SCIENCE CHINA
Information Sciences

• REVIEW •
Special Focus on High-Confidence Software Technologies

May 2015, Vol. 58 052101:1–052101:12
doi: 10.1007/s11432-015-5313-y

# Service fault tolerance for highly reliable service-oriented systems: an overview

ZHENG ZiBin[1], LYU Michael Rung Tsong[1] & WANG HuaiMin[2]*

[1]*Shenzhen Research Institute, The Chinese University of Hong Kong, Hong Kong* 518057*, China;*
[2]*National Laboratory for Parallel & Distributed Processing, National University of Defense Technology, Changsha* 410073*, China*

Received January 27, 2015; accepted March 2, 2015; published online April 2, 2015

**Abstract** Service-oriented systems are widely-employed in e-business, e-government, finance, management systems, and so on. Service fault tolerance is one of the most important techniques for building highly reliable service-oriented systems. In this paper, we provide an overview of various service fault tolerance techniques, including sections on fault tolerance strategy design, fault tolerance strategy selection, and Byzantine fault tolerance. In the first section, we introduce the design of static and dynamic fault tolerance strategies, as well as the major problems when designing fault tolerance strategies. After that, based on various fault tolerance strategies, in the second section, we identify significant components from a complex service-oriented system, and investigate algorithms for optimal fault tolerance strategy selection. Finally, in the third section, we discuss a special type of service fault tolerance techniques, i.e., the Byzantine fault tolerance.

**Keywords** fault tolerance, software reliability, Web service, SOA

**Citation** Zheng Z B, Lyu M R T, Wang H M. Service fault tolerance for highly reliable service-oriented systems: an overview. Sci China Inf Sci, 2015, 58: 052101(12), doi: 10.1007/s11432-015-5313-y

## 1 Introduction

Service-oriented architecture (SOA) has been widely employed by many organizations as a means to improve reusability of software assets and to increase the agility of service-oriented systems. Similar to component-based systems, a service-oriented system is also composed of different software components, which are implemented as Web services. Web services are self-contained, self-describing, and loosely-coupled computational software components designed to support machine-to-machine interaction by programmatic Web method calls. The major difference between traditional component-based systems and service-oriented systems is that service components in the service-oriented systems are usually provided by other organizations, deployed on the Internet, and invoked remotely through the Internet. Examples of service-oriented systems span different application domains, such as e-commerce, e-government, and so on.

In the service-oriented environment, although the Web service technique is mature and widely-used, there is still a serious lack of systematic reliability techniques for building highly reliable service-oriented

* Corresponding author (email: whm_w@163.com)

systems. Software failures can lead to disruption of online services and loss of revenues, and in extreme cases, even lead to severe and fatal consequences in critical service-oriented systems. Consequently, it is a vital challenge to design and implement highly reliable service-oriented systems, due to the following reasons: (1) Service-oriented systems are becoming more and more complex, involving a large number of distributed Web services; (2) the employed Web services are provided by third party companies without source codes and deployed over the Internet; and (3) the Internet environment is highly dynamic, where performance of Web services may change with time. The unpredictable nature of the Internet as well as the compositional and dynamic characteristics of service-oriented systems make it a great challenge in building highly reliable service-oriented systems.

Software reliability engineering is focused on engineering techniques to develop software with high reliability. The major software reliability engineering techniques can be classified under the following areas: fault avoidance, fault removal, fault tolerance, and fault prediction [1]. Fault avoidance technique is the initial defensive mechanism against unreliability. When faults are introduced into the software, fault removal is another protective means. When inherent faults remain undetected through the testing and inspection processes, fault tolerance is the last defence line in preventing software failures. Finally, if software failures are destined to occur, it is critical to estimate and predict them. Since source code and internal design of Web services are unavailable to developers of the service-oriented systems, it is difficult to employ fault avoidance and fault removal techniques to build fault-free service-oriented systems.

In building reliable systems, software fault tolerance comes as the main technique to accomplish the task by masking faults instead of removing faults. Software fault tolerance employs functionally equivalent software components to tolerate faults [2]. Due to the cost of developing and deploying diverse software components, software fault tolerance was employed for only critical systems in the past. In the service-oriented environment, however, it is possible to construct a fault-tolerant service-oriented system efficiently without developing diverse components, since there are already a number of functionally equivalent Web services provided by different organizations on the Internet. Service fault tolerance has become a handy technique in building highly reliable service-oriented systems, which have attracted many research investigations in recent years. This paper provides an overview of the various service fault tolerance techniques based on the diversified Web services on the Internet.

The rest of this paper is organized as follows: Section 2 introduces the design of static and dynamic fault tolerance strategies, and discusses the major problems when designing fault tolerance strategies (i.e., the multithread replication problem and the redundant nested invocation problems); Section 3 proposes several approaches to identify significant components from a complex service-oriented system and investigates the optimal fault tolerance strategy selection; Section 4 provides an overview of various Byzantine fault tolerance approaches for Web services and Section 5 concludes the paper.

## 2 Design of fault tolerance strategies

### 2.1 Basic fault tolerance strategies

To enhance the reliability and trustworthiness of Internet-based software [3], especially service-oriented systems, various service fault tolerance strategies are proposed. The major Fault tolerance (FT) strategies can be divided into passive replication strategies and active replication strategies. Passive strategies [4–6] employ a primary service to process the request and invoke another alternative backup service when the primary service fails, while active strategies [7–11] invoke all functionally equivalent services in parallel. For example, Recovery Block (RB) [12] is a widely-used passive FT strategy, while N-Version Programming (NVP) [13] is a well-known active FT strategy for reliability improvement.

In this section, we introduce some well-accepted basic fault tolerance strategies. Same as the work in [14], we also assume that each user request is independent, and the server-side Web service fails at a fixed rate. Here, we use response time to represent the time duration between sending out a request and receiving a response at the user-side.

- **Retry.** Retry is a very simple and popular fault tolerance strategy when building service-oriented

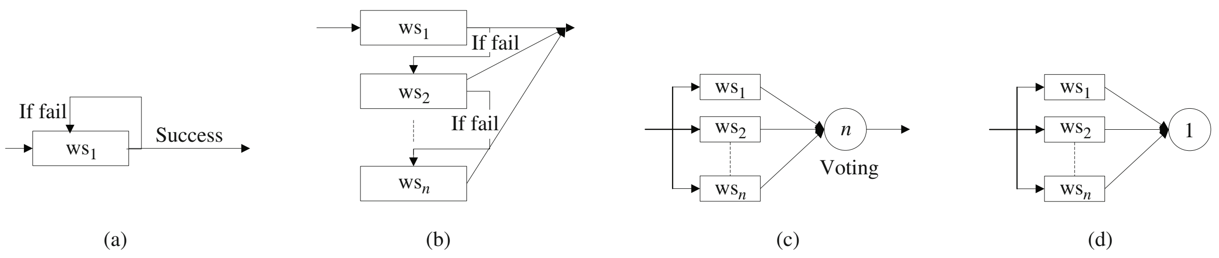

**Figure 1** Basic fault tolerance strategies. (a) Retry; (b) Recovery Block; (c) NVP; (d) Active.

systems. As shown in Figure 1(a), the original Web service will be attempted to be invoked for one or more times when the original service invocation is unsuccessful. Let $m$ denote the number of retries and $f_1$ denote the failure probability of the target Web service. The overall failure probability of the Retry FT strategy (denoted as $f$) can be calculated by

$$f = f_1^m. \tag{1}$$

The overall response time of the Retry FT strategy (denoted as $t$) is

$$t = \sum_{i=1}^{m} t_i (f_1)^{i-1}, \tag{2}$$

where $t_i$ is the response time of the $i$th request.

• **RB.** Recovery Block (RB) [12] is another widely-used FT strategy, which contains a series of alternatives that are to be executed in the listed order. As shown in Figure 1(b), another alternative Web service ws$_2$ will be invoked sequentially if the primary Web service ws$_1$ fails, as usually judged by an acceptance test. The overall failure probability of the RB strategy can be calculated by

$$f = \prod_{i=1}^{m} f_i, \tag{3}$$

where $f_i$ is the failure probability of the $i$th alternative Web service, and $m$ is the number of alternative Web services. The overall response time of the RB strategy can be calculated by

$$t = \sum_{i=1}^{m} t_i \prod_{k=1}^{i-1} f_k, \tag{4}$$

where $t_i$ is the response time of the $i$th alternative Web service.

• **NVP.** N-Version Programming (NVP), also known as multiversion software, is a method to generate multiple functionally equivalent programs from the same initial specifications to improve software reliability [13]. As shown in Figure 1(c), NVP invokes different functionally equivalent Web service replicas at the same time and determines the final result by majority voting. In (5), $n$ (an odd number) represents the number of replicas. $F(i)$ represents the failure probability that $i$ ($i \leqslant n$) replicas fail:

$$f = \sum_{i=n/2+1}^{n} F(i). \tag{5}$$

For example, assuming $n = 3$, then $f = \sum_{i=2}^{3} F(i) = F(2) + F(3) = f_1 \times f_2 \times (1 - f_3) + f_2 \times f_3 \times (1 - f_1) + f_1 \times f_3 \times (1 - f_2) + f_1 \times f_2 \times f_3$. The overall response time performance can be computed by

$$t = \max(t_i), \quad i = 1, 2, \ldots, n, \tag{6}$$

where $t_i$ is the response time of the $i$th Web service replica. Since all the $n$ replicas are invoked in parallel, the overall response time of NVP is the maximum response time value of all replicas.

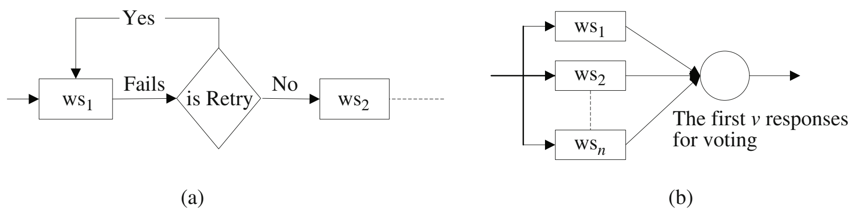

**Figure 2** Dynamic fault tolerance strategies. (a) Dynamic sequential strategy; (b) dynamic parallel strategy.

• **Parallel.** This is another active replication strategy. As shown in Figure 1(d), Parallel strategy invokes different replicas in parallel and takes the first properly-returned response as the final result. Compared to NVP, Parallel strategy can obtain better response time performance. The overall failure probability of the parallel strategy can be calculated by

$$f = \prod_{i=1}^{u} f_i. \tag{7}$$

And the overall response time can be calculated by

$$t = \begin{cases} \min(T_c), & |T_c| > 0, \\ \max(T), & |T_c| = 0, \end{cases} \tag{8}$$

where $T$ is a set of response times of all parallel invoked replicas, $T_c$ is a set of response times of the properly-returned responses, and $u$ is the parallel replica number. If all the invoked replicas fail (i.e., $|T_c| = 0$), the whole strategy fails and the overall response time is equal to the maximum response time among all invoked replicas in parallel (i.e., $t = \max(T)$).

The passive replication strategies (e.g., recovery block) and active replication strategies (e.g., n-version programming) have their own advantages and disadvantages. Passive strategies consume less runtime resources, since they need not invoke multiple alternative Web services at the same time. However, the recovery time is longer when the primary Web service fails, since we need to invoke anther backup Web service and wait for the new response. On the other hand, active replication strategies consume more runtime resources due to multiple Web service invocations, but they recover much faster from failure. Moreover, the recovery block needs to have knowledge of various faults (as captured by the acceptance test) in order to detect failures at runtime, while n-version programming need not have knowledge of faults since it can mask faults by majority voting.

### 2.2 Dynamic fault tolerance strategies

The highly dynamic nature of the Internet makes the above basic fault tolerance strategies ineffective in real-world environment, since they are too homogeneous and static. In the Internet environment, performance of Web services change dynamically. Web service software/hardware may be updated without any notification, and the Internet traffic load and server workload also change from time to time. Moreover, some Web service replicas may become unavailable permanently, while new replicas may join in. These unpredictable characteristics of Web services post a challenge for designing fault tolerance for the service-oriented systems. To attack this critical challenge, Zheng et al. [15] proposed two dynamic fault tolerance strategies, which are more adaptable and can be automatically configured by a QoS-aware middleware in runtime.

• **Dynamic sequential strategy.** As shown in Figure 2(a), the dynamic sequential strategy is a combination of the Retry and RB strategies. When the primary Web service replica fails, the dynamic sequential strategy will consider various related information, such as performance of replicas, user requirements, network conditions, and so on and then determine whether to retry the original replica or another backup replica at runtime.

The overall failure probability of the dynamic sequential fault tolerance strategy can be calculated by

$$f = \prod_{i=1}^{n} f_i^{m_i}, \tag{9}$$

where $m_i$ is the number of retries of the $i$th replica, $n$ is the total number of replicas, and $f_i$ is the failure probability of the $i$th replica. The overall response time of this strategy can be calculated by

$$t = \sum_{i=1}^{n} \left( \left( \sum_{j=1}^{m_i} t_i f_i^{j-1} \right) \prod_{k=1}^{i-1} f_k^{m_i} \right). \tag{10}$$

This strategy equals Recovery Block when $m_i = 1$ for all replicas, and equals Retry when $n = 1$.

• **Dynamic parallel strategy.** As shown in Figure 2(b), the dynamic parallel strategy is the combination of NVP and Active. It will invoke $u$ replica at the same time and employ the first $v$ ($v$ is an odd number, and $v \leqslant u$) properly-returned responses for majority voting. This strategy equals Active when $v = 1$, and equals NVP when $v = u$. The overall failure probability of the dynamic parallel fault tolerance strategy can be calculated by

$$f = \sum_{i=v/2+1}^{v} F(i). \tag{11}$$

As in the same as (5), $F(i)$ represents the failure probability that $i$ ($i \leqslant n$) replicas fail at the same time. The overall response time of this strategy can be calculated by

$$t = \begin{cases} \mathrm{middle}(v, T_c), & |T_c| \geqslant v, \\ \max(T), & |T_c| < v, \end{cases} \tag{12}$$

where $\mathrm{middle}(v, T_c)$ represents the response time of the $v$th properly-returned responses. Since we invoke $u$ replica in parallel and include the first $v$ for voting, the overall response time is equal to the time of the $v$th properly-returned response. If less than $v$ replicas are successful (i.e., $|T_c| < v$), the whole strategy fails and the overall response time is equal to the maximal response time among all the invoked replicas in parallel (i.e., $t = \max(T)$).

When designing fault tolerance strategies, some problems need to be addressed, e.g., the multithread replication problem and the redundant nested invocation problem, which will be discussed in the following subsections.

### 2.3 The multithread replication problem

In an active replication strategy, different functionally equivalent Web service replicas are executed at the same time. These replicas may be implemented as multithreads for improving execution performance. When the tasks in different threads share the same data or impose state dependency, managing the execution order of these tasks in different replicas becomes a great challenge. Different execution order will lead to inconsistent state in different replicas for the same request, which will make the execution of replicas non-deterministic and thus increase complexity of the voting procedure.

To address the problem, Ye et al. [16] proposed a middleware named PWSS to support the client-transparent active replication strategy. PWSS is responsible for maintaining the consistency of the multithreaded Web service replicas. It acts as a thread scheduler and is embedded in every replica. PWSS first divides the threads of a replica into several scheduling groups, where the threads within the same group share common data and must execute in the same order in different replicas for consistency maintenance. PWSS assigns timestamps to the local threads in the same scheduling group based on the thread creation order and announces these timestamps to all other replicas. After receiving all the announcements of the same thread from other replicas, PWSS chooses the largest timestamp as the global timestamp for the thread. In this way, the execution order of the threads in the same group is the same for all replicas, which ensures state consistency on all replicas. PWSS assumes that the replicas are implemented exactly the same and thus have the same thread tree. It uses a multicast protocol to ensure the total order of multicast message, thus the performance of the active replication strategy may be greatly degraded, since each replica needs to wait for timestamps from other replicas for determining the thread execution order.

Osrael et al. [17] proposed a replication middleware for Web services built on the Java-based Axis2 SOAP engine, which can also support active replication management. The middleware uses the primary-backup replication model, when all the backup replicas are "active" and process the requests. The replication model is the same with PWSS but requires weaker multicast primitives (FIFO order instead of total order) than PWWS, since all client requests are forwarded to the primary replica first. Although this replication model can reduce communication cost, the crash of a primary replica requires reconfiguration since a new primary must be elected. Furthermore, synchronous update propagation is used in this middleware, which may also lead to performance degradation.

To reduce the high synchronization cost amongst the replicas caused by group communication primitives, Ye et al. [18] proposed a timestamp based replication (TSR) protocol to maintain the consistency of the replicas' state. The TSR protocol is based on a state machine approach, which allows the replicas to reach an agreement in the order in which client requests are processed. TSR outperforms the group communication primitives when clients rarely send their requests to the system simultaneously. However, guaranting request processing order is not sufficient to solve the multithread problem, if the code executing on each replica is non-deterministic. In work [19], the authors proposed a software transactional memory (STM) based speculative mechanism to efficiently support active replication. This approach allows replicas to optimistically start processing before the order of request is known with certainty. When a transaction finishes processing, it will be validated. If the speculation is wrong, the transaction will be rolled back. By using the transactional mechanisms provided by STM, consistent executions are guaranteed among different replicas. This approach shows its efficiency in stream event processing, while for single service replication with low request load, a non-speculative mechanism may obtain better performance.

In summary, the following two requirements should be taken into consideration when handling the multithread replication problem: (1) the order of the requests received by each replica should be the same; and (2) the code executing each replication should be deterministic. Group communication and timestamp based protocols can be used to ensure the order of requests processed by each replica. Techniques such as thread scheduling, state synchronization and software transactional memory can be used to preserve execution ordering of each replica.

### 2.4 The redundant nested invocation problem

Object management group (OMG) revealed an open issue in its RFP of Fault-tolerant CORBA, i.e., the redundant nested invocations (RNI) problem [20]. A nested invocation refers to the situation where server $A$ needs to invoke server $B$ when processing a user request. When employing an active replication strategy to enhance reliability of server $A$, a group of active replicas of server $A$ are processing the same user request at the same time. The RNI problem arises when the replicas within the same group all make the same nested invocations to another server (e.g., server $B$ in the above example). Redundant invocations will not only increase the workload of server $B$, but also cause inconsistent states when the requests lead to state changes in server $B$.

OMG's RFP of Fault-tolerant CORBA [20] advocated the installation of a suppression mechanism (SM) for redundant nested invocation on active replication groups. When the server is implemented as a single thread and the nested invocations are in deterministic sequence, the RNI problem can be solved by first assigning a sequence number to each invocation. Then the SM will block the redundant nested invocations based on the historical invocation records and the number of the newly arriving invocations. However, when the sever is implemented as multi-threads, the problem becomes more complex since the server invocations have no deterministic sequence.

Narasimhan et al. [21] proposed a solution to address the RNI problem in their fault-tolerant CORBA system by installing a deterministic thread scheduler in the CORBA kernel. This implementation ensures that all replicas produce an identical nested invocation sequence. In this case, modification on thread control in the operating system is required.

Fang et al. [22] addressed the RNI problem for active replication fault tolerant Web services, especially for the multi-threads implementation of the replicas. A thread-tree is built for each replica and a name is assigned for each thread of the replica. Different invocations within the same thread is assigned a sequence number. By employing the thread information (thread name and sequence in the thread) and nested invocation context (target, operation, and arguments), the redundant nested invocations can be suppressed automatically.

When applying active replication strategies for building highly reliable service-oriented systems, the Multithread Replication Problem and the RNI problem should be taken into account carefully.

## 3 Fault tolerance strategy selection

In the field of service fault tolerance, various static as well as dynamic fault tolerances have been designed. How to select the optimal fault tolerance strategy has become an important research problem. Subsection 3.1 identifies the most significant components from a complex service-oriented system and Subsection 3.2 selects the optimal fault tolerance strategy from a set of candidates.

### 3.1 Significant component identification

Service fault tolerance is widely employed to ensure high reliability of service-oriented systems. A service-oriented system typically involves a number of service components, where it is too expensive to provide alternative services for all the service components in the system. Moreover, the failures of different service components impose different impact on the whole system. For example, failures of non-critical components would lead to limited impact on the systems and may not require fault tolerance strategies. To reduce the cost in developing highly reliable service-oriented systems within a limited budget, effective methods are needed to identify significant components from the complex service-oriented systems.

To address this problem, Zheng et al. [23] proposed a PageRank-like component ranking model FTCloud based on the famous PageRank model [24]. FTCloud calculates the significance values of the service components by incorporating the component invocation relationships and frequencies:

$$V(c_i) = \frac{1-d}{n} + d \sum_{k \in N(c_i)} V(c_k) W(e_{ki}), \tag{13}$$

where $N(c_i)$ is a set of service components that invoke the service component $c_i$, $n$ is the number of service components in the service-oriented system, $d$ is a configurable parameter, and $W(e_{ij})$ is the weight value of an invocation relation $e_{ij}$ (representing component $i$ invokes component $j$),which can be calculated by

$$W(e_{ij}) = \frac{\mathrm{frq}_{ij}}{\sum_{j=1}^{n} \mathrm{frq}_{ij}}. \tag{14}$$

By (13), a component $c_i$ inherits a larger significance value if the values of $|N(c_i)|$, $V(c_k)$, and $W(e_{ki})$ are large, indicating that component $c_i$ is invoked by a lot of other significant components frequently. In this way, the service components can be ranked based on their significant values and fault tolerance strategies can be applied to the components ranked at the top.

The FTCloud model was mainly based on the component invocation relationship, and did not take the designers' prior knowledge into consideration (e.g., the system designer may divide the components into critical components and non-critical components). The authors further improved the FTCloud model and propose a hybrid model by taking component characteristics (i.e., critical components or non-critical components) into consideration [25]. Failures of the critical components impose great impact on the system and thus have higher fault tolerance requirement. In the hybrid model, the significance values for critical components are calculated by

$$V(c_i) = (1-d)\frac{\beta}{|C|} + d \sum_{k \in N(c_i)} V(c_k)W(e_{ki}), \tag{15}$$

and the significance values for non-critical components are calculated by

$$V(c_i) = (1-d)\frac{1-\beta}{|\mathrm{NC}|} + d \sum_{k \in N(c_i)} V(c_k)W(e_{ki}), \tag{16}$$

where $|C|$ and $|\mathrm{NC}|$ are the numbers of critical components and non-critical components, respectively, $|C|+|\mathrm{NC}| = n$, and $N(c_i)$ is a set of components that invoke component $c_i$. A parameter $\beta$ ($\frac{|C|}{n} \leqslant \beta \leqslant 1$) is employed to determine how much the hybrid approach relies on the critical components and the non-critical components. When $\beta = \frac{|C|}{n}$, the hybrid approach degrades to the structure-based approach FTCloud, which treats the critical components and the non-critical components equally.

The two previously models are appropriate for newly developed service-oriented systems, where the failure rates of different components do not vary much. However, the failure rates of different components in a legacy application can vary, where the components with significantly high failure rate may impose great impact on application reliability. To address this problem, Qiu et al. [26] proposed another component ranking model, named ROCloud. ROCloud identifies significant components whose failures would have a great impact on application reliability based on the application structure information and components reliability properties. The significance value for a component $c_i$ is computed by

$$V(c_i) = \frac{1-d}{n} f(c_i)p(c_i) + d \sum_{k \in N(c_i)} V(c_k)w_{ki}, \tag{17}$$

where $f(c_i)$ is the component failure rate and $p(c_i)$ is the component failure impact on the application. By (17), the components that are invoked by a lot of other significant frequently components and tend to cause application failures will be identified as significant components.

Although the above models can identify significant components in service-oriented systems to reduce the cost of diversity in service fault tolerance, the analyses are based on the assumption that component failures are independent. Furthermore, none of the models take factors such as data transfer, invocation latency, and so on into consideration. More sophisticated models need to be investigated to get more accurate ranking results in the future.

## 3.2 Optimal fault tolerance strategy selection

After identifying the significant components from a complex service-oriented system, we need to apply suitable fault tolerance strategies to mask the faults of these components. As introduced in Section 2, there are a number of different fault tolerance strategies with different characteristics. How to select the optimal fault tolerance strategies for different components becomes an important research problem.

Given different fault tolerance strategies and their variations, an important research issue is to find out optimal fault tolerance strategies for the service components under various user requirements. User requirements on a single component can be modeled as local constraints, e.g., "response-time of component 1 should be less than 1 second". On the other hand, user requirements on the whole service-oriented system can be modeled as global constraints, e.g., "availability of the service-oriented system should be higher than 99%".

In [27], the optimal fault tolerance strategy selection for a single component is modeled as follows.

**Problem 1.**

$$\text{Minimize: } \sum_{j=1}^{m} u_j z_j,$$

$$\text{Subject to: } \sum_{j=1}^{m} q_j^k z_j \leqslant \mathrm{lc}^k, \quad (k = 1, 2, \ldots, c),$$
$$\sum_{j=1}^{m} z_j = 1,$$
$$z_j \in \{0, 1\}.$$

In Problem 1, $u_j$ is the utility value of the candidate $s_j$ calculated by weighted average of different QoS values of the candidate, $z_j$ is used as an indicator ($z_j = 1$ if the candidate $s_j$ is selected and $z_j = 0$ otherwise), $q_j = (q_j^k)_{k=1}^c$ is the QoS vector of the fault tolerance candidate $s_j$, $m$ is the number of fault tolerance strategy candidates, $\mathrm{lc}^k$ refers to the local constraints specifying user requirements for a single component, and $c$ is the number of local constraints. With this formulation, the user can provide constraints on various QoS properties (e.g., response time, availability, throughput, etc.) of a service component.

To solve Problem 1, we first calculate the QoS values of various fault tolerance strategy candidates. Then the utility values of different candidates are calculated by

$$u_j = \sum_{k=1}^{c} w_k \times q_j^k, \tag{18}$$

where $w_k$ is the weight of the QoS property $q_j^k$. After that, the candidate that meets all the local constraints with the best utility value is selected. By solving Problem 1, optimal fault tolerance strategy can be identified for a component.

Local constraints only specify user requirements on a single service component. It is impractical and time-consuming to provide a number of local constraints on different components. Global constraints, on the other hand, specify user requirements on the whole service-oriented system. Under global constraints, it is not an easy task to determine optimal fault tolerance strategies for service components, since the fault tolerance strategy selection of one component is influenced by the selections of other components. Since each service component in the service-oriented system has a number of fault tolerance strategy candidates, the number of possible combinations increases exponentially with the number of components and fault tolerance candidates. The optimal fault tolerance strategy selection problem under global constraints can eventually be modeled as a 0-1 Integer Programming problem as follows.

**Problem 2.**

$$\text{Minimize: } \sum_{\mathrm{ER}_l \in \mathrm{SP}} \mathrm{freq}_l \times \mathrm{utility}(\mathrm{ER}_l), \tag{19}$$

$$\text{Subject to: } \forall l, \sum_{i \in \mathrm{ER}_l} \sum_{j \in S_i} q_{ij}^k z_{ij} \leqslant \mathrm{gc}^k, \quad (k = 1, 2, 3), \tag{20}$$

$$\forall h, \sum_{i \in \mathrm{SR}_h} \sum_{j \in S_i} q_{ij}^k z_{ij} \leqslant \mathrm{gc}^k, \quad (k = 4), \tag{21}$$

$$\forall l, \prod_{i \in \mathrm{ER}_l} \prod_{j \in S_i} (q_{ij}^k)^{z_{ij}} \leqslant \mathrm{gc}^k, \quad (k = 5), \tag{22}$$

$$\forall i, \sum_{j \in S_i} z_{ij} = 1, \tag{23}$$

$$z_{ij} \in \{0, 1\}. \tag{24}$$

In Problem 2, ER denotes execution route, which includes only one branch in each branch structure, SR is the sequential route, which includes one branch in each branch and parallel structures, and $\mathrm{gc}^k$ is the global constraint for the $k$th QoS property.

Eq. (19) is the objective function, where $\mathrm{freq}_l$ and $\mathrm{utility}(\mathrm{ER}_l)$ are the execution frequency and utility value of the $l$th execution route, respectively. Eq. (19) aims at minimizing the overall utility value of different execution routes. Eqs. (20)–(22) are global constraints on five different QoS properties, i.e., price, popularity, data-size, response time, and availability ($k = 1, 2, 3, 4, 5$, respectively).

In (20), the aggregated QoS values (i.e., price, popularity, and date-size) of an execution route are the sum of QoS values of all components within the route. In (21), the response time of an execution route is equal to the maximal response time of its sequential routes. All sequential routes should meet the global constraints to make sure that every execution route encounters the global constraints. In sequential routes, the aggregated response time values are the sum of response time values of all components within the route. In (22), the aggregated availability value of an execution route is the product of all components within the route. $z_{ij}$ is an indicator. If $z_{ij} = 0$, then $(q_{ij}^k)^{z_{ij}} = 1$, indicating that the candidate is not selected. Eqs. (23) and (24) are employed to ensure that only one fault tolerance candidate will be selected for each component in the service-oriented system. $z_{ij} = 1$ or $z_{ij} = 0$ indicate that a candidate $j$ is selected or not selected for component $i$, respectively.

Since the Integer Programming problem is NP-Complete [28], the well-known Branch-and-Bound algorithm [29] can be employed to reduce the search space. Work in [27] also proposed a heuristic algorithm to further speed up the computation process of the fault tolerance strategy selection.

## 4 Byzantine fault tolerance

A Byzantine fault [30] is an arbitrary fault that may disseminate conflicting information to different parts of the system, which constitutes a serious threat to the system consistency. A Byzantine fault may also choose not to respond. Therefore, crash faults can be considered as a special case of Byzantine faults as well. The traditional fault tolerance strategies (e.g., the passive replication strategies and active replication strategies) are designed to mask crash faults only, while Byzantine fault tolerance (BFT) [31] is a replication technique designed to tolerate Byzantine faults. BFT requires a minimum of $3f + 1$ replicas to tolerate $f$ Byzantine faults [30].

In the service-oriented environment, Web service are deployed on the Internet. The trustworthy Web services may be infected and there may exist some malicious Web services to create Byzantine faults deliberately. Compared with traditional distributed systems, it is more critical to tolerate Byzantine faults in the service-oriented environment. To address this problem, BFT-WS [32], Thema [8], SWS [33], and Perpetual [9] are four different Byzantine fault tolerance strategies, which we will be introduced here.

BFT-WS [32] is a Byzantine fault tolerance framework for Web services. Based on Castro and Liskov's practical BFT algorithm [31], BFT-WS considers client-server application model running in an asynchronous distributed environment with Byzantine faults. $3f + 1$ replications (functionally identical Web services) are employed in the server-side to tolerate $f$ Byzantine faults. BFT-WS is built by extending Sandesha2[1)], which is an implementation of the Web Service Reliable Messaging (WS-RM) standard[2)] for Apache Axis2 in Java.

Thema [8] is a Byzantine Fault-Tolerant (BFT) middleware for Web services. Differing from BFT-WS, Thema supports a three-tiered application model, where the $3f + 1$ Web service replicas in the server-side need to invoke an external Web service for accomplishing their executions. Thema consists of a client library (Thema-C2RS), a BFT service library (Thema-RS), and an external service library (Thema-US). The libraries are deployed to corresponding parties and intended to mask the Byzantine faults automatically to realize user-transparency. Thema extends the BASE [34] BFT system, the Web Service toolkits gSOAP [35], and Apache Axis.

SWS [33] is a survivable Web service framework that supports continuous operation in the presence of general failures and security attacks. SWS applies replication schemes and the N-Modular Redundancy concept. In an m-tier environment, Web services need to invoke other Web services for handling user requests. A Web service plays as server and client at the same time. Each Web service is replicated into a service group to mask faults. SWS employs BFT to design SWS-IGC, an Inter-Group Communication protocol, for reducing the number of messages transferred among multiple Web service groups. It also

1) Apache. Aandesha2. http://ws.apache.org/sandesha/sandesha2, 2008.

2) OASIS. Web services reliable messaging protocol. http://specs.xmlsoap.org/ws/2005/02/rm, 2005.

designs an SWS-MO protocol to guarantee the update consistency of the Web service replicas within a group.

Perpetual [9] is a practical algorithm for tolerating Byzantine faults of deterministic m-tier Web services. It supports interaction between Web service replica groups with different degree of replication. Perpetual enforces strict fault isolation between service replica groups to ensure both safety and liveness in spite of Byzantine faults. It also supports long-running threads of computation as well as asynchronous invocation and processing, which results in improved performance and flexibility over prior protocols.

Group communication is required among the Web service replicas to mask Byzantine faults. Since the Web services are usually distributed across the Internet, too many communications between different Web service replicas may significantly increase the response-time of a Web service invocation. How to enhance response-time performance becomes an important research issue of the Byzantine fault tolerant Web services. When Web service replicas are provided by different organizations, how to enable seamless communication between these heterogeneous replicas becomes another important research issue. Without standardized specification/interface, it is quite difficult to employ Byzantine fault tolerance techniques with high interoperability, which is the nature of Web services.

## 5 Conclusion

With the prevalence of Web services, building highly reliable service-oriented systems has become a critical yet natural need. In this paper, we provided an overview of various service fault tolerance techniques for service-oriented systems, including static and dynamic fault tolerance strategies, the Multithread Replication and Redundant Nested Invocation Problems, significant component identification algorithms, fault tolerance strategy selection algorithms, and some Byzantine fault tolerance approaches.

The highly dynamic and compositional characteristics of Web services post new challenges in building fault-tolerate service-oriented systems. Although the functionally equivalent Web services on the Internet can be used to mask faults, these Web services may subtly differ from each other, since they can be developed and provided by different organizations. These differences may greatly impact the reliability of the service-oriented systems. This problem requires further research investigations. Moreover, network performance will greatly influence the fault tolerance strategies and the service-oriented systems, thus more systematic studies on the network factor need to be conducted in the future.

**Acknowledgements**

This work was supported by National Basic Research Program of China (973 Program) (Grant No. 2011CB3026-03), National Natural Science Foundation of China (Grant Nos. 61332010, 61472338), Guangdong Natural Science Foundation (Grant No. 2014A030313151), and Research Grants Council of the Hong Kong Special Administrative Region, China (Grant No. 415113).

**References**

1 Lyu M R. Handbook of Software Reliability Engineering. New York: McGraw-Hill, 1996
2 Lyu M R. Software Fault Tolerance. Chichester: John Wiley & Sons, 1995
3 Wang H, Tang Y, Yin G, et al. Trustworthiness of internet-based software. Sci China Ser-F: Inf Sci, 2006, 49: 759–773
4 Fang C-L, Liang D, Lin F, et al. Fault-tolerant Web services. J Syst Architect, 2007, 53: 21–38
5 Salatge N, Fabre J-C. Fault tolerance connectors for unreliable Web services. In: Proceedings of 37th International Conference on Dependable Systems and Networks, Edinburgh, 2007. 51–60
6 Sheu G-W, Chang Y-S, Liang D, et al. A fault-tolerant object service on CORBA. In: Proceedings of 17th International Conference on Distributed Computing Systems, Baltimore, 1997. 393
7 Luckow A, Schnor B. Service replication in grids: ensuring consistency in a dynamic, failure-prone environment. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing, Miami, 2008. 1–7
8 Merideth M G, Iyengar A, Mikalsen T, et al. Thema: Byzantine fault-tolerant middleware for Web service applications. In: Proceedings of 24th IEEE Symposium on Reliable Distributed Systems, Orlando, 2005. 131–142

9 Pallemulle S L, Thorvaldsson H D, Goldman K J. Byzantine fault-tolerant Web services for n-tier and service oriented architectures. In: Proceedings of 28th International Conference on Distributed Computing Systems, Beijing, 2008. 260–268

10 Salas J, Perez-Sorrosal F, Marta Pati N-M, et al. WS-replication: a framework for highly available Web services. In: Proceedings of 15th International Conference on World Wide Web, Edinburgh, 2006. 357–366

11 Santos G T, Lung L C, Montez C. FTWeb: a fault tolerant infrastructure for Web services. In: Proceedings of 9th IEEE International Conference on Enterprise Computing, Enschede, 2005. 95–105

12 Randell B, Xu J. The evolution of the recovery block concept. In: Lyu M R, ed. Software Fault Tolerance. Chichester: John Wiley & Sons, 1995. 1–21

13 Avizienis A. The methodology of n-version programming. In: Lyu M R, ed. Software Fault Tolerance. Chichester: John Wiley & Sons, 1995. 23–46

14 Leu D, Bastani F, Leiss E. The effect of statically and dynamically replicated components on system reliability. IEEE Trans Rel, 1990, 39: 209–216

15 Zheng Z, Lyu M R. An adaptive QoS-aware fault tolerance strategy for Web services. Springer J Empir Softw Eng, 2010, 15: 323–345

16 Ye X, Shen Y. Replicating multithreaded web services. In: Proceedings of 3rd International Symposium on Parallel and Distributed Processing and Applications, Nanjing, 2005. 162–167

17 Osrael J, Froihofer L, Weghofer M, et al. Axis2-based replication middleware for Web services. In: Proceedings of IEEE International Conference on Web Services, Salt Lake City, 2007. 591–598

18 Ye X. Providing reliable Web services through active replication. In: Proceedings of 6th IEEE/ACIS International Conference on Computer and Information Science, Melbourne, 2007. 1111–1116

19 Brito A, Fetzer C, Felber P. Multithreading-enabled active replication for event stream processing operators. In: Proceedings of 28th IEEE International Symposium on Reliable Distributed Systems, Niagara Falls, 2009. 22–31

20 Object Management Group. Fault-tolerant COBRA using entity redundancy: request for proposal. 98-04-01, 1998

21 Narasimhan P, Moser L E, Melliar-Smith P M. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In: Proceedings of 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, 1999. 263

22 Fang C-L, Liang D, Chen C, et al. A redundant nested invocation suppression mechanism for active replication fault-tolerant Web service. In: Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service, Taipei, 2004. 9–16

23 Zheng Z, Zhou T C, Lyu M R, et al. FTCloud: a ranking-based framework for fault tolerant cloud applications. In: Proceedings of International Symposium on Software Reliability Engineering, San Jose, 2010. 398–407

24 Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine. In: Proceedings of 7th Internationl World Wide Web Conference, Brisbane, 1998

25 Zheng Z, Zhou T, Lyu M R, et al. Component ranking for fault-tolerant cloud applications. IEEE Trans Serv Comput, 2012, 5: 540–550

26 Qiu W, Zheng Z, Wang X, et al. Reliability-based design optimization for cloud migration. IEEE Trans Serv Comput, 2014, 7: 223–236

27 Zheng Z, Lyu M R. Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. IEEE Trans Comput, 2015, 64: 219–232

28 Cormen T, Leiserson C, Rivest R. Introduction to Algorithms. Cambridge: MIT Press, 1990

29 Shahadat Khan E G M, Li Kin F, Akbar M. Solving the knapsack problem for adaptive multimedia systems. Stud Inf Univ, 2002, 2: 157–178

30 Lamport L, Shostak R, Pease M. The Byzantine generals problem. ACM Trans Program Lang Syst, 1982, 4: 382–401

31 Castro M, Liskov B. Practical Byzantine fault tolerance. In: Proceedings of 3rd Symposium on Operating Systems Design and Implementation, New Orleans, 1999. 1–14

32 Zhao W. BFT-WS: a Byzantine fault tolerance framework for Web services. In: Proceedings of 7th International IEEE EDOC Conference Workshop, Annapolis, 2007. 89–96

33 Li W, He J, Ma Q, et al. A framework to support survivable Web services. In: Proceedings of 19th IEEE International Symposium on Parallel and Distributed Processing, Denver, 2005. 93–94

34 Rodrigues R, Castro M, Liskov B. BASE: using abstraction to improve fault tolerance. In: Proceedings of 18th Symposium on Operating Systems Principles, Banff, 2001. 15–28

35 Engelen R A V, Gallivan K A. The gSOAP toolkit for Web services and peer-to-peer computing networks. In: Proceedings of IEEE International Symposium on Cluster Computing and the Grid, Berlin, 2002. 128