:     $d_{ij} \leftarrow Reminder(S_{ij})$;
27: end if
28: } EndFor
29: return $(d_{ij})$;
```

- $\forall i, \sum_{j=1}^{m} x_{ij} = 1$
- $x_{ij} \in \{0, 1\}$

In Problem 1, x_{ij} is set to 1 if the j^{th} candidate strategy for the i^{th} failed service is selected and 0 otherwise. Furthermore, d_{ij}, e_{ij}, r_{ij}, and c_{ij} are the decision probability, execution time, response time, and cost of the strategy candidates, respectively. n is the number of failed services in the same scope, and m is the number of fault handling resilient strategy candidates for a failed service. u_1, u_2, and u_3 are the user constraints for execution time, response time, and cost, respectively. Problem 1 can be extended by adding more constraints in the future.

By solving Problem 1 using the open source Integer

Table III
THE USABILITY OF EACH FAULT HANDLING STRATEGY FOR EACH TYPE OF FAULT IN WEB SERVICE ORIENTED SYSTEMS.

(Note: Network connection failure is denoted by NCF; servers shut down is denoted by SSD; packets drop is denoted by RPD; partner computer unavailable is denoted by PCU; system runtime error is denoted by SRE; process logic faults is denoted by PLF; Web service failure during transaction is denoted by WSF; user defined faults is denoted by UDE; exception in Web service name or port name is denoted by EWP; input or output parameter mismatching including name, type, and number is denoted by IOPM; Web service style mismatching is denoted by WSSM; target name space fault is denoted by TSF; QoS dissatisfaction is denoted by QOSD; SLA violation is denoted by SLAV; user defined rules violation are denoted by UDRV.)

No.	Name	Null	Abort	Hang	Reminder	Retry	Skip	Substitute	Rollback
1	NCF	0.08	0.10	0.25	0.22	0.35	0	0	0
2	SSD	0.06	0	0	0.14	0.20	0	0	0.60
3	RPD	0.05	0.06	0.15	0.05	0.69	0	0	0
4	PCU	0.01	0.06	0.29	0.04	0.15	0.10	0.35	0
5	SRE	0.01	0.03	0	0.06	0.35	0	0	0.55
6	PLF	0.01	0.04	0	0.15	0.27	0	0	0.53
7	WSF	0	0	0.15	0.15	0.37	0.03	0	0.30
8	UDE	0.02	0.18	0	0.25	0	0.55	0	0
9	EWP	0	0.10	0	0.30	0.05	0	0.55	0
10	IOPM	0	0.11	0	0.32	0.04	0	0.53	0
11	WSSM	0	0.13	0	0.28	0.06	0	0.53	0
12	TSF	0	0.16	0	0.22	0.10	0	0.52	0
13	QOSD	0.05	0.05	0	0.26	0	0.64	0	0
14	SLAV	0.03	0.06	0	0.21	0	0.66	0	0
15	UDRV	0.02	0.03	0	0.28	0	0.67	0	0

Program solver [4], a set of strategies are generated for failed services in a scope. Finally, according to the strategy set, the strategy selector automatically generates a resilient fault handling for each failed service in the scope by retrieving strategy template database.

IV. EVALUATION OF EXPERIMENTS

All components in the framework are implemented in Java 6. Apache Tomcat 6 is used as Web server, and Apache Axis [5] is used as a Web service container, which can generate and deploy Web service applications.

A. *Effectiveness Evaluation*

In our experiment, test data derived from system log including 15 different types of faults are divided into 3 groups as 3 cases. Then, we record the success numbers of fault handling strategies.

In Figure 3, let x-axis be type of fault, y-axis be fault handling strategy, and z-axis be success numbers of fault handling strategies. This figure shows that the system performs success times under the each group.

In the same scenario, in order to show the success rate for each type of fault, we calculate the success rate and draw Figure 4. In Figure 4, x-axis denotes type of fault and y-axis indicates the success rate of fault handling strategy. The average success rate is 97.07%. In the above, all the results are from random test data, which further shows that our component is very effective and usable. We also test whether the solution is correct. The test validates whether the output values after executing the fault handling strategy is the same as manually selecting fault handling strategy. Then, we compare the output values obtained from executing

[4]http://lpsolve.sourceforge.net/5.5/Java/README.html
[5]http://http://axis.apache.org/axis/

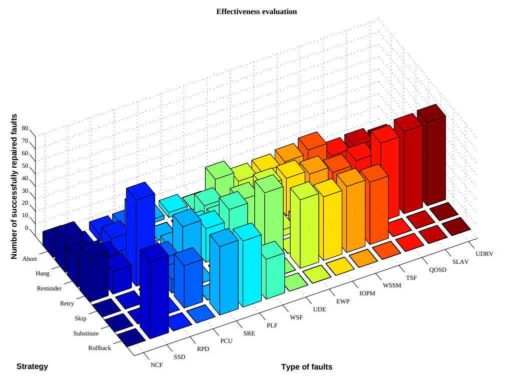


Figure 3. Effectiveness evaluation under different types of faults.

the strategy with the output values obtained from original executing process. The strategy is valid if these results are equal. Consequently, the test result proves that our approach is valid. More importantly, the fault handling strategy produced by our framework can provide a new evidence to further improve the performance of Web service systems.

B. *Performance Study*

We conduct experiment to evaluate execution time and response time. First, we evaluate the performance in two scenarios: with resilient fault handling strategy and without any strategy. We deploy the framework on the computer where a composite service runs on, and then execute the service without any fault handling in another computer. Finally, we test the response time and the execution time related to the composite service, and compare them in different scenarios with resilient fault handling strategy and without any strategy. Here, all computers have the same

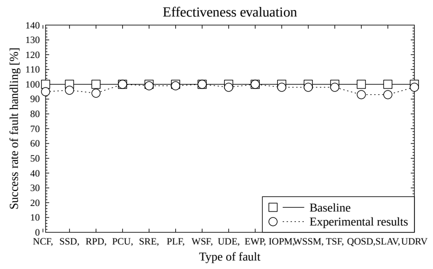
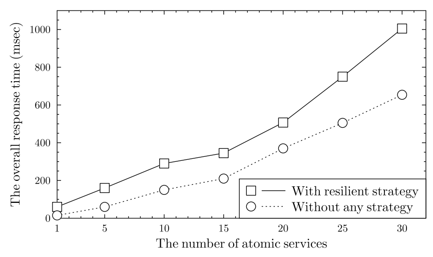
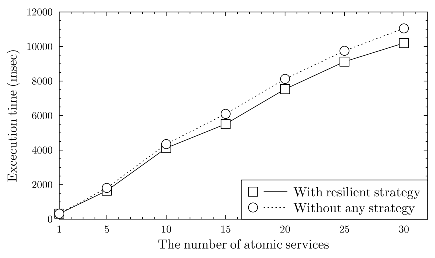


Figure 4. Success rate under different types of faults.



(a) Response time.



(b) Execution time.

Figure 5. Performance with and without resilient fault handling strategy.

configuration: HP 8280 with Intel Core i5-2400 with four CPU at the clock speed of 3.1GHz and 8GB RAM. All the computers are connected by Ethernet (100 Mbps).

In the experiment, there are 3 cases, each of which has certain number of atomic services (e.g., the number of services is 5, 10, 15, 20, 25, or 30) consisting of a composite service. In each case, we consider two scenarios: the case with resilient fault handling strategy, and the case without any strategy.

1) Response time indicates the time that a system takes to react to a given fault handling strategy.
2) Execution time indicates the time required to complete all failed services in a scope.

In Figure 5(a), we can see that both response time with and without our strategy are obviously increasing, while the number of atomic services in a composite service increases. The reason is that the invocation of each service in a composite service needs some time, which depends on the number of services and the invocation time for each service. In addition, the response time with the strategy is slightly longer than the response time without any strategy. For example, when the number of services is 20, the response time with the strategy is 507 (**msec**), and the response time without any strategy is 210 (**msec**). From this result, we can conclude that additional time 297 (**msec**) is used to generate a resilient fault handling strategy as the additional overhead.

In Figure 5(b), both execution time with and without resilient fault handling strategy are obviously increasing while the number of atomic services in a composite service increases. In other words, the execution time is increasing as the number of atomic services increases. In each case, for example, when the number of services is 15, the execution time with resilient fault handling strategy is 6100 (**msec**), and the execution time without any strategy is 5562 (**msec**). The additional overhead is 538 (**msec**), which means that the system needs 538 (**msec**) to execute a fault handling resilient strategy, including the interface operation, reading and writing the information from database and so on. Although this process increases execution time, the additional overhead is usually not a big burden for a composite service. On the other side, the resilience of a composite service has been improved. The reason is that if the system suffers from fault handling strategy or without any strategy, we can only manually deal with the fault. This process will generate more overhead than our approach.

In the above, we analyze the performance in terms of response time and execution time, and the overheads is less than 10% in total execution time and is acceptable to users.

C. Performance Comparison

To study the resilience improvement, we compare four methods as follows:

1) *NoRT*. No resilient fault handling strategy is added in the components.
2) *RandomRT*. The resilient strategy is employed based on random selection.
3) *UserRT*. The resilient strategy is employed based on specified strategy according to users' subjective experiences.
4) *SLRT*. The resilient strategy is employed based on the system log.

In our experiments, a group of atomic services are contained for each composite service (*e.g.,* 10, 50, 100 atomic services). The above four methods (*i.e., NoRT, RandomRT, UserRT*, and *SLRT*) are conducted on these different volume of service numbers, and the results are reported in Table IV.

In Table IV, Service FP represents the failure of probability of the service. The details of experimental results are shown below.

Table IV
EVALUATION OF RESILIENCE IMPROVEMENT

Services Numbers	Method	Service $FP = 0.5\%$			Service $FP = 1\%$			Service $FP = 3\%$		
		Case 1	Case 2	Case 3	Case 1	Case 2	Case 3	Case 1	Case 2	Case 3
10	NoRT	0.912	0.912	0.913	0.710	0.710	0.711	0.350	0.350	0.350
	RandomRT	0.921	0.923	0.924	0.710	0.711	0.713	0.357	0.355	0.358
	UserRT	0.922	0.934	0.945	0.739	0.745	0.746	0.611	0.611	0.624
	SLRT	1.000	1.000	1.000	0.993	0.992	0.995	0.987	0.988	0.985
50	NoRT	0.532	0.531	0.531	0.321	0.321	0.321	0.068	0.066	0.068
	RandomRT	0.531	0.532	0.534	0.330	0.334	0.333	0.069	0.067	0.069
	UserRT	0.710	0.742	0.781	0.670	0.678	0.674	0.520	0.524	0.524
	SLRT	0.991	0.991	0.990	0.980	0.984	0.983	0.967	0.965	0.965
100	NoRT	0.211	0.210	0.210	0.101	0.102	0.102	0.014	0.014	0.016
	RandomRT	0.212	0.214	0.218	0.102	0.103	0.101	0.015	0.015	0.016
	UserRT	0.762	0.767	0.811	0.612	0.612	0.613	0.459	0.459	0.459
	SLRT	0.989	0.989	0.989	0.976	0.976	0.976	0.956	0.950	0.955

1) In the above four testing methods, *SLRT* gives the best success probability performance while the *NoRT* provides the worst success probability performance, because the *SLRT* method uses the components based on the analysis of system log, and the *NoRT* method provides no resilience strategy.
2) Compared with the *NoRT*, the *RandomRT* obtains little performance improvement in success probability.
3) Compared with the *NoRT* and *RandomRT*, the *UserRT* method can obtain a certain success probability, because the approach uses users' experiences as the basis of strategy selection. But *UserRT* is more complicated and needs certain field knowledge, which is subjective.
4) Compared with the *NoRT*, *RandomRT*, and *UserRT*, *SLRT* obtains the better resilience than the other methods, because the *SLRT* is based on the analysis of system log.
5) When the number of activities increases from 10 to 100, the success probability of the *UserRT* slightly decreases, while success probabilities of *RandomRT* and *NoRT* have no change or decrease. The results indicate that fault handling performance can be improved by our resilient strategy.
6) While the service failure probability increases from 0.5% to 3%, the success probabilities of all methods except *SLRT* dramatically decrease. This indicates the *SLRT* has better resilience than other methods.

In summary, our resilient fault handling strategy has been proved to be valid and accurate.

V. RELATED WORK

In the area of Web service fault handling, Liu et al. [2] propose FACTS, a framework for fault-tolerant composition of transactional Web services. Meanwhile, fault handling resilient mechanisms including exception detection are designed and implemented. However, the resilient mechanism more or less depends on manually operation but not automatic operation. Guan et al. [1] present a policy driven based framework (EHF-S) for web service exception handling to simplify the exception handling process. But the framework lacks software performance verification, and software test work is more or less missing in the paper. Modafferi et al. [8] propose self-healing plug-in for a WS-BPEL engine, which utilizes a BPEL4WS compatible engine that has the ability of executing processes defined using BPEL language, to provide process-based recovery actions. In the area of self-healing software systems [9], David et al. [10] propose architecture models for problem diagnosis and repair for self-healing systems. This architecture can handle specified simple failed services and less consider coupled complex services. Shin [11] proposes an approach to design a self-healing component for robust, concurrent and distributed software systems. The plans for reconfiguration and repair are considered as black boxes. Gerhard friedrich et al. [12] propose a self-healing approach to cure exceptions and a model-based approach to repair the faults in service-based processes. Moreover, a platform is introduced for fault diagnosis. A prototype is developed to validate the proposed repair approach for Web service composition. However, the heuristic-based repair ability does not give concrete description at design-time. Especially for a complex Web service-based process, a reasoning model needs to be further validated.

There are also several traditional resilience strategies. Moorsel et al. [13] note that retries or restarts are considered as a phenomenon in computing system in software rejuvenation, preventive maintenance, or when a suspicious failure is spotted. Okamura et al. [14] consider that retries or restarts are typical recover strategies to satisfy a deadline in real-time systems as a significant environmental diversity technique in service computing. The approaches in [15], [16] show that transaction mechanism has been widely used in Web service-oriented system. Li et al. [16] propose two kinds of transaction types, coordination mechanisms, and a transaction processing coordination model based on BPEL. However, these fault handling strategies focus on some types of faults, but fail to handle a batch of different types of failed services.

VI. Conclusion

It is important to locate faults rapidly and give a fault handling solution when faults occur in Web service system. In this paper, we propose a resilient framework to automatically generate a fault handling strategy for each failed service. In the framework, we design and implement three components including exception analyzer, decision maker, and strategy selector. The exception analyzer generates a record for each failed service. The decision maker constructs a decision according to these faults. Based on the decision, the strategy selector gives a fault handling strategy for each failed service. The experiment shows that the framework improves the system's resilience under acceptable overheads with high accuracy. In the future, we will further extend the framework by considering more constraints in the process of selecting strategy. Meanwhile, more efficient selection algorithms will be investigated. In addition, we will adapt the framework to generate fault handling strategies for cloud services by combining our works [17], [18], [19], [20] in the fields of service computing and cloud computing.

Acknowledgment

This work is supported in part by NSF under Grant 1118059 and NSFC under Grant 61272353, 61428201.

References

[1] H. Guan, S. Ying, and C. Jiang, "An exception handling framework for web service," in *Proceedings of International Conference on Computer Engineering and Network*. Springer, 2013, pp. 1173–1180.

[2] A. Liu, Q. Li, L. Huang, and M. Xiao, "FACTS : A Framework for Fault-Tolerant Composition of Transactional Web Services," *IEEE Transactions on Service Computing*, vol. 3, no. 1, pp. 46–59, 2010.

[3] R. Boyer and J. Moore, "A fast string searching algorithm," *Communication of ACM*, vol. 20, no. 1, pp. 762–772, 1977.

[4] S. Muchun and C. Chienhsing, "A modified version of the K-means algorithm with a distance based on cluster symmetry," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 6, pp. 674–680, 2001.

[5] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. ACM, 2002, pp. 380–388.

[6] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, "An efficient k-means clustering algorithm: analysis and implementation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[7] P. La Rosa, A. Nehorai, H. Eswaran, C. Lowery, and H. Preissl, "Detection of Uterine MMG Contractions Using a Multiple Change Point Estimator and the K-Means Cluster Algorithm," *IEEE Transactions on Biomedical Engineering*, vol. 55, no. 2, pp. 453–467, 2008.

[8] S. Modafferi, E. Mussi, and B. Pernici, "Sh-bpel: A self-healing plug-in for ws-bpel engines," in *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*. ACM Press, 2006, pp. 48–53.

[9] R. Hamadi, B. Benatallah, and B. Medjahed, "Self-adapting Recovery Nets for Policy-driven Exception Handling in Business Processes," *Distrib. Parallel Databases*, vol. 23, no. 1, pp. 1–44, 2008.

[10] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the First Workshop on Self-healing Systems*. ACM Press, 2002, pp. 27–32.

[11] M. E. Shin, "Self-healing components in robust software architecture for concurrent and distributed systems," *Science of Computer Programming*, vol. 57, no. 1, pp. 27–44, 2005.

[12] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni, "Exception Handling for Repair in Service-Based Processes," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 198–215, 2010.

[13] A. van Moorsel and K. Wolter, "Analysis of Restart Mechanisms in Software Systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 547–558, 2006.

[14] H. Okamura, T. Dohi, and K. S. Trivedi, "On-Line Adaptive Algorithms in Autonomic," in *IEEE-EURASIP Workshop*, 2010, pp. 32–46.

[15] S. Chang-ai, E. El-Khoury, and M. Aiello, "Transaction Management in Service-Oriented Systems: Requirements and a Proposal," *IEEE Transactions on Services Computing*, vol. 4, no. 2, pp. 167–180, 2011.

[16] L. Wenjuan, P. Shanliang, and W. Yabei, "Research of web service transaction extending model based on ws-bpel," in *2010 2nd Int'l Conf.on Information Engineering and Computer Science (ICIECS)*, 2010, pp. 1–6.

[17] W. Wang, W. Lu, L. Wang, W. Xing, and Z. Li, "A ranking-based approach for service composition with multiple qos constraints," in *In International Conference on Information Technology and Software Engineering*. Springer, 2013, pp. 185–195.

[18] H. Huang and L. Wang, "Pp: A combined push-pull model for resource monitoring in cloud computing environment," in *In 3rd International Conference on Cloud Computing*. IEEE, 2010, pp. 260 – 267.

[19] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang, "Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances," in *In the IEEE 6th International Conference on Cloud Computing*. IEEE, 2013, pp. 228–235.

[20] L. Wang, S. Lu, and X. Fei, "Atomicity and provenance support for pipelined scientific workflows," *In Journal of Future Generation Computer Systems*, vol. 25, no. 5, pp. 568–576, 2009.