Selecting an Optimal Fault Tolerance Strategy for Reliable Service-Oriented Systems with Local and Global Constraints

Zibin Zheng, *Member*, *IEEE*, and Michael R. Lyu, *Fellow*, *IEEE*

Abstract—Functionally equivalent web services can be composed to form more reliable service-oriented systems. However, the choice of fault tolerance strategy can have a significant effect on the quality-of-service (QoS) of the resulting service-oriented systems. In this paper, we investigate the problem of selecting an optimal fault tolerance strategy for building reliable service-oriented systems. We formulate the user requirements as local and global constraints and model the selection of fault tolerance strategy as an optimization problem. A heuristic algorithm is proposed to efficiently solve the optimization problem. Fault tolerance strategy selection for semantically related tasks is also investigated in this paper. Large-scale real-world experiments are conducted to illustrate the benefits of the proposed approach. The experimental results show that our problem modeling approach and the proposed selection algorithm make it feasible to manage the fault tolerance of complex service-oriented systems both efficiently and effectively.

Index Terms—Fault tolerance, web service, service composition, quality-of-service (QoS)

1 Introduction

Web services are self-contained applications that can be described, published, and invoked over the Internet. In the service-oriented environment, complex distributed systems can be dynamically composed by discovering and integrating Web services provided by different organizations. As service-oriented architecture (SOA) is becoming a large part of IT infrastructures, building reliable service-oriented systems is more and more important. However, comparing with the traditional stand-alone software systems, building reliable service-oriented systems is much more challenging, because: (1) Web services are usually distributed across the unpredictable Internet; (2) remote Web services are developed and hosted by other providers without internal design and implementation details; (3) performance of Web services may change dynamically (e.g., caused by workload change of servers, internal updates of Web services, performance fluctuation of communication links, etc.); and (4) remote Web services may even become unavailable without any advance notifications.

In software reliability engineering [19], there are four main approaches to increase system reliability, which are fault prevention, fault removal [11], fault tolerance, and fault forecasting [12]. Since source-codes and internal designs of Web services are unavailable to service users (usually developers of the SOA systems), it is difficult to use fault prevention and fault removal techniques to build fault-free service-oriented systems. Another approach for building reliable systems, software fault tolerance [18], makes the system more robust by masking faults instead of removing faults. One approach of software fault tolerance, also known as *design diversity*, is to employ functionally equivalent yet independently designed components to tolerate faults [18]. Due to the cost of developing redundant components, design diversity is usually only employed for critical systems. In the area of service computing [34], however, it is possible to construct a fault-tolerant service-oriented system without having to pay the cost of developing diverse components. There are a number of functionally equivalent Web services already diversely implemented by different organizations on the Internet. These Web services can be employed as alternative components for building diversity-based fault-tolerant service-oriented systems.

Fault tolerance strategies can be divided into passive replication strategies and active replication strategies. Passive strategies [9], [28], [30] employ a primary service to process the request and invoke another alternative backup service when the primary service fails, while Active strategies [16], [21], [24], [27], [29] invoke all functionally equivalent services in parallel. Complementary to previous approaches which mainly focus on applying various fault tolerance strategies for service-oriented systems, this paper investigates how to select the optimal fault tolerance strategy for building reliable service-oriented systems, considering not only objective Quality-of-Service (QoS) performance of Web services, but also subjective requirements of service users.

In this paper, user requirements are formulated as local constraints and global constraints. A service-oriented system typically includes a set of tasks. Suitable Web services need to be selected to fulfill these tasks. Service users may provide constraints for a single task (named as *local constraints*), such

• *The authors are with the Shenzhen Key Laboratory of Rich Media Big Data Analytics and Applications, Shenzhen Research Institute, The Chinese University of Hong Kong, and with Ministry of Education Key Laboratory of High Confidence Software Technologies (CUHK Sub-Lab), The Chinese University of Hong Kong. E-mail: {zbzheng, lyu}@cse.cuhk.edu.hk.*

Manuscript received 05 June 2010; revised 10 Jan. 2013; accepted 04 Sep. 2013. Date of publication 16 Sep. 2013; date of current version 12 Dec. 2014. Recommended for acceptance by D. Avresky.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.189

as *response-time of task 1 should be less than 1 second*. Service users can also provide constraints for the whole service-oriented system (named as *global constraints*), such as *availability of the service-oriented system should be higher than 99%*. The research problem of this paper is how to identify the optimal fault tolerance strategy for a service-oriented system under these local and global constraints.

To address this research problem, based on our previous work [36], [37], this paper proposes a systematic and extensible framework. The main features of this framework are: (1) an extensible QoS model of Web services, (2) a number of fault tolerance strategies, (3) a QoS composition model of Web services, (4) a consistency checking algorithm for complex service-oriented systems, and (5) various QoS-aware algorithms for selecting the optimal fault tolerance strategy.

In our framework, we model the problem of selecting an optimal fault tolerance strategy as a 0-1 Integer programming (IP) problem. A heuristic algorithm is proposed to efficiently solve the problem. We select the optimal fault tolerance strategy not only for a single task, but also for semantically-related tasks where multiple tasks have strong correlation (e.g., contain state dependency) and must be performed by the same type of Web services. In contrast to previous research on fault-tolerant Web services [9], [16], [21], [24], [27]–[30], which typically consider only one single metric (i.e., reliability), our framework considers not only reliability, but also a number of other QoS properties (e.g., response-time, cost, etc.) and user requirements. Comprehensive experiments are conducted based on our WS-DREAM (Distributed REliability Assessment Mechanism for Web Services) architecture [35]. The experimental results show the effectiveness and efficiency of our proposed optimization algorithm. Moreover, a real-world Web service QoS dataset is released for future research.[1]

This paper advances the current state-of-the-art in software fault tolerance for Web services by proposing a systematic and extensible framework for selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. The main contributions of this paper include: (1) modeling the problem of selecting the optimal fault tolerance strategy as a specific optimization problem and designing a heuristic algorithm to efficiently solve the problem; (2) specifying the user requirements as local and global constraints, and formulating the selection of the optimal fault tolerance strategy for semantically-related tasks as a constraint in the optimization problem; and (3) proposing an extensible framework which integrates different modules together for selecting an optimal fault tolerance strategy.

The rest of this paper is organized as follows: Section 2 introduces a motivating example. Section 3 presents preliminaries. Section 4 investigates optimal fault tolerance strategy selection. Section 5 describes experimental design and results. Section 6 reviews related work and Section 7 concludes the paper.

2 Motivating Example

We begin by a motivating example to show the research problems. In this paper, a *service plan* is an abstract description of activities for a business process, which includes a set of tasks executing according to a certain workflow. Fig. 1 shows a simple service plan including six tasks. Each task can be executed by invoking a Web service. Following the same assumption of work [2], [4], [32], we assume that for each task in a service plan, there are multiple functionally equivalent Web service candidates that can be adopted to fulfill the task. These functionally equivalent Web services can be obtained from *service communities* [4], [33], which define common terminologies to guarantee that Web services developed by different organizations have the same application programming interface.

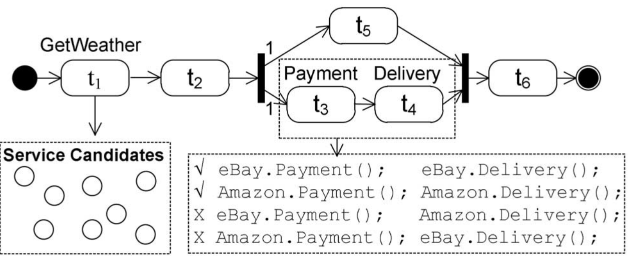

GetWeather
t_1
t_2
t_5
t_3
t_4
t_6
1
1
Payment Delivery
Service Candidates
√ eBay.Payment(); eBay.Delivery();
√ Amazon.Payment(); Amazon.Delivery();
X eBay.Payment(); Amazon.Delivery();
X Amazon.Payment(); eBay.Delivery();

Fig. 1. A motivating example.

For the example shown in Fig. 1, there are several challenges to be addressed: (1) There are a number of Web service candidates for the task t_1 (*GetWeather*). Which candidate would be optimal? Does task t_1 requires fault tolerance strategy? If so, which fault tolerance strategy is suitable? (2) Assuming that task t_3 (*Payment*) is non-refundable, and task t_4 (*Delivery*) is unreliable. The failure of t_4 (*Delivery*) will lead to inconsistency of the process, since the user has paid the money (which cannot be refunded) but cannot get the good due to delivery fails. How do we detect and avoid such kinds of consistency violations? (3) Tasks t_3 and t_4 are semantically related. It is incorrect to pay one company (e.g., *eBay.Payment()*) and require another company who did not receive any money to deliver the good (e.g., *Amazon.Deliver()*). How to apply fault tolerance strategy for such kind of semantically-related tasks? (4) Service users have different preferences and may provide local constraints for a single task or global constraints for a whole service plan. Under both local constraints and global constraints, how do we determine optimal fault tolerance strategy for the service plan?

This paper addresses the above challenges by proposing a systematic framework for selecting fault tolerance strategy, which defines QoS model of Web services, identifies commonly-used fault tolerance strategies, and designs selection algorithms to attack these challenges.

3 Preliminaries

Fig. 2 shows an overview of how to employ our framework to select an optimal fault tolerance strategy for service-oriented systems. Fig. 2 includes a number of service users, a communication bus (usually the Internet), and a lot of Web services. The execution engine is in charge of selecting and invoking Web services to fulfill the tasks in the service plan. The execution engine includes several modules: *QoS Model*, *Composition Model*, *Fault Tolerance Strategies*, *Consistency Checking*, *and Fault Tolerance Strategy Selection*. Details of the first four modules will be introduced in Sections 3.1 to 3.4, respectively,

1. http://www.wsdream.net.

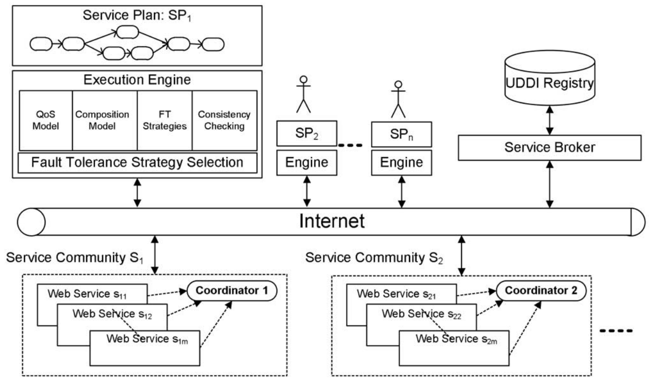


Fig. 2. Conceptual overview.

TABLE 1
Formulas for Basic Compositional Structures

QoS k=	Basic Compositional Structures			
	Sequence	Parallel	Branch	Loop
1,5,7	$\prod_{i=1}^{n} q_i^k$	$\prod_{i=1}^{n} q_i^k$	$\sum_{i=1}^{n} p_i q_i^k$	$\sum_{i=0}^{n} p_i (q_1^k)^i$
2,3,4	$\sum_{i=1}^{n} q_i^k$	$\sum_{i=1}^{n} q_i^k$	$\sum_{i=1}^{n} p_i q_i^k$	$\sum_{i=0}^{n} p_i q_1^k i$
6,8	$\sum_{i=1}^{n} q_i^k$	$\max_{i=1}^{n} q_i^k$	$\sum_{i=1}^{n} p_i q_i^k$	$\sum_{i=0}^{n} p_i q_1^k i$

and details of selecting fault tolerance strategies will be presented in Section 4.

The work procedures of Fig. 2 are as follows: (1) A service provider obtains the address of a certain service community from the UDDI and registers its Web service in the service community; (2) a service user designs a service plan; (3) the execution engine obtains a list of service candidates for each task in the service plan from the service communities; (4) the *consistency checking* module checks whether the service plan will cause inconsistency; (5) the *fault tolerance strategy selection* module determines optimal fault tolerance strategies for the tasks in the service plan; (6) the execution engine executes the service plan by invoking selected Web services and activating selected fault tolerance strategies to mask faults; and (7) the execution engine records the QoS values of the invoked Web services, sends them to the community coordinators, and obtains updated QoS from the community coordinator from time to time.

3.1 QoS Model of Web Services

In the presence of multiple service candidates with identical or similar functionalities, QoS (quality-of-service) properties provide non-functional characteristics for service selection. Based on the previous investigations [2], [20], [33], we identify several representative QoS properties of Web services in the following:

1. **Availability (av)** q^1: the probability that a Web service is operational. The value of availability is in the range of [0,1].
2. **Price (pr)** q^2: the fee that a service user has to pay for invoking a Web service.
3. **Popularity (po)** q^3: the number of totally received invocations of a Web service.
4. **Data-size (ds)** q^4: the size of the Web service invocation response.
5. **Success-probability (sp)** q^5: the probability that a request is successfully completed at the server side and the corresponding response is successfully received by the service requestor.
6. **Response-time (rt)** q^6: the average time duration between a service user sending a request and receiving a response.
7. **Overall Success-probability (osp)** q^7: the average value of the invocation success probability (q^5) of a Web service observed by different service users.
8. **Overall Response-time (ort)** q^8: the average value of the response-time (q^6) of a Web service observed by different service users.

In the above QoS model, q^1-q^4 are provided by service providers and are the same for all service users. q^5 and q^6 are measured at the user-side since they are affected by communication links. Besides these commonly-used QoS properties of Web services, we also consider the *overall success-probability* (q^7) and *overall response-time* (q^8). Given the above QoS properties, the QoS performance of a Web service can be represented as $q = (q^1, \ldots, q^8)$. More QoS properties can be added in the future easily since our QoS model is extensible.

The overall performance of Web services (i.e., q^7 and q^8) provides helpful information for better Web service selection, especially when a user is new and has no idea on the performance of different service candidates. For example, overall success probability of 99.9% indicates that a service candidate has been successfully invoked in most cases. Most likely, this service candidate is better than another service candidate with 50% overall success probability. To obtain the vales of q^7 and q^8, we have designed a user-collaborative QoS evaluation mechanism for Web services, together with its prototyping system WS-DREAM [35]. In WS-DREAM, the service users are encouraged to contribute their individually observed QoS data of Web service to exchange for the data of other users. In this way, service users can obtain the QoS data of other users.

3.2 Service Composition Model

There are two types of services, i.e., atomic service and composite service. An *atomic service* is a self-contained Web service that provides service to users independently without relying on any other Web services, while a *composite service* represents a Web service that provides service by integrating other Web services. Atomic services can be aggregated by different compositional structures (i.e., sequence, branch, loop, and parallel) that describe the order in which a collection of tasks is executed. The QoS values of composite services by these structures can be calculated by the formulas in Table 1. In the *branch* structure, $\{p_i\}_{i=1}^n$ is a set of branch execution probabilities, where $\sum_{i=1}^n p_i = 1$. In the *loop* structure, $\{p_i\}_{i=0}^n$ is a set of probabilities of executing the loop for i times, where n is the maximum loop times and $\sum_{i=0}^n p_i = 1$. In the $parallel$ structure, the *response-time (rt)* is the maximal value of the n parallel branches. The $parallel$ structure is counted as a success if and only if all the n branches succeed.

Algorithm 1. FlowQoS.

```
Input: $SP$: a service plan
Output: $q$: QoS values of the service plan
switch structure type do
    case atomic task $t$
        $q = QoS(t)$;
        return $q$;
    case sequence
        //$SP_i$ is the sub service plans in the sequence.
        foreach $SP_i$ do $q_i = flowQoS(SP_i)$;
        //$l$ is the number of sub service plans.
        $q = sequence(q_1, \ldots, q_l)$;
        return $q$;
    case branch-split
        foreach $SP_i$ do $q_i = flowQoS(SP_i)$;
        $q = branch(P, q_1, \ldots, q_l)$;
        return q;
    case Parallel-split
        foreach $SP_i$ do $q_i = flowQoS(SP_i)$;
        $q = parallel(q_1, \ldots, q_l)$;
        return q;
    case loop-enter
        $q_1 = flowQoS(SP_1)$;
        $q = loop(P, q_1)$;
        return q;
    end
end
```

The basic structures can be nested and combined in arbitrary ways. For calculating the aggregated QoS values of a service plan, we decompose the service plan to basic structures hierarchically. As the example shown in Fig. 3, a service plan is decomposed into basic compositional structures, which will employ the formulas in Table 1 to calculate the aggregated QoS values. Algorithm 1 is designed to calculate the aggregated QoS values of a service plan hierarchically. The QoS values of the sub-plans can be stored for reducing the recalculation time when QoS performance of some tasks in the service plan are updated. For example, when the QoS values of t_3 in Fig. 3 are updated, we only need to recalculate the QoS values of the service plans SP_{112}, SP_{11}, and SP_1. The QoS values of SP_{111} and SP_{12} do not need recalculation, since their values remain the same. This design will speedup the QoS recalculation, especially when the QoS values are updated frequently.

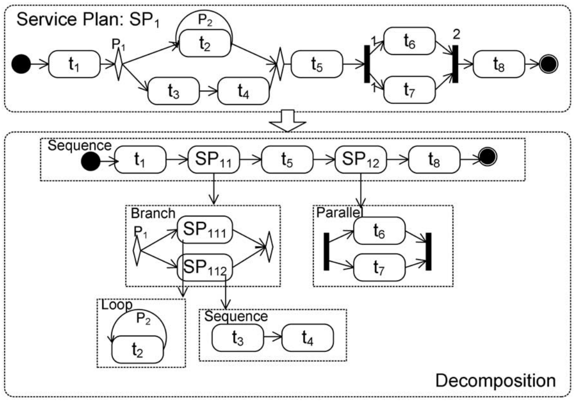


Fig. 3. Example of service plan decomposition.

3.3 Fault Tolerance Strategies

To build reliable service-oriented systems, functionally equivalent service candidates can be employed for tolerating faults [28]. In this paper, a fault tolerance strategy represents a specified approach for masking software faults. Four well-known fault tolerance strategies are employed in this paper, i.e., Retry, Recovery block, N-version programming, and Active. Retry is a very simple and widely-employed strategies for tolerating software faults. Recovery block [25] and N-version programming (NVP) [3] are two well-known fault tolerance strategies in the field of software reliability. Active is a variation of NVP, which employs the first returned response as the final result instead of the voting result. Similar to the way to compute QoS values of various compositional structures, QoS values of different fault tolerance strategies can also be calculated from the QoS of the selected Web services. The formulas for calculating the QoS values of the fault tolerance strategies are listed in Table 2.

- **Retry**. The original Web service will be tried for a certain number of times if it fails. In Table 2, m ($m \geq 2$) is the maximal number of executions of the original Web service. p_i is the probability that the Web service will be executed for i times, where the first $i-1$ executions fail and the ith execution succeed. p_i can be calculated by $p_i = (1-q^5)^{(i-1)} \times q^5$, where q^5 is the *success-probability* of the original Web service. We assume that service failures are independent. When the Web service failure is persistent instead of transient, an immediate retry is unlikely to succeed. In this case, other fault tolerance strategies are needed.

TABLE 2
Composition Formulas for Fault Tolerance Strategies

QoS k=	Fault Tolerance Strategies			
	Retry	RB	NVP	Active
1,5,7	$1\text{-}(1\text{-}q^k)^m$	$1\text{-}\prod_{i=1}^{m}(1\text{-}q_i^k)$	$\sum_{i=\frac{m+1}{2}}^{m} S^k(i)$	$1\text{-}\prod_{i=1}^{m}(1\text{-}q_i^k)$
2,3,4	$q^k \sum_{i=1}^{m} p_i i$	$\sum_{i=1}^{m} p_i (\sum_{j=1}^{i} q_j^k)$	$\sum_{i=1}^{m} q_i^k$	$\sum_{i=1}^{m} q_i^k$
6,8	$q^k \sum_{i=1}^{m} p_i i$	$\sum_{i=1}^{m} p_i (\sum_{j=1}^{i} q_j^k)$	$\max_{i=1}^{m} q_i^k$	$\min_{i=1}^{m} q_i^k$

- **Recovery Block (RB)**. Another standby service candidate will be invoked sequentially if the primary Web service fails. When using recovery blocks, an acceptance test is needed to identify whether a response is success or not. Acceptance tests are usually application independent and need to be implemented by application designers. In Table 2, m ($m \leq$ *number of candidates*) is the maximal number of recoveries, q_i^k means the kth QoS value of the ith service candidate, and p_i is the probability that the ith candidate will be executed. p_i can be calculated by $p_i = q_i^5 \prod_{j=1}^{i-1}(1 - q_j^5)$. Please note that the symbol p_i represents different meaning in Retry and RB in Table 2.
- **N-Version Programming (NVP)**. All the m functionally equivalent service candidates are invoked in parallel and the final result will be determined by majority voting. In Table 2, when $k = 1, 5, 7$, $S^k(i)$ means the probability that i parallel candidate from all the m candidates succeed. For example, when $m = 3$, $q^k = S^k(2) + S^k(3)$, where $S^k(2) = q_1^k q_2^k (1 - q_3^k) + q_1^k (1 - q_2^k) \cdot q_3^k + (1 - q_1^k) q_2^k q_3^k$ and $S^k(3) = q_1^k q_2^k q_3^k$.
- **Active**. All the m candidates are invoked in parallel and the first returned response without communication errors will be employed as the final result.

Using the formulas in Table 2, the aggregated QoS values of different fault tolerance strategies can be calculated. The QoS properties are divided into three groups in Table 2 based on their own features. For example, for the *Active* strategy, the aggregated QoS values of *price* (q^2), *popularity* (q^3) and *data-size* (q^4) are the value sum of their parallel Web services, while the aggregated QoS values of *response-time* (q^6) and *overall response-time* (q^8) are the minimum values of their parallel Web services. In Table 2, some of the calculations are deterministic while others are probabilistic, so the result is an expected value rather than the actual value.

3.4 Consistency Checking

To detect inconsistency problems in complex service plans, we propose two properties for the tasks in the service plans:

1. **Compensable**: A task is compensable if its effects can be undone after committing. In case the cost of compensating the task is unacceptable, the task is non-compensable. For example, a *payment* task is non-compensable if it is non-refundable.
2. **Reliable**: A task is reliable if its execution success-probability is higher than a predefined threshold.

The *compensable* and *reliable* properties of a task t_i are presented as $C(t_i)$ and $R(t_i)$, respectively, where $C(t_i) = true$ means task t_i is compensable and $C(t_i) = false$ means task t_i is non-compensable. In contrast to the previous approaches [10], [31], our *reliable* property is quantified, which makes our consistency checking approach more practical. In our approach, service users can present their judgement on whether a task is reliable or not by setting a threshold.

Before proposing our consistency checking algorithm, we first simplify a service plan by transforming the *loop* structures to *branch* structures using the *loops peeling* technique [2], where loop iterations are presented as a sequence of branches and each branch condition indicates whether the loop has to continue or has to exit. We then decompose a service plan to different execution routes. An execution route is defined as:

Definition 1. *Execution route (ER_i) is a sub service plan ($ER_i \subseteq SP$) which includes only one branch in each branch structure. Each execution route has an execution probability $pro(ER_i)$, which is the product of all probabilities of the selected branches in the route.*

For example, the service plan shown in Fig. 3 includes two execution routes, i.e., $ER_1 = (t_1, t_2, t_5, t_6, t_7, t_8)$, and $ER_2 = (t_1, t_3, t_4, t_5, t_6, t_7, t_8)$. Each execution route can be further decomposed into a set of sequential routes. A sequential route is defined as:

Definition 2. *Sequential route (SR_i) is a sub service plan ($SR_i \subseteq SP$) which includes only one branch in each parallel structure and only one branch in each branch structure of a service plan.*

For example, ER_2 in the above example can be decomposed into two sequential routes, i.e., (t_1, t_3, t_4, t_5, t_6, t_8) and (t_1, t_3, t_4, t_5, t_7, t_8). In this way, a service plan can be decomposed into a set of sequential routes. Each sequential route includes a set of tasks which are executed sequentially. A service plan satisfies consistency checking if and only if no unreliable tasks are executed after non-compensable tasks in every sequential route, which is formalized as follows:

Definition 3. *A sequential route satisfies consistency checking if and only if: $\neg\exists t_i, t_j \in SR : C(t_i) = false \land R(t_j) = false \land j > i$.*

A service plan satisfies consistency checking if and only if all its sequential routes satisfy consistency checking. Algorithm 2 is designed to check whether a service plan satisfies the consistency requirement. Using this algorithm, a designer can discover a consistency violation of a service plan at design time and improve the design to avoid causing any inconsistency problems.

Algorithm 2. Consistency Checking of a Service Plan.

```
Input: a service plan $SP$
Output: true or false, and the violation task pairs if false
1 $SR =$ get a set of sequential routes from $SP$;
2 int routeNumber $= |SR|$;
3 for $(i = 1; i \leq routeNumber; i{+}{+})$ do
4     if $check(SR_i) == false$ then
5         return false;
6     end
7 end
8 return true;
```

Algorithm 3. Function check (*SequentialRoute SR_i*).

```
1 $T =$ get the tasks from $SR_i$;
2 int taskNumber $= |T|$;
3 int flag $= 0$;
4 for $(i = taskNumber; i \geq 1; i{-}{-})$ do
```

```
5        if $flag == 0$ then
6            if $(R(t_i) == false)$ then flag = i;
7        else
8            if $(C(t_i) == false)$ then return false;
9        end
10   end
11   return true;
```

4 Fault Tolerance Strategy Selection

4.1 Notations and Utility Function

Table 3 summaries the major notations used in the reminder of this paper. Given a service plan SP, the set of tasks in the service plan is denoted by T. For a task $t_i \in T$, let $S_i = candidate(t_i)$ be the set of candidate Web services for implementing t_i. Each candidate $s_{ij} \in S_i$ has a quality vector $q_{ij} = (q_{ij}^k)_{k=1}^c$ representing the nonfunctional QoS characteristics, where c is the number of QoS properties. We assume that values of QoS properties are real numbers in a bounded range with minimum and maximum values. A larger value means better quality for some QoS properties (e.g., *availability* and *popularity*), whereas a lower value means better quality for other QoS properties (e.g., *price* and *response-time*). For consistency purpose, we transform all the former QoS properties to be the latter format (i.e., lower value means better quality) by:

$$q_{ij}^k = \max q^k - q_{ij}^k. \quad (1)$$

We then normalize values of QoS properties, which have different scales, to be within the interval of [0,1] by employing the Simple Additive Weighting technique [5]:

$$q_{ij}^k = \begin{cases} \dfrac{q_{ij}^k - \min q^k}{\max q^k - \min q^k}, & \text{if} \quad \max q^k \neq \min q^k, \\ 1, & \text{if} \quad \max q^k = \min q^k, \end{cases} \quad (2)$$

where $\min q^k$ and $\max q^k$ are the minimum and maximum QoS values of the kth QoS property, respectively. In the remainder of this paper, we assume that this transformation has been applied, and represent the value of the kth QoS property of the jth candidate for the ith task as q_{ij}^k, which is in the interval of [0,1] where a smaller value represents better quality. To quantify the performance of a candidate s_{ij}, a utility function is defined as:

$$u_{ij} = utility(s_{ij}) = \sum_{k=1}^{c} w_k \times q_{ij}^k, \quad (3)$$

where u_{ij} is the utility value of the jth candidate of task i and w_k is the user-defined weight of the kth QoS property ($\sum_{k=1}^c w_k = 1$). By setting the values of w_k, users can prioritize the different QoS properties.

4.2 Selection Candidates

For each task in a service plan, there are two types of candidates that can be adopted for implementing the task: (1) Atomic services without any fault tolerance strategies, and (2) services with fault tolerance strategies (e.g., *Retry, RB, NVP and Active*).

TABLE 3
Notations of the Selection Algorithm

Notations	Description
SP	a service plan, which is a triple (T, P, B)
T	a set of tasks in the service plan, $T = \{t_i\}_{i=1}^n$
n	number of tasks in a service plan
S_i	a set of service candidates for t_i, S_i=$\{s_{ij}\}_{j=1}^{m_i}$
m_i	number of candidates of task t_i
q_{ij}	a quality vector for candidate s_{ij}, q_{ij}=$(q_{ij}^k)_{k=1}^c$
c	the number of QoS properties
u_{ij}	the utility value of the candidate s_{ij}
LC_i	local constraints for task t_i, LC_i=$\{lc_i^k\}_{k=1}^c$
GC	global constraints for SP, $GC = \{gc^k\}_{k=1}^c$
z_{ij}	set to 1 if s_{ij} is selected and 0 otherwise
ρ	a set of optimal candidate indexes
ER	a set of execution routes in SP
SR	a set of sequential routes in SP

The fault tolerance strategies include a number of variations based on different configurations. For the $Retry$ strategy, there are a total of $(r-1)e$ variations, where r is the maximal number of executions of $Retry$, and e is the number of alternative atomic services. Note we do not consider the case of one execution to be a fault tolerant strategy. For the *RB, NVP* and *Active* strategies, there are $(e-1)$ variations for each, where each variation uses the top x $(2 \leq x \leq e)$ best performing candidate services, identified by their utility values (Equation (3)). By selecting the top x candidate services, the possible combinations of different candidates is greatly reduced. The number of candidates for a task t_i in a service plan can be calculated by $m_i = atomicService + basicFTStrategies = e + ((r-1)e + 3(e-1))$. In reality, the values of r and e are usually very small, making the total number of candidates acceptable. If there are too many atomic services (the value if e is too large), we can reduce the value of e by only considering a subset of the best performing candidates based on their utility values. Since our selection framework is extensible, new candidates (e.g., new atomic services or new fault tolerance strategies) can be added easily in the future without fundamental changes.

To select the optimal fault tolerance strategy for a service plan, we model the target problem as a candidate selection problem. By solving the problem, suitable candidates are determined for the tasks. In case the selection result for a task is an atomic service, it indicates that no fault tolerance strategy is required for this task (e.g., the service candidate is already performing well).

4.3 Candidate Selection with Local Constraints

Local constraints ($LC_i = \{lc_i^k\}_{k=1}^c$) specify user requirements for a single task t_i in a service plan. For example, *response-time of the task t_i has to be smaller than 1000 milli-seconds* is a local constraint. For each task, there are c local constraints for the c QoS properties, respectively. Since service users may only provide a small number of local constraints, the untouched local constraints are set to be $+\infty$ by default, so that all candidates meet the constraints. The candidate selection problem for a single task t_i with local constraints can be formulated mathematically as:

Problem 1.

$$\textbf{Minimize: } \sum_{j=1}^{m_i} u_{ij} z_{ij},$$

Subject to:

- $\sum_{j=1}^{m_i} q_{ij}^k z_{ij} \leq lc_i^k (k = 1, 2, \ldots, c),$
- $\sum_{j=1}^{m_i} z_{ij} = 1,$
- $z_{ij} \in \{0, 1\}.$

In Problem 1, z_{ij} is used as an indicator ($z_{ij} = 1$ if the candidate s_{ij} is selected and $z_{ij} = 0$ otherwise), $q_{ij} = (q_{ij}^k)_{k=1}^c$ is the QoS vector of candidate s_{ij}, u_{ij} is the utility value of the candidate s_{ij} calculated by Equation 3, and m_i is the number of candidates of task t_i.

To solve Problem 1, for a task t_i, we first use the formulas in Table 2 to calculate the aggregated QoS values of the fault tolerance strategy candidates. Then Algorithm 4 can be employed to select the optimal candidate for each task in a service plan. Firstly, utility values of candidates which meet local constraints are calculated by Equation 3 (line 5). Then, the index of the candidate with the smallest (best) utility value is recorded by setting $\rho_i = x$ (lines 8-9). Finally, the indexes ρ of optimal candidates are returned (line 11).

Algorithm 4. Candidate Selection with LC.

```
Input: Service plan $SP$, local constraints $LC$, candidates $S$
Output: a set of optimal candidate indexes $\rho$.
1  $n =$ number of tasks;
2  $m_i =$ number of candidates of the task $t_i$;
3  for ($i = 1; i \leq n; i{+}{+}$) do
4      for ($j = 1; j \leq m_i; j{+}{+}$) do
5          if $q_{ij}$ meets $LC$ then $u_{ij} = utility(s_{ij})$;
6      end
7      if no candidate meets $LC$ then Throw exception;
8      Select $u_{ix}$ which has minimal utility value $u_{ij}$;
9      $\rho_i = x$;
10 end
11 return $\rho$;
```

Algorithm 5. Candidate Selection for Semantically-Related Tasks with LC.

```
Input: Service plan $SP$, a set of semantically-related tasks
       $SRT$, local constraints $LC$, and candidates $S$
Output: a set of optimal candidate indexes $\rho$.
1  $n =$ number of semantically-related tasks $|SRT|$;
2  $m_i =$ number of candidates of the $i$th semantically-related
   task $SRT_i$;
3  for ($i = 1; i \leq n; i{+}{+}$) do
4      for ($j = 1; j \leq m_i; j{+}{+}$) do
5          if candidate meets $LC$ then
6              $q = flowQoS(SP, s_{ij})$;
7              $u_{ij} = utility(q)$;
8          end
9      end
10     if no candidate meets $LC$ then Throw exception;
11     Select $u_{ix}$ which has minimal utility value $u_{ij}$;
12     forall tasks in $SRT_i$ do $\rho_{ik} = x$;
13 end
14 return $\rho$;
```

As discussed in Section 2, a service plan may contain semantically-related tasks. A semantically-related task includes multiple tasks which have strong correlation (e.g., contain state dependency) with each other. The optimal candidates for the tasks within the same semantically-related task need to be selected together. For example, as shown in Fig. 1, t_3 (*Payment*) and t_4 (*Delivery*) are semantically related. Assume there are two candidates to implement these tasks, i.e., Amazon and eBay. If we select optimal candidates for these two tasks independently, the selection results may be: *eBay.Payment() + Amazon.Delivery()*. However, since these two tasks need to maintain states across them, it is inconsistent to pay eBay and require Amazon who did not receive any money to deliver the good. Therefore, the optimal candidates for these two tasks should be provided by the same provider. For example, for task t_3 and task t_4, there are two candidates, i.e., candidate 1: *Amazon.Payment() + Amazon.Delivery()*, and candidate 2: *eBay.Payment() + eBay.Delivery()*.

To select optimal candidates for the semantically-related tasks, Algorithm 5 can be employed. In this algorithm, firstly, if a candidate meets local constraints, the overall QoS value of the whole service plan with this candidate is calculated by Algorithm 1 (line 6), and the utility value of the service plan with this candidate is calculated by Equation (3) (line 7). After that, the candidate with the best utility performance is selected as the optimal candidate for a semantically-related task (lines 11-12). The above procedure is applied to different semantically-related tasks one by one to identify the optimal candidates. Finally, the optimal indexes of the selected candidates are returned (line 14).

4.4 Candidate Selection with Global Constraints

Local constraints require service users to provide detailed constraint settings for individual tasks, which is time consuming and requires good knowledge of the tasks. Moreover, local constraints cannot specify user requirements for the whole service plan, such as *the response-time of the whole service plan should be smaller than 5000 milli-seconds*. To address these drawbacks, we employ global constraints ($GC = \{gc\}_{i=1}^c$) for specifying user constraints for the whole service plan.

As shown in Section 3.4, a service plan may include multiple execution routes. To ensure that a service plan meets the global constraints, each execution route should meet the global constraints. For determining optimal candidates for a service plan under global constraints, the simplest way is employing an exhaustive searching approach to calculate utility values of all candidate combinations and select out the one which meets all the constraints and with the best utility performance. However, the exhaustive searching approach is impractical when the task number or candidate number is large, since the number of candidate combinations $\prod_{i=1}^{n} m_i$ increases exponentially, where m_i is the candidate number for task t_i and n is the task number in the service plan.

To determine the optimal candidates for a service plan under both global and local constraints, we model the candidate selection problem as a 0-1 Integer Programming (IP) problem as follows:

Problem 2.

Minimize:

$$\sum_{ER_l \in SP} freq_l \times utility(ER_l), \tag{4}$$

Subject to:

$$\forall l, \sum_{i \in ER_l} \sum_{j \in S_i} q_{ij}^k z_{ij} \leq gc^k, (k = 2, 3, 4), \tag{5}$$

$$\forall h, \sum_{i \in SR_h} \sum_{j \in S_i} q_{ij}^k z_{ij} \leq gc^k, (k = 6, 8), \tag{6}$$

$$\forall l, \prod_{i \in ER_l} \prod_{j \in S_i} (q_{ij}^k)^{z_{ij}} \leq gc^k, (k = 1, 5, 7), \tag{7}$$

$$\forall i, \sum_{j \in S_i} z_{ij} = 1, \tag{8}$$

$$z_{ij} \in \{0, 1\}. \tag{9}$$

In Problem 2, Equation (4) is the objective function, where $freq_l$ and $utility(ER_l)$ are the execution frequency and utility value of the lth execution route, respectively. The detailed definition of $utility(ER_l)$ will be introduced in the later part of this section. Equation (5) is the global constraints for the *price, popularity* and *date-size* ($q^k, k = 2, 3, 4$), where the aggregated QoS values of an execution route are the sum of all tasks within the route. Equation (6) is the global constraints for *response-time* and *overall response-time* ($q^k, k = 6, 8$). In a service plan, an execution route may have parallel execution and thus includes multiple sequential routes. The response time of an execution route is equal to the maximal response time of its sequential routes. If all the sequential routes in an execution route meet the global constraint, then this execution route meets the global constraint. Therefore, for q^6 and q^8, all sequential routes should meet the global constraints to make sure that every execution of the service plan meets the global constraints. In sequential routes, the aggregated QoS values are the sum of QoS values of all tasks within the route. Equation (7) is the global constraints for the *availability, success-probability and overall success-probability* ($q^k, k = 1, 5, 7$), where the aggregated QoS values of an execution route are the product of all tasks within the route. In Equation (7), z_{ij} is employed as an indicator. If $z_{ij} = 0$, then $(q_{ij}^k)^{z_{ij}} = 1$, indicating that the candidate is not selected. For the tasks that are semantically related, we ensure that these tasks must be implemented by the same Web service provider (the same candidate index). We assume that service candidates are numbered consistently within the sets of semantically-related tasks. Equations (8) and (9) are employed to ensure that only one candidate will be selected for each task in the service plan, where $z_{ij} = 1$ and $z_{ij} = 0$ indicate that a candidate j is selected and not selected for task i, respectively. In Integer Programming, the objective function and constraint functions should be linear. Therefore, we need to transform Equation (7) from non-linear to linear. By applying the logarithm function to Equation (7), we obtain a linear equation:

$$\forall l, \sum_{i \in ER_l} \sum_{j \in S_i} z_{ij} ln(q_{ij}^k) \leq ln(gc^k) (k = 1, 5, 7). \tag{10}$$

The objective function needs to be changed accordingly. We define the execution route utility function in the new objective function as:

$$utility(ER_l) = \sum_{k=1}^{c} w_k \times q_{ER_l}^k. \tag{11}$$

In Equation (11), c is the number of QoS properties, w_k is the user-defined weight for the QoS properties, and $q_{ER_l}^k$ is the aggregated QoS value of the execution path ER_l, which can be calculated by:

$$q_{ER_l}^k = \begin{cases} \sum\limits_{i \in ER_l} \sum\limits_{j \in S_i} z_{ij} ln(q_{ij}^k), (k = 1, 5, 7) \\ \sum\limits_{i \in ER_l} \sum\limits_{j \in S_i} z_{ij} q_{ij}^k, (k \neq 1, 5, 7). \end{cases} \tag{12}$$

In this way, the fault tolerance strategy selection problem is formulated as a 0-1 IP problem. The IP problem is NP-Complete [8]. The problem solving time increases exponentially with the problem size, which makes runtime reconfiguration impractical for complex service plans. Therefore, it is not feasible to solve the 0-1 IP problem using an exhaustive search. The well-known Branch-and-Bound algorithm [14] can be employed to reduce the search space. To further speedup the computation process, we propose a heuristic algorithm to efficiently solve the fault tolerance strategy selection problem in the following section.

4.5 Heuristic Algorithm FT-HEU

Algorithm 6. FT-HEU.

```
  Input: $SP$, $GC$, $LC$, $S$
  Output: a set of optimal candidate indexes $\rho$
1 $\rho = \text{findInitSol}(SP, GC, LC, S)$;
2 $q_{all} = flowQoS(SP, \rho_1, \dots, \rho_n)$;
3 while ($q_{all}$ does not meet $GC$) do
4     $S' = \text{findExCandidate}(SP, GC, LC, S, \rho)$;
5     if $|S'| == 0$ then
6         throw exception FeasibleSolutionNotFound
```

```
7       else
8           forall $s_{xy} \in S'$ do $\rho_x = y$;
9       end
10      $q_{all} = flowQoS(SP, \rho_1, \ldots, \rho_n)$;
11  end
12  repeat
13      $\rho = \text{feasibleUpgrade}(SP, GC, LC, S, \rho)$;
14  until $\rho$ do not change;
15  return $\rho$;
```

For a service plan, a solution is a set of candidate selection results for the tasks. A solution is a *feasible solution* if the selected candidates meet all their corresponding local constraints as well as all the global constraints. Otherwise, it is an *infeasible solution*. To solve the 0-1 IP problem efficiently, we propose a heuristic algorithm *FT-HEU* in Algorithm 6 by extending and customizing traditional heuristic algorithms. Our proposed *FT-HEU* algorithm integrates the elements we have described in this paper (e.g., local constraints and global constraints, flowQoS(), user-defined weights of QoS properties, etc.) to solve the fault tolerance selection problem. Compared with traditional heuristic algorithms, our *FT-HEU* algorithm explores the following capacities: (1) When selecting candidates, *FT-HEU* considers local constraints which are not considered in traditional heuristic algorithms; (2) When calculating aggregated QoS values, *FT-HEU* employs our proposed flowQoS() algorithm to handle the QoS aggregation of different compositional structures; and (3) in *FT-HEU*, we propose *accumulated feasible value, infeasible factor*, and *QoS saving* for the fault tolerance strategy selection problem.

Algorithm 6 includes the following steps:

Step 1 (line 1): The function $findInitialSol()$ is invoked to find an initial solution for the service plan SP.

Step 2 (lines 2-11): The Function $flowQoS()$ is employed to get the aggregated QoS values of the initial solution. If the initial solution cannot meet the global constraints (*infeasible*), then the $findExCandidate()$ function is invoked to find an exchangeable candidate to improve the solution. If such an exchangeable candidate cannot be found, then the *FeasibleSolutionNotFound* exception will be thrown to the user. Otherwise, the above candidate-exchanging procedures will be repeated until a feasible solution becomes available.

Step 3 (lines 12-15): Iterative improvement of the feasible solution by invoking the $feasibleUpgrade()$ function. The final solution will be returned when the values of ρ do not change in the iterations.

We provide a brief introduction to the functions $findInitialSol()$, $findExCandidate()$, and $feasibleUpgrade()$ in Sections 4.5.1 to 4.5.3, respectively. More technical details of these functions are provided in the Appendix, which can be found in the Computer Society Digital Library at https://doi.ieeecomputersociety.org/10.1109/TC.2013.189/. Additional mentions of supplemental material are available online at http://ieeexplore.ieee.org.

4.5.1 Find Initial Solution: $findInitialSol()$

To find an initial solution for a service plan, we first set the QoS values of all the tasks to be the optimized values (e.g., *response-time* to be 0, *availability* to be 100%, etc.), so that the function $flowQoS()$ (which has been introduced in Algorithm 1) can be employed for calculating the accumulated QoS values for the selected candidates. For example, when the candidates of the first two tasks are selected, $flowQoS()$ will return the accumulated QoS values of the first two tasks, since the values of other unselected tasks are set to be optimal.

To initially select suitable candidate for a task, we first exclude candidates that do not meet the local constraints. After that, $flowQoS()$ is employed to calculate the accumulated QoS values of different candidates. An *accumulated feasible value* λ_{ij} is defined to quantify the feasibility degree of the jth candidate for the ith task:

$$\lambda_{ij} = \sum_{k=1}^{c} w_k \frac{q^k_{all}}{gc^k}, \tag{13}$$

where q^k_{all} is the accumulated QoS value of the selected candidate, calculated by $flowQoS()$, w_k is the weight for the corresponding QoS property, and a smaller λ_{ij} value means the candidate is more suitable. For a task in the service plan, by calculating the λ values of all its candidates, we can determine an initial candidate for a task. By repeating the above procedure to all tasks in the service plan, we can obtain an initial solution.

4.5.2 Find Exchange Candidate: $findExCandidate()$

If the initial solution is infeasible, the function $findExCandidate()$ is invoked to find an exchangeable candidate, which makes the solution feasible. For an *infeasible solution*, the *infeasible factor*, which is calculated by $\frac{q^k_{all}}{gc^k}$, is employed to quantify the degree of infeasibility of the kth QoS property. The exchangeable candidate should meet the following requirements:

- It will decrease the highest *infeasible factor* of the QoS properties.
- It will not increase the *infeasible factor* of any other previously infeasible properties.
- It will not make any previously feasible QoS properties become infeasible.

If there is more than one candidate which meet the above requirements, we will select the one with the largest improvement on the infeasible factor.

4.5.3 Feasible Upgrade: $feasibleUpgrade()$

If the solution is feasible, the $feasibleUpgrade()$ function is invoked to continuously improve the solution. In this function, we iterate all the tasks. For each task, we iteratively replace the original selected candidate with other candidates to find a better solution. In the function, the *QoS saving* v_{ij} is defined as:

$$v_{ij} = \sum_{k=1}^{c} w_k \frac{q^k_{new} - q^k_{old}}{gc^k}, \tag{14}$$

where w_k is the weight for the kth QoS property, q^k_{new} and q^k_{old} represent the accumulated QoS values of the new candidate and original candidate, respectively.

The feasible upgrade procedure includes the following steps: (1) If there exists at least one feasible upgrade which provides QoS savings ($v_{ij} < 0$, indicating that the new candidate is better than the original candidate), the candidate with maximal QoS savings (minimal v_{ij} value) is chosen for exchanging; and (2) if no feasible upgrade with QoS saving exists, the solution that contains the best utility value improvement compared with the old solution will be selected as the new solution.

4.5.4 *Computational Complexity of FT-HEU*

The FT-HEU algorithm has convergence property, since (1) Step 2 (lines 2-11 of Algorithm 6) never makes any feasible QoS property become infeasible or any infeasible QoS property become more infeasible; (2) for each exchange in Step 2, the property with the maximal infeasible factor will be improved; and (3) Step 3 (lines 12-15) always upgrades the utility value of the solution, which cannot cause any infinite looping, since there are only a finite number of feasible solutions.

For calculating the upper bound of the worst-case computational complexity of the FT-HEU algorithm, we assume there are n tasks, m candidates for each task, and c QoS properties in a service plan. In Step 1, when finding the initial solution, the computation of λ_{ij} is $O(nm)$. In Step 2, finding an exchange candidate requires maximal $n(m-1)$ calculations of the alternative candidates, where each calculation will invoke a function $flowQoS()$ with computation complexity of $O(nc)$. Therefore, the computation complexity is $O(n^2mc)$ for each exchange. The $findExCandidate()$ function will be invoked at most $n(m-1)$ times since there are at most $(m-1)$ upgrades for each task. Consequently, the total computation complexity of Step 2 is $O(n^3m^2c)$. In Step 3, for each upgrade, there are $n(m-1)$ iterations for the alternative candidates. For each iteration, the $flowQoS()$ function with computation complexity $O(nc)$ is invoked. Thus, the computation complexity of each upgrade is $O(n^2mc)$. Since there are maximal $n(m-1)$ upgrades for the whole service plan, the total computation complexity of Step 3 is $O(n^3m^2c)$. Since Step 1, Step 2 and Step 3 are executed in sequence, the combined complexity of the whole *FT-HEU* algorithm is $O(n^3m^2c)$.

4.6 Dynamic Reconfiguration

The Internet environment is highly dynamic, where the QoS performance of Web services may change unexpectedly due to internal changes or workload fluctuations. Moreover, new service candidates may become available and requirements of service users may also be updated. Dynamic reconfiguration of the fault tolerance strategy makes the system more adaptive to the dynamic environment. The reconfiguration procedures are as follows: (1) the initial fault tolerance strategy is selected by employing our candidate selection approach; (2) the service-oriented system invokes the remote Web services with the selected fault tolerance strategy, and records their observed QoS performance of the invoked Web services; and (3) the service-oriented system reconfigures the optimal candidates for the tasks when system performance is unacceptable, the renewal time is reached, new candidates become available, or the user requirements are updated. By this reconfiguration procedure, service users can handle the frequent changes of candidate performance as well as user requirements. The reconfiguration frequency is application-dependent and controlled by application designers.

TABLE 4
Locations of Service Users and Web Services

User Locations	**Num**	**Web Service Locations**	**Num**
User Locations	Num	WS Locations	Num
United States	72	United States	33
European Union	45	Canada	10
Japan	6	China	8
Canada	5	Germany	7
Brazil	3	France	6
United Kindom	3	Spain	6
Republic of Korea	2	United Kingdom	5
Cyprus	1	Netherlands	4
Republic of Czech	1	Poland	3
Finland	1	Republic of Korea	3
Greece	1	Switzerland	3
Hungary	1	Italy	2
Ireland	1	Australia	1
Norway	1	Belgium	1
Poland	1	Ireland	1
Portugal	1	Islamic Republic of Iran	1
Puerto Rico	1	Japan	1
Slovenia	1	New Zealand	1
Spain	1	Norway	1
Taiwan	1	Serbia and Montenegro	1
Uruguay	1	South Africa	1
		Thailand	1
Total	**150**	**Total**	**100**

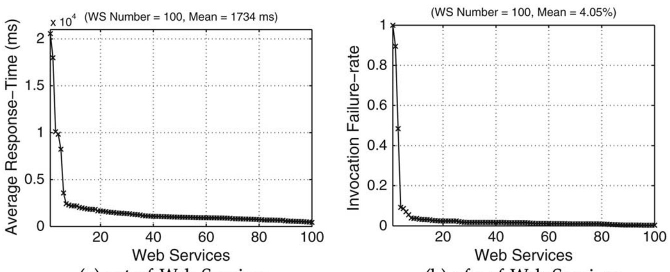
5 Experiments

In this section, we first describe our employment of a real-world prototype to evaluate and collect QoS data of real-world Web services. After that, our fault tolerance strategy selection approach is illustrated by a case study. Finally, the computational time and selection accuracy of various selection algorithms are investigated.

5.1 Implementation and Data Collection

To obtain real-world Web service QoS data, we obtained a list of 21,197 publicly available Web services by crawling Web service information from the Internet. We randomly selected 100 real-world Web services which are located in more than 20 countries for the experiments. 150 computer nodes from Planet-Lab [7], which are distributed in more than 20 countries, were employed to serve as service users to run our client-side evaluation programs for evaluating QoS performance of the selected Web services. Table 4 shows the detailed location information of Web services and service users.

In the experiment, each Web service is invoked by each service user for 100 times. Therefore, there are a total of $100 \times 150 \times 100 = 1,500,000$ Web service invocations being executed. By processing the experimental results, we obtain a 150×100 matrix, where each entry in the matrix is a vector of QoS values observed by a service user on a Web service. Fig. 4 shows the experimental results of overall response-time (*ort*) and overall failure-probability of the 100 Web services. More detailed experimental raw data are provided online at *wsdream.net*. Fig. 4a shows that among the 100 Web services,



(a) ort of Web Services (b) ofp of Web Services

Fig. 4. Overall QoS performance.

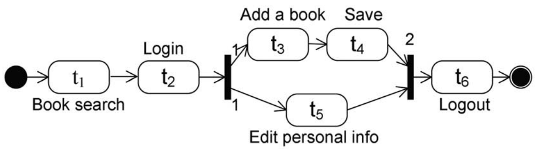


Fig. 5. Service plan for case study.

there are 12 Web services providing longer than 2000 milliseconds response-time for service users. These Web services may require a long time for processing the user requests. In Fig. 4b, most of the Web services maintain small overall failure probabilities, while there is a Web service with 100% invocation failure-probability, which is caused by the permanent unavailability of the Web service.

5.2 Case Study

In this section, we illustrate the fault tolerance strategy selection procedure via a case study: A service user in China (CN) plans to build a simple service-oriented application as shown in Fig. 5, where t_1 does not have dependency with other tasks and t_2–t_6 are semantically related. There are six functionally equivalent *Amazon Web services* (located in US, Japan, Germany, Canada, France and UK, respectively) that can be employed for executing these tasks.

Researchers in different geographic locations (SYSU@CN, SUT@AU, NASA@US, NTU@SG, NTHU@TW, and CUHK@HK) are invited to run our evaluation program to conduct this real-world Web service evaluations. The evaluation results are shown in Tables 5 and 6. In these two tables, the rows represent the six functionally equivalent *Amazon Web services* (named $aus, \ldots, auk$); the columns show response-time (*rt*) and success probability (*sp*) values of distributed service users (named $CN, \ldots, HK$); and Avg represents the *overall response-time* (*ort*) and *overall success-probability* (*osp*) of a particular service observed by different users.

Tables 5 and 6 show that: (1) *response-time* performance is greatly influenced by the communication links. For example, the response-time performance of the user in *US* is much better than the user in *CN* in our experiment; (2) optimal service candidates are different from users to users (e.g, *aus* for the user *US* and *ajp* for the user *AU*); (3) invocation *success-probabilities* are also different from users to users; and (4) the *success-probability* of the semantically-related task t_2–t_6 is lower than that of the task t_1, since the semantically-related task is counted as successful only if all the tasks t_2–t_6 are successful.

TABLE 5
QoS Values of the Task (t_1)

WS	Q	CN	AU	US	SG	TW	HK	Avg
aus	rt	3659	1218	121	544	934	491	681
	sp	0.819	1.000	1.000	1.000	1.000	0.977	0.989
ajp	rt	3310	1052	338	472	824	469	686
	sp	0.788	1.000	1.000	1.000	1.000	0.980	0.987
ade	rt	3233	1476	303	596	1178	612	846
	sp	0.813	1.000	1.000	1.000	1.000	0.973	0.987
aca	rt	3530	1190	130	456	916	509	714
	sp	0.807	1.000	1.000	1.000	0.998	0.983	0.988
afr	rt	3289	1309	306	600	1193	630	864
	sp	0.844	0.998	1.000	1.000	1.000	0.974	0.989
auk	rt	3550	1326	305	671	1178	633	862
	sp	0.837	0.997	1.000	1.000	1.000	0.971	0.988

TABLE 6
Aggregated QoS Values of the Semantically-Related Task (t_2–t_6)

WS	Q	CN	AU	US	SG	TW	HK	Avg
aus	rt	16434	5625	717	2708	4166	2328	3297
	sp	0.450	1.000	1.000	1.000	1.000	0.972	0.940
ajp	rt	14763	4980	1751	2505	3730	2058	3335
	sp	0.450	1.000	1.000	1.000	0.998	0.973	0.944
ade	rt	14640	6718	1646	3038	5209	2730	3985
	sp	0.438	1.000	1.000	1.000	1.000	0.972	0.935
aca	rt	15602	5527	1403	2488	4150	2305	3427
	sp	0.452	1.000	1.000	1.000	0.996	0.979	0.944
afr	rt	14560	5983	2211	3009	5175	2862	4045
	sp	0.496	0.992	1.000	1.000	1.000	0.969	0.937
auk	rt	15898	6066	1630	3044	5209	2819	4048
	sp	0.484	0.988	1.000	1.000	0.998	0.970	0.939

Since the six *Amazon Web Services* are independent systems and t_2–t_6 are semantically related, the optimal candidates for these tasks should be provided by the same Web service. To determine the optimal fault tolerance strategy for the user in China (CN), we set the weights of the eight QoS properties as: $(0, 0.2, 0, 0.2, 0.2, 0.2, 0.1, 0.1)$. The weights of q^1 ($availability$) and q^3 ($popularity$) are set to be 0, since the service provider *Amazon* does not offer any such information. After calculating the candidate utility values, the selection algorithm is employed to determine the optimal candidates. The selection results are as follows: an $active$ strategy with the top 2 performing candidates for t_1, and an $active$ strategy with 3 parallel branches for the semantically-related task t_2–t_6. This selection result is reasonable, since the user in CN is under poor network condition in the experiment, and the $active$ strategy can improve *response-time* performance (by employing the first response as the final result) and improve *success-probability* since it fails only if all the redundant candidates fail.

By employing our fault tolerance strategy selection approach, service users can determine optimal fault tolerance strategies for both single tasks and semantically-related tasks.

5.3 Performance Study of the Selection Algorithms

To study the selection performance, we randomly select different number of Web services to create service plans with different compositional structures and execution routes. We implement three different selection algorithms, i.e., *FT-ALL, FT-BAB, and FT-HEU*. *FT-ALL* represents the exhaustive searching approach introduced in Section 4.4, *FT-BAB* represents the traditional Branch-and-Bound algorithm for solving the IP problem, and *FT-HEU* represents the heuristic

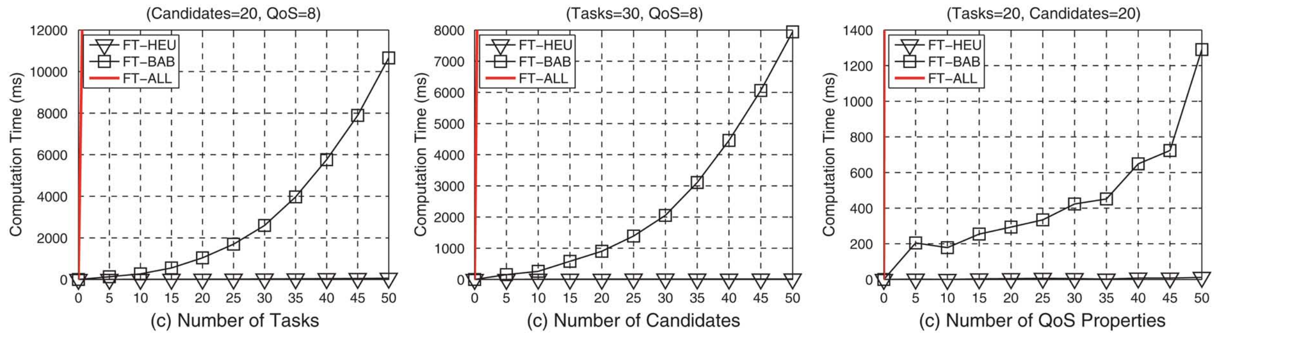


Fig. 6. Performance of computation time.

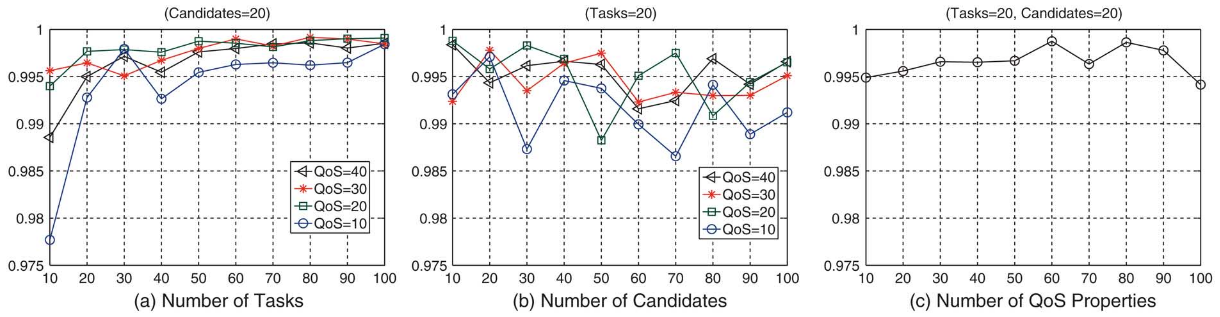


Fig. 7. Performance of selection results.

algorithm shown in Algorithm 6. The configurations of the computers for running the experiments are: Intel(R) Core(TM)2 2.13 G CPU with 1 G RAM, 100Mbits/sec Ethernet card, Window XP and JDK 6.0.

5.3.1 Computation Time

Fig. 6(a)–(c) show the computation time of different selection algorithms with different number of tasks, candidates and QoS properties, respectively. The experimental result shows: (1) the computation time of *FT-ALL* increases exponentially even with a very small problem size (the curve of *FT-ALL* is almost overlap with the y-axis); (2) the computation time of *FT-BAB* is acceptable when the problem size is small; however, it increases quickly when the numbers of tasks, candidates and QoS properties are large; (3) the computation time of *FT-HEU* is very small in all the experiments even with a large problem size; and (4) the curve of FT-BAB in Fig. 6c is more irregular since computation times of different types of QoS properties are different.

5.3.2 Selection Results

Fig. 7 compares the selection results of *FT-BAB* and *FT-HEU* algorithms with different numbers of tasks, candidates and QoS properties. The y-axis of Fig. 7 is the values of *Utility(BAB)/ Utility(HEU)*, which is the utility ratio of the two algo rithms, where the value of 1 means the selection results obtained from *FT-HEU* is identical to that obtained from *FT-BAB*. Fig. 7(a) and (b) show the experimental results of *FT-BAB* and *FT-HEU* with different numbers of tasks and candidates, respectively. The experimental results show that: (1) under different numbers of QoS properties (10, 20, 30 and 40 in the experiment), the utility values of *FT-HEU* are near *FT-BAB* (i.e., larger than 0.975 in the experiments) with different numbers of tasks and candidates; (2) with the increasing of the task number, the performance of *FT-HEU* becomes better. Fig. 7c shows the selection results of *FT-BAB* and *FT-HEU* with different numbers of QoS properties. The results show that the performance of *FT-HEU* is steady with different numbers of QoS properties. The experimental results show that *FT-HEU* algorithm can provide near optimal selection result with excellent computation time performance even under a large problem size. The *FT-HEU* algorithm enables dynamic fault tolerance strategy reconfiguration. *FT-HEU* can be employed in different environments, such as real-time applications (requiring quick-response), mobile Web services (with limited computation resource), and large-scale service-oriented systems (with a large problem size).

6 Discussion and Related Work

Software fault tolerance is widely employed for building reliable service-oriented systems, including passive strategies and active strategies [6], [28]. Passive strategies have been discussed in FT-SOAP [9] and FT-CORBA [30], while active strategies have been investigated in FTWeb [29], Thema [21], WS-Replication [27], SWS [15], and Perpetual [24]. Complementary to the design of various fault tolerance strategies, this paper focuses on selecting optimal fault tolerance strategies for service-oriented systems.

A number of research efforts have been performed in the research topic of QoS-aware Web service selection and composition. Zeng et al. [33] proposed a QoS-aware middleware for Web service selection employing five generic QoS properties (i.e., *execution price, execution duration, reliability, availability, and reputation*). Ardagna and Pernici [2] investigated the problem of adaptive service composition in flexible processes based on five QoS properties (i.e., *execution time, availability, price, reputation, and data quality*). Alrifai and Risse [1] proposed an efficient service composition approach by considering both generic QoS properties and domain-specific QoS properties. Yu et al. [32] designed a combinatory model and a graph model for Web service selection. Some previous work also takes subjective information (e.g., provider reputations, user requirements, etc.) into consideration to enable more accurate Web service selection [26]. These previous efforts investigate the selection of atomic services. However, influenced by quality of the selected services, reliability of the resulting service-oriented systems may not be able to meet user requirements. This paper combines the selection of atomic services together with associated fault tolerance strategies to further enhance reliability of the resulting service-oriented system. This paper models the problem of selecting the optimal fault tolerance strategy as an optimization problem and proposes a heuristic algorithm to efficiently solve the problem. Moreover, the previous efforts assume that tasks in a service plan are independent of each other and Web services can be selected separately for these tasks. However, in reality, it is common that some tasks inherit correlations, where the choice of one Web service implies the choice of another Web service. To address this problem, in this paper, we model the selection of an optimal fault tolerance strategy for semantically-related tasks as a general constraint in the optimization problem.

Hagen and Alonso [10] investigated the *compensable*, *retriable*, and *pivot* transactional properties for exception handling. Ye et al. [31] discussed the *compensable* and *retriable* transactional properties for service-oriented systems from the perspective of atomicity sphere. In contrast to these approaches, we propose a *reliable* property, which is quantifiable. The advantages of our approach include: (1) Our approach can be customized by setting the user-defined threshold. By allowing different service users for different judgements on whether a task is reliable or not, our approach turns out to be more feasible and practical. (2) Traditionally, a task is *retriable* means that the execution of this task will eventually succeed by retying or resorting to other options [10], [31]. However, in the area of service computing, a task may still fail even though it can retry the original Web service or try other alternative Web service candidates. By calculating the detailed execution success-probability of a task, our approach provides more realistic and accurate determination on whether a task is reliable or not.

There are complementary techniques to our approach. The Service Level Agreement (SLA) [17] can be employed to maintain a certain level of service from the service provider to service users. WS-Reliability [23] can be adopted for enabling reliable communications. WSRF [22], which describes the state as XML datasheets, can be employed for transferring states between alternative replicas. Our framework can be integrated into SOA runtime governance middlewares [13] and applied to industry projects.

7 Conclusion

In this paper, we investigate the problem of selecting an optimal fault tolerance strategy for building reliable service-oriented systems with local and global constraints. Comprehensive experiments involving world-wide Web service invocations are conducted. The experimental results show that our proposed FT-HEU selection algorithm can provide near optimal selection results with small computation time.

In the current work, we employ the average values of historical QoS data for making a selection. In the future, more comprehensive investigations will be made on QoS value distributions and their correlations with time and day. When making consistency checking, we will consider compensation cost and transaction commit overheads. When calculating the aggregated execution success-probability, we assume that the failures are independent of each other. It is noted that various Web services, even developed independently, may still experience failure dependency. To probe further, more studies can be carried out on the correlation of failures of different Web services. Our on-going research also includes the investigations of more QoS properties of Web services.

Acknowledgments

The work described in this paper was supported by the National Basic Research Program of China (973 Project No. 2014CB347701), the National Natural Science Foundation of China (Project No. 61100078), the Shenzhen Basic Research Program (Project No. JCYJ20120619153834216), and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. N_CUHK405/11 of the NSFC/RGC Joint Research Scheme).

References

[1] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient QoS-aware service composition," in *Proc. 18th Int. Conf. World Wide Web (WWW'09)*, 2009, pp. 881–890.

[2] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 369–384, Jun. 2007.

[3] A. Avizienis, "The methodology of N-version programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. New York, NY: Wiley, 1995, pp. 23–46.

[4] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu, "Declarative composition and peer-to-peer provisioning of dynamic web services," in *Proc. 18th Int. Conf. Data Eng. (ICDE'02)*, 2002, pp. 297–308.

[5] C.-L. Hwang and K. Yoon, Eds., "Multiple criteria decision making," *Lecture Notes in Economics and Mathematical Systems*. Berlin, Germany: Springer-Verlag, 1981.

[6] P. P.-W. Chan, M. R. Lyu, and M. Malek, "Making services fault-tolerant," in *Proc. 3rd Int. Serv. Availability Symp.*, 2006, pp. 43–61.

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An overlay testbed for broad-coverage services," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.

[8] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[9] C.-L. Fang, D. Liang, F. Lin, and C.-C. Lin, "Fault-Tolerant web services," *J. Syst. Archit.*, vol. 53, no. 1, pp. 21–38, 2007.

[10] C. Hagen and G. Alonso, "Exception handling in workflow management systems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 10, pp. 943–958, Oct. 2000.

[11] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures," *Computer*, vol. 27, no. 9, pp. 60–69, Sep. 1994.

[12] C.-Y. Huang, M. Lyu, and S.-Y. Kuo, "A unified scheme of some non-homogenous Poisson process models for software reliability estimation," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 261–269, Mar. 2003.

[13] M. Kavianpour, "SOA and large scale and complex enterprise transformation," in *Proc. 5th Int. Conf. Serv.-Oriented Comput. (ICSOC'07)*, 2007, pp. 530–545.

[14] S. Khan, K. F. Li, E. G. Manning, and M. M. Akbar, "Solving the knapsack problem for adaptive multimedia systems," *Stud. Inf. Universalis*, vol. 2, no. 1, pp. 157–178, 2002.

[15] W. Li, J. He, Q. Ma, I.-L. Yen, F. Bastani, and R. Paul, "A framework to support survivable web services," in *Proc. 19th IEEE Int. Symp. Parallel Distrib. Process. (IPDPS'05)*, 2005, pp. 93–102.

[16] A. Luckow and B. Schnor, "Service replication in grids: Ensuring consistency in a dynamic, failure-prone environment," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–7.

[17] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "A service level agreement language for dynamic electronic services," *Electron. Commerce Res.*, vol. 3, no. 1–2, pp. 43–59, 2003.

[18] M. R. Lyu, Ed., *Software Fault Tolerance* (Trends in Software). New York, NY: Wiley, 1995.

[19] M. R. Lyu, *Handbook of Software Reliability Engineering*. New York, NY: McGraw-Hill, 1996.

[20] D. A. Menasce, "QoS issues in web services," *IEEE Internet Comput.*, vol. 6, no. 6, pp. 72–75, Nov./Dec. 2002.

[21] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-Fault-Tolerant middleware for web-service applications," in *Proc. 24th IEEE Symp. Reliable Distrib. Syst. (SRDS'05)*, 2005, pp. 131–142.

[22] OASIS. (2005). *Web Service Resource Framework* [Online]. Available: http://www.oasisopen.org/committees/wsrf/.

[23] OASIS. (2005). *Web Services Reliable Messaging Protocol* [Online]. Available: http://specs.xmlsoap.org/ws/2005/02/rm.

[24] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman, "Byzantine fault-tolerant web services for n-tier and service oriented architectures," in *Proc. 28th Int. Conf. Distrib. Comput. Syst. (ICDCS'08)*, 2008, pp. 260–268.

[25] B. Randell and J. Xu, "The evolution of the recovery block concept," in *Software Fault Tolerance*, M. R. Lyu, Ed. New York, NY: Wiley, 1995, pp. 1–21.

[26] S. Rosario, A. Benveniste, S. Haar, and C. Jard, "Probabilistic QoS and soft contracts for transaction-based web services orchestrations," *IEEE Trans. Services Comput.*, vol. 1, no. 4, pp. 187–200, Oct./Dec. 2008.

[27] J. Salas, F. Perez-Sorrosal, M. Patiño Martínez, and R. Jiménez-Peris, "WS-Replication: A framework for highly available web services," in *Proc. 15th Int. Conf. World Wide Web (WWW'06)*, 2006, pp. 357–366.

[28] N. Salatge and J.-C. Fabre, "Fault tolerance connectors for unreliable web services," in *Proc. 37th Int. Conf. Dependable Syst. Netw. (DSN'07)*, 2007, pp. 51–60.

[29] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: A fault tolerant infrastructure for web services," in *Proc. 9th IEEE Int. Conf. Enterprise Comput.*, 2005, pp. 95–105.

[30] G.-W. Sheu, Y.-S. Chang, D. Liang, S.-M. Yuan, and W. Lo, "A fault-tolerant object service on CORBA," in *Proc. 17th Int. Conf. Distrib. Comput. Syst. (ICDCS'97)*, 1997, p. 393.

[31] C. Ye, S. C. Cheung, W. K. Chan, and C. Xu, "Atomicity analysis of service composition across organizations," *IEEE Trans. Softw. Eng.*, vol. 35, no. 1, pp. 2–28, Jan./Feb. 2009.

[32] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end QoS constraints," *ACM Trans. Web*, vol. 1, no. 1, pp. 1–26, 2007.

[33] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware middleware for web services composition," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 311–327, May 2004.

[34] L.-J. Zhang, J. Zhang, and H. Cai, *Services Computing*. Berlin and Beijing: Springer and Tsinghua Univ. Press, 2007.

[35] Z. Zheng and M. R. Lyu, "WS-DREAM: A distributed reliability assessment mechanism for web services," in *Proc. 38th Int. Conf. Dependable Syst. Netw. (DSN'08)*, 2008, pp. 392–397.

[36] Z. Zheng and M. R. Lyu, "A QoS-aware fault tolerant middleware for dependable service composition," in *Proc. 39th Int. Conf. Dependable Syst. Netw.*, 2009, pp. 239–248.

[37] Z. Zheng and M. R. Lyu, "Optimal fault tolerance strategy selection for web services," *Int. J. Web Service Res. (JWSR)*, vol. 7, no. 4, pp. 21–40, 2010.

Zibin Zheng received the PhD degree from Department of Computer Science and Engineering, The Chinese University of Hong Kong, in 2010. He is an associate research fellow at the Shenzhen Research Institute, The Chinese University of Hong Kong. He received Outstanding Thesis Award of CUHK at 2012, ACM SIGSOFT Distinguished Paper Award at ICSE2010, Best Student Paper Award at ICWS2010, and IBM PhD Fellowship Award 2010–2011. He served as a program committee member of IEEE CLOUD2009, SCC2011, SCC2012, ICSOC2012, etc. His research interests include cloud computing, service computing, and software engineering.

Michael R. Lyu is currently a professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include software reliability engineering, distributed systems, service computing, information retrieval, social networks, and machine learning. He has published over 400 refereed journal and conference papers in these areas. He is an AAAS Fellow and a Croucher Senior Research Fellow for his contributions to software reliability engineering and software fault tolerance. He received IEEE Reliability Society 2010 Engineer of the Year Award.