Empir Software Eng (2010) 15:323–345
DOI 10.1007/s10664-009-9126-8

An adaptive QoS-aware fault tolerance strategy for web services

Zibin Zheng · Michael R. Lyu

Published online: 10 December 2009

Editor: Laurie Williams

Abstract Service-Oriented Architecture (SOA) is widely adopted for building mission-critical systems, ranging from on-line stores to complex airline management systems. How to build reliable SOA systems becomes a big challenge due to the compositional nature of Web services. This paper proposes an adaptive QoS-aware fault tolerance strategy for Web services. Based on a user-collaborated QoS-aware middleware, SOA systems can dynamically adjust their optimal fault tolerance configurations to achieve optimal service reliability as well as good overall performance. Both the subjective user requirements and the objective system performance of the Web services are considered in our adaptive fault tolerance strategy. Experiments are conducted to illustrate the advantages of the proposed adaptive fault tolerance strategy. Performance and effectiveness comparisons of the proposed adaptive fault tolerance strategy and various traditional fault tolerance strategies are also provided.

Keywords Web service · Fault tolerance · QoS · Middleware

1 Introduction

Web services are self-contained, self-describing, and loosely-coupled computational components designed to support machine to machine interaction via networks. Web services are usually distributed across the Internet and invoked by the service-oriented applications (SOA) via communication links. By providing a standardized

Z. Zheng (✉) · M. R. Lyu
Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Hong Kong, China
e-mail: zbzheng@cse.cuhk.edu.hk

M. R. Lyu
e-mail: lyu@cse.cuhk.edu.hk

XML-based interface (WSDL) and communication message descriptions (SOAP), Web services provide unprecedented opportunities for building agile and versatile applications by integrating existing Web services offered by various service providers.

The compositional nature of Web services and the unpredictable nature of Internet pose a new challenge for building reliable SOA systems, which are widely employed in critical domains such as e-commerce and e-government. In contrast to traditional stand-alone systems, an SOA system may break down due to: 1) the errors of the SOA system itself, 2) Internet errors (e.g., connect break off, packet loss, etc.), and 3) remote Web service problems (e.g., too many users, crashes of the Web services, etc.).

There are four technical areas to build reliable software systems, which are fault prevention (Lyu 1996), fault removal (Zheng et al. 2009a), fault tolerance (Lyu 1995), and fault forecasting (Lyu 1996). Since it is difficult to completely remove software faults, software fault tolerance (Lyu 1995) is an essential approach to building highly reliable systems. Critics of software fault tolerance state that developing redundant software components for tolerating faults is too expensive and the reliability improvement is questionable when comparing to a single system, considering all the overheads in developing multiple redundant components. In the modern era of service-oriented computing, however, the cost of developing multiple component versions is greatly reduced. This is because the functionally equivalent Web services designed/developed independently by different organizations can be readily employed as redundant alternative components for building diversity-based fault tolerant systems.

A number of fault tolerance strategies for Web services have been proposed in the recent literature (Chan et al. 2007; Foster et al. 2003; Tsai et al. 2003; Zheng and Lyu 2008a). However, most of these strategies are not feasible enough to be applied to various systems with different performance requirements, especially the service-oriented Internet systems in the highly dynamic environment. There is an urgent need for more general and "smarter" fault tolerance strategies, which are context-aware and can be dynamically and automatically reconfigured for meeting different user requirements and changing environments. Gaining inspiration from the *user-participation* and *user-collaboration* concepts of Web 2.0, we design an adaptive fault tolerance strategy and propose a user-collaborated QoS-aware middleware in making fault tolerance for SOA systems efficient, effective and optimal.

This paper is a comprehensive version of our previous work (Zheng and Lyu 2008b). Our work aims at advancing the current state-of-the-art of fault tolerance in the field of service reliability engineering. The contributions of this paper are two-fold: (1) A QoS-aware middleware for achieving fault tolerance by employing user-participation and collaboration. By encouraging users to contribute their individually-obtained QoS information of the target Web services, more accurate evaluation on the Web services can be achieved; (2) an adaptive fault tolerance strategy. We propose an adaptive fault tolerance strategy for automatic system reconfiguration at runtime based on the subject user requirements and objective QoS information of the target Web services.

The rest of this paper is organized as follows: Section 2 introduces the QoS-aware middleware design and some basic concepts. Section 3 designs models for user requirements and QoS. Section 4 presents various fault tolerance strategies.

Section 5 proposes the adaptive fault tolerance strategy. Section 6 presents a number of experiments. Section 7 reviews related work and Section 8 concludes the paper.

2 Design of the QoS-aware Middleware

In this section, some basic concepts are explained and the architecture of our QoS-aware middleware for fault tolerant Web services is presented.

2.1 Basic Concept Introduction

We divide faults into two types based on the cause of the faults:

- **Network faults.** Network faults are generic to all Web services. For example, *Communication Timeout, Service Unavailable (http 503), Bad Gateway (http 502), Server Error (http 500)*, and so on, are network faults. Network faults can be easily identified by the middleware.
- **Logic faults.** Logic faults are specific to different Web services. For example, calculation faults, data faults, and so on, are logic faults. Also, various exceptions thrown by the Web service to the service users are classified into the logic-related faults. It is difficult for the middleware to identify such type of faults.

In this paper, *atomic services* present Web services which provide particular services to users independently. *Atomic services* are self-contained and do not rely on any other Web services. On the other hand, *composite services* presents Web services which provide services to users by integrating and calling other Web services (Benatallah et al. 2002; Zeng et al. 2004).

With the popularization of service-oriented computing, various Web services are continuously emerging. The functionalities and interfaces defined by the Web Service Description Language (WSDL) are becoming more and more complex. Machine learning techniques (Salatge and Fabre 2007; Wu and Wu 2005) are proposed to identify Web services with similar or identical functionalities automatically. However, the effect and accuracy of these approaches are still far from practical usage. Since functionally equivalent Web services, which are developed independently by different organizations, may appear with completely different function names, input parameters and return types, it is really difficult for machines to know that these services are actually providing the same functionalities.

To solve the problem of identical/similar Web services identification, a service community defines a common terminology that is followed by all participants, so that the Web services, which are developed by different organizations, can be described in the same interface (Benatallah et al. 2002; Zeng et al. 2004). Following a common terminology, automatical Web service composition by programs can be achieved, which will attract more users and make the development of the community better.

Companies can enhance their business benefit by joining into communities, since a lot of service users will go to the communities to search for suitable services. The coordinator of the community maintains a list of the registered Web services of the community. Before joining the community, a Web service has to follow the interface definition requirements of the community and registers with the community

coordinator. By this way, the service community makes sure that various Web services from different organizations in the community come with the same interface.

In this paper, we focus on engaging the Web services in the service communities for fault tolerance and performance enhancement purposes. The design and development of the service communities, which have been discussed in Zeng et al. (2004), are out of our scope. We use the word *replica* to represent the functionally equivalent Web services within the same service community.

2.2 Architecture of the Middleware

The architecture of the proposed QoS-aware middleware for fault tolerant Web services is presented in Fig. 1. The work procedure of this middleware is described as follows:

(1). From the Universal Description, Discovery and Integration (UDDI), the middleware obtains the addresses of the service community coordinators.
(2). By contacting the community coordinator, the middleware obtains an address list of the replicas in the community and the overall QoS information of these replicas. The overall QoS information will be used as the initial values in the middleware for optimal fault tolerance strategy configuration. Detailed design of the QoS-model of Web services will be introduced in Section 4.2.
(3). The proposed QoS-aware middleware determines the optimal fault tolerance strategy dynamically based on the user QoS requirements and the QoS information of the target replicas.

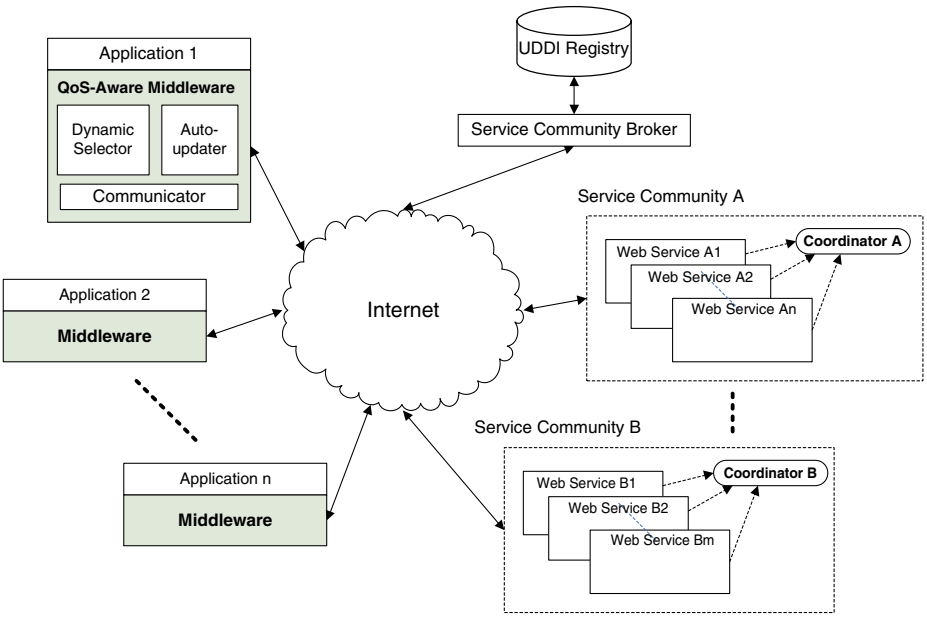


Fig. 1 Architecture of the middleware

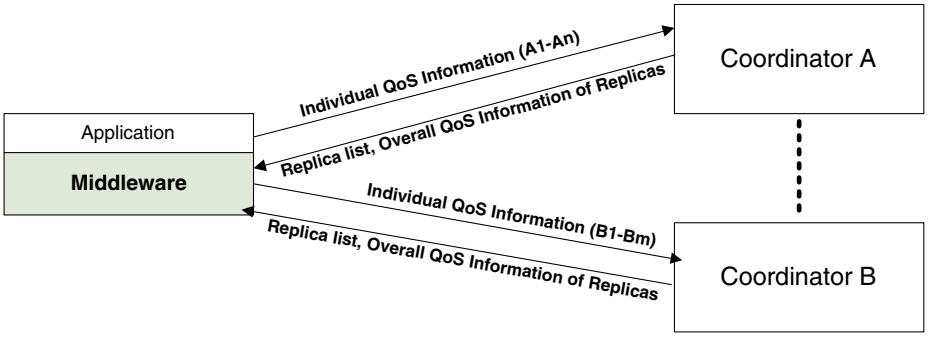


Fig. 2 Interaction between the middleware and the coordinators

(4). The middleware invokes certain replicas with the optimal fault tolerance strategy and records down the QoS performance of the invoked replicas.
(5). The middleware dynamically adjusts the optimal fault tolerance strategy based on the overall QoS information and the individually recorded QoS information of the replicas.
(6). As shown in Fig. 2, in order to obtain the most up-to-date QoS information of the target replicas for better optimal fault tolerance strategy determination, the middleware will send its individually obtained replica QoS information to the community coordinators in exchange for the newest overall replica QoS information from time to time. By the design of this QoS information exchange mechanism, the community coordinator can obtain replica QoS information from various service users in different geographical locations, and use it for providing the overall replica QoS information to the service users.

As shown in Fig. 1, the middleware includes the following three parts:

- **Dynamic selector:** in charge of determining the optimal fault tolerance strategy, based on user requirements and the QoS information of replicas dynamically.
- **Auto updater:** updating the newest overall replica QoS information from the community coordinator and providing the obtained QoS information to the coordinator. This mechanism promotes user collaboration to achieve more accurate optimal fault tolerance strategy selection.
- **Communicator:** in charge of invoking certain replicas with the optimal fault tolerance strategy.

3 Basic Fault Tolerance Strategies

When applying Web services to critical domains, reliability becomes a major issue. With the popularization of Web services, more and more functionally equivalent Web services are diversely designed and developed by different organizations, making software fault tolerance an attractive choice for service reliability improvement.

There are two major types of fault tolerance strategies: sequential and parallel. *Retry* (Chan et al. 2007) and *Recovery Block (RB)* (Randell and Xu 1995) are two major sequential approaches that employ time redundancy to obtain higher

reliability. On the other hand, *N-Version Programming (NVP)* (Avizienis 1995) and *Active* (Salatge and Fabre 2007) strategies are two major parallel strategies that engage space/resource redundancy for reliability improvement.

In the following, we provide detailed introductions and formula of response time and failure-rate for these basic fault tolerance strategies. As discussed in the work (Leu et al. 1990), we assume that each request is independent, and the Web service fails at a fix rate. Here, we use RTT (Round-Trip-Time) to represent the time duration between sending out a request and receiving a response of a service user.

- **Retry:** As shown in Fig. 3(1), the original Web service will be retried for a certain number of times when it fails. Equation (1) is the formula for calculating failure-rate f and RTT t, where m is the number of retries, f_1 is the failure-rate of the target Web service, and t_i is the RTT of the i^{th} request.

$$f = f_1^m; \qquad t = \sum_{i=1}^{m} t_i (f_1)^{i-1} \tag{1}$$

- **RB:** As shown in Fig. 3(2), another standby Web service ($A2$) will be tried sequentially if the primary Web service fails.

$$f = \prod_{i=1}^{m} f_i; \qquad t = \sum_{i=1}^{m} t_i \prod_{k=1}^{i-1} f_k \tag{2}$$

- **NVP:** As shown in Fig. 3(3), NVP invokes different replicas at the same time and determines the final result by majority voting. It is usually employed to mask logical faults. In (3), n, which is an odd number, represents the total replica number. $F(i)$ represents the failure-rate that i ($i \leq n$) replicas fail. For example, assuming $n = 3$, then $f = \sum_{i=2}^{3} F(i) = F(2) + F(3) = f_1 \times f_2 \times (1 - f_3) + f_2 \times f_3 \times (1 - f_1) + f_1 \times f_3 \times (1 - f_2) + f_1 \times f_2 \times f_3$.

$$f = \sum_{i=n/2+1}^{n} F(i); \qquad t = \max(\{t_i\}_{i=1}^{n}) \tag{3}$$

- **Active:** As shown in Fig. 3(4), *Active* strategy invokes different replicas in parallel and takes the first properly-returned response as the final result. It is usually employed to mask network faults and to obtain better response time

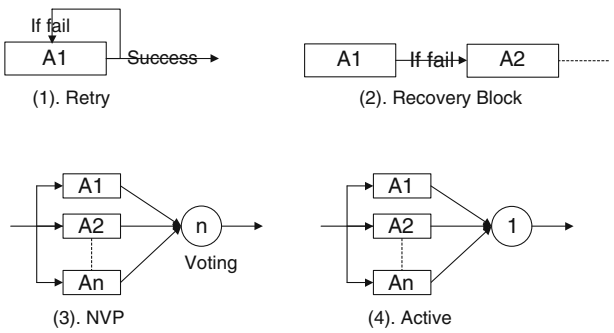


Fig. 3 Basic fault tolerance strategies

performance. In (4), T_c is a set of RTTs of the properly-returned responses. u is the parallel replica number.

$$f = \prod_{i=1}^{u} f_i; t = \begin{cases} \min(T_c) : |T_c| > 0 \\ \max(T) : |T_c| = 0 \end{cases} \tag{4}$$

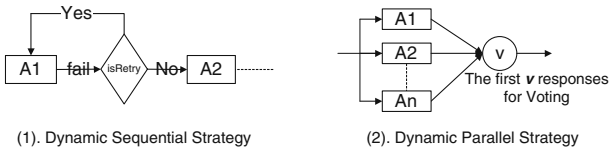
The highly dynamic nature of Internet and the compositional nature of Web services make the above *static* fault tolerance strategies unpractical in real-world environment. For example, some replicas may become unavailable permanently, while some new replicas may join in. Moreover, Web service software/hardware may be updated without any notification, and the Internet traffic load and server workload are also changing from time to time. These unpredictable characteristics of Web services provide a challenge for optimal fault tolerance strategy determination. To attack this critical challenge, we propose the following two dynamic fault tolerance strategies, which are more adaptable and can be automatically configured by a QoS-aware middleware in runtime. These two dynamic strategies will be employed in our dynamic fault tolerance strategy selection algorithm in Section 5.3.

– **Dynamic Sequential Strategy:** As shown in Fig. 4(1), the dynamic sequential strategy is the combination of *Retry* and *RB* strategies. When the primary replica fails, our algorithm will dynamically determine whether to employ *Retry* or *RB* at runtime based on the QoS of the target replicas and the requirements of service users. The determination algorithm will be introduced in Section 5.3. In (5), m_i is the number of retries of the i^{th} replica, and n is the total replica quantity. This strategy equals RB when $m_i = 1$, and equals *Retry* when $m_1 = \infty$.

$$f = \prod_{i=1}^{n} f_i^{m_i}; t = \sum_{i=1}^{n} \left(\left(\sum_{j=1}^{m_i} t_i f_i^{j-1} \right) \prod_{k=1}^{i-1} f_k^{m_i} \right) \tag{5}$$

– **Dynamic Parallel Strategy:** As shown in Fig. 4(2), 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 $v \leq u$) properly-returned responses for majority voting. This strategy equals *Active* when $v = 1$, and equals *NVP* when $v = u$. Note $middle(v, T_c)$ is employed to calculate the RTT of invoking u replica in parallel and includes the first v for voting, which is equal to the RTT of the v^{th} properly-returned response.

$$f = \sum_{i=v/2+1}^{v} F(i); t = \begin{cases} middle(v, T_c) : |T_c| \geq v \\ \max(T) : |T_c| < v \end{cases} \tag{6}$$



(1). Dynamic Sequential Strategy (2). Dynamic Parallel Strategy

Fig. 4 Dynamic fault tolerance strategies

4 Models of User Requirements and QoS

4.1 User Requirement Model

Optimal fault tolerance strategies for SOA systems vary from case to case, which are influenced not only by the QoS of the target replicas, but also by the characteristics of the SOA systems. For example, realtime systems may prefer parallel strategies for better response time performance, while resource-constrained systems (e.g., mobile applications) may prefer sequential strategies for better resource conservation.

It is usually difficult for a middleware to automatically detect the characteristics of an SOA system, such as whether it is latency-sensitive or resource-constrained. The strategy selection accuracy will be greatly enhanced if the service users can provide some concrete requirements/constraints. However, it is impractical and not user-friendly to require the service users, who are often not familiar with fault tolerance strategies, to provide detailed technical information. To address this problem, we design a simple user requirement model for obtaining necessary requirement information from the users. In this model, the users are required to provide the following four values:

1. t_{max}: the largest RTT that the application can afford. t_{max} with a smaller value means higher requirement on response time, indicating that the application is more latency-sensitive. If the response-time of a Web service invocation is larger than t_{max}, the invocation is regarded as *TimeOut* failure to the service user.
2. f_{max}: the largest failure-rate that the application can afford. If the failure-rate of a Web service is larger than f_{max}, it is not suitable to be employed without fault tolerance strategies.
3. r_{max}: the largest resource consumption constraint. The amount of parallel connection is used to approximately quantify the resource consumption, since connecting more Web services in parallel will consume more computing and networking resources. r_{max} with a smaller value indicates that the application is resource-constraint.
4. *mode*: the *mode* can be set by the service users to be *sequential*, *parallel*, or *auto*. *Sequential* means invoking the replicas sequentially (e.g., for the payment-oriented Web services). *Parallel* means that the user prefers invoking the target replicas in parallel. *Auto* means that the users let the middleware determine the optimal mode automatically. We need the service users to provide this *mode* information, because the middleware may not be *smart* enough to detect whether the target replicas are payment-oriented services or not.

The user requirements obtained by this model will be used in our dynamic fault tolerance strategy selection algorithm in Section 5.3.

4.2 QoS Model of Web Service

In addition to the subjective user requirements, the objective QoS information of the target Web service replicas are also needed for the optimal fault tolerance strategy determination. A lot of previous tasks are focused on building the QoS model

for Web services (Deora et al. 2003; Maximilien and Singh 2002; Wu et al. 2007). However, there are still several challenges to be solved:

- **It is difficult to obtain performance information of the target Web services.** Service users do not always record the QoS information of the target replicas, such as RTT, failure-rate and so on. Also, most of the service users are unwilling to share the QoS information they obtain.
- **Distributed geographical locations of users make evaluation on target Web services difficult.** Web service performance is influenced by the communication links, which may cause performance evaluation results provided by one user to be inapplicable to others. For example, a user located in the same local area network (LAN) with the target Web service is more likely to yield good performance. The optimistic evaluation result provided by this user may misguide other users who are not in the same LAN as the target Web service.
- **Lack of a convenient mechanism for service users to obtain QoS information of Web services.** QoS information can help service users be aware of the quality of a certain Web service and determine whether to use it or not. However, in reality, it is very difficult for the service users to obtain accurate and objective QoS information of the Web services.

To address the above challenges, we design a QoS model for Web services employing the concept of user-participation and user-collaboration, which is the key innovation of Web2.0. The basic idea is: by encouraging users to contribute their individually obtained QoS information of the target replicas, we can collect a lot of QoS data from the users located in different geographical locations under various network conditions, and engage these data to make the objective overall evaluation on the target Web services.

Based on the concept of service community and the architecture shown in Fig. 1, we use the community coordinator to store the overall QoS information of the replicas. Users will periodically send their individually-obtained replica QoS information to the service community in exchange for the the newest overall replica QoS information, which can be engaged for better optimal strategy determination. Since the middleware will record QoS data of the replicas and exchange it with the coordinator automatically, updated replica QoS information is conveniently available for service users.

For a single replica, the community coordinator will store the following information:

- t_{avg} : the average RTT of the target replica.
- t_{std} : the standard deviation of RTT of the target replica.
- fl : the logic failure-rate of the target replica.
- fn : the network failure-rate of the target replica.

Currently, we only consider the most important QoS properties in our QoS model, which includes RTT, logic faults, network faults and resource consumption. Other QoS properties, however, can be easily included in the future. For those users who are not willing to exchange QoS data with the community coordinator, they can simply close the exchange functionality of the middleware, although this will reduce the dynamic optimal strategy selection performance. This is similar to

BitTorrent (Bram 2003) download, where stopping uploading files to others will hurt the download speed of the user.

5 Adaptive Fault Tolerance Strategy Configuration

5.1 Notations

The notations used in this paper are listed as follows:

- $\{ws_i\}_{i=1}^{n}$: a set of functionally equivalent replicas.
- $\{c_{ij}\}_{j=1}^{k+2}$: a set of (k+2) counters for the ws_i.
- $\{p_{ij}\}_{j=1}^{k+2}$: the probability of an RTT belonging to different categories for ws_i.
- $\{t_i\}_{i=1}^{k}$: a set of time values, where t_i is the presentative time of the i^{th} time-slot. $t_i = \frac{t_{max} \times (i-0.5)}{k}$
- $RTT_v = \{rtt_j\}_{j=1}^{v}$: a set of RTT values of the v replicas.

5.2 Scalable RTT Prediction

Accurate RTT prediction is important for the optimal fault tolerance strategy selection. Assuming, for example, that there are totally n replicas $\{ws\}_{i=1}^{n}$ in the service community. We would like to invoke v ($v \leq n$) replicas in parallel and use the first properly-returned response as the final result. The question is, then, how to find out the optimal set of replicas that will achieve the best RTT performance?

To solve this problem, we need the RTT distributions of all the replicas. In our previous work (Zheng and Lyu 2008a), all the historical RTT results are stored and employed for RTT performance prediction. However, sometimes it is impractical to require the users to store all the past RTT results, which are ever growing and will consume a lot of storage memory. On the other hand, without historical RTT performance information of the replicas, it is extremely difficult to make an accurate prediction. To address this challenge, we propose a scalable RTT prediction algorithm, which scatters the RTT distributions of a replica to reduce the required data storage.

We divide the user required maximum response-time t_{max}, which is provided by the service user, into k time slots. Instead of storing all the detailed historical RTT results, the service user only needs to store $k+2$ distribution counters $\{c_i\}_{i=1}^{k+2}$ for each replica, where c_1–c_k are used to record the numbers of the Web service invocations which fit into the corresponding time slots, c_{k+1} is used to record network-related faults fn, and c_{k+2} is for recording logic-related faults fl. By describing the RTT distribution information by these counters, (7) can be employed to predict the probability that a future Web service invocation belonging to a category, where p_1 to p_k are the probabilities that the invocation will fit into the corresponding time-slots, p_{k+1} is the probabilities that a Web service invocation will fail due to network-related

faults, and p_{k+2} is the probability that an invocation will fail due to logic-related faults.

$$p_i = \frac{c_i}{\sum_{i=1}^{k+2} c_i} \tag{7}$$

By the above design, we can obtain approximate RTT distribution information of a replica by storing only $k+2$ counters. The values of time-slot number k can be set to be a larger value for obtaining more detailed distribution information, making this algorithm scalable.

The approximate RTT distributions of the replicas, which are obtained by the above approach, can be engaged to predict RTT performance of a particular set of replicas $\{ws_i\}_{i=1}^{v}$. We use $rtt_i == t_j$ to present that an RTT value belongs to the j^{th} time-slot. Assuming that the RTT values of future invocations of the selected v replicas are $RTT_v = \{rtt_i\}_{i=1}^{v}$. The probability that rtt_i fits into a certain time-slot t_j ($rtt_i == t_j$) is provided by p_{ij}. For *Active* strategy, the problem of predicting RTT performance of invoking a set of replicas at the same time can be formulated as (8), where $rtt_x = min\{RTT_v\}$ and $RTT_v = \{rtt_i\}_{i=1}^{v}$.

$$\tilde{r}tt = \sum_{i=1}^{k} (p(rtt_x == t_i) \times t_i); \tag{8}$$

Equation (9) is employed for calculating the value of $p(rtt_x == t_i)$, which is needed in (8).

$$p(rtt_x == t_i) = p(rtt_x \leq t_i) - p(rtt_x \leq t_{i-1}); \tag{9}$$

Therefore, the RTT prediction problem becomes calculating the values of $p(rtt_x \leq t_i)$. Equation (10) is employed for calculating the value of $p(rtt_x \leq t_i)$, where $p(rtt_v \leq t_i)$ is the probability that the RTT value rtt_v of the last Web service ws_v is smaller than t_i, which can be calculated by $p(rtt_v \leq t_i) = \sum_{k=1}^{i} p_{vk}$. If rtt_v is smaller than t_i, then $rtt_x = \min(RTT_v)$ will be smaller than t_i; otherwise, the remaining Web services ws_i–ws_{v-1} will be calculated by the same procedure recursively.

$$p(rtt_x \leq t_i) = p(rtt_v \leq t_i) + p(rtt_v > t_i) \times p(\min(RTT_{v-1}) \leq t_i); \tag{10}$$

By the above calculation, the RTT performance of the *Active* strategy, which invokes the given replicas in parallel and employs the first returned response as final result, can be predicted. By changing the $rtt_x = \min(RTT_v)$ to $rtt_x = \max(RTT_v)$, the above calculation procedure can be used to predict the RTT performance of the NVP strategy, which needs to wait for all responses of replicas before the majority voting. By changing the $\min(RTT_v)$ to $middle(RTT_v, y)$, which means the RTT value of the y^{th} returned response, the above algorithm can be used to predict the RTT performance of the *Dynamic parallel strategy*. For example, in the *Dynamic parallel strategy*, if we invoke 6 replicas in parallel and employ the first 3 returned responses for voting, then the RTT performance of the whole strategy is equal to the RTT of the 3^{rd} returned response.

Therefore, to solve the problem proposed in the beginning of this section, we can predict the RTT performance of different replica sets with v replicas from all the n replicas $\{ws\}_{i=1}^{n}$ and select the set with the best RTT performance.

5.3 A Dynamic Fault Tolerance Strategy Selection Algorithm

By employing and integrating the user requirement model designed in Section 4.1, the QoS model of Web services designed in Section 4.2, and the RTT prediction algorithm designed in Section 5.2, we propose a dynamic fault tolerance strategy selection algorithm in this section. As shown in Algorithm 1, the whole selection procedure is composed of three parts: sequential or parallel strategies determination, dynamic sequential strategy determination, and dynamic parallel strategy determination. The detailed descriptions of these three sub-components are presented in the following sections.

Algorithm 1 The Optimal Fault Tolerance Strategy Determination Algorithm

```
   Data: t_max, f_max, r_max, QoS of the replicas
   Result: Optimal fault tolerance strategy
 1 Sequential or parallel strategy determination;
 2 if sequential then
 3 |   d = 1/m × ((t_{i+1} − t_i)/t_max + (f_{i+1} − f_i)/f_max);
 4 |   if d > e then
 5 |   |   Retry;
 6 |   else
 7 |   |   RB (try another replica);
 8 |   end
 9 else
10 |   calculate performance of the parallel strategies with different v values;
11 |   select the strategy with minimize s_i value as optimal strategy;
12 end
13 return optimal fault tolerance strategy;
```

5.3.1 Sequential or Parallel Strategy Determination

If the value of the attribute *mode* in the user requirement model equals to *auto*, we need to conduct sequential or parallel strategy determination based on the QoS performance of the target replicas and the subjective requirements of the users. Equation (11) is used to calculate the performance of different strategies, where w_1–w_3 are the user defined weights for different QoS properties.

$$s_i = w_1 \frac{t_i}{t_{max}} + w_2 \frac{f_i}{f_{max}} + w_3 \frac{r_i}{r_{max}}; \tag{11}$$

The underlying consideration is that the performance of a particular response time is related to the user requirement. For example, 100 *ms* is a large latency for the latency-sensitive applications, while it may be negligible for non-latency-sensitive applications. By using $\frac{t_i}{t_{max}}$, where t_{max} represents the user requirement on response time, we can have a better representation of the response time performance for service users with different requirements. Failure-rate f_i and resource consumption r_i are similarly considered.

By employing (11), the performance of sequential strategies and parallel strategies can be computed and compared. For sequential strategies, the value of t_i can be calculated by (5), where the value of f_i can be obtained from the middleware and the

value of r_i is 1 (only one replica is invoked at the same time). For parallel strategies, the value of t_i can be estimated by using the RTT prediction algorithm presented in Section 5.2, where the value of f_i can be obtained from the middleware, and the value of r_i is the number of parallel invocation replicas. From the sequential and parallel strategies, the one with smaller s_i value will be selected.

5.3.2 Dynamic Sequential Strategy Determination

If the value of the attribute *mode* provided by the service user is equal to *sequential*, or the sequential strategy is selected by the above selection procedure conducted by the middleware, we need to determine the detailed sequential strategy dynamically based on the user requirements and the QoS values of replicas.

$d = \frac{1}{m} \times (\frac{t_{i+1}-t_i}{t_{max}} + \frac{f_{i+1}-f_i}{f_{max}})$ is used to calculate the performance difference between two replicas, where $\frac{1}{m}$ is a degradation factor for the *Retry* strategy and m is the retried times. When $d > e$, where e is the performance degradation threshold, the performance difference between the two selected replicas is large, therefore, retrying the original replica is more likely to obtain better performance. By increasing the number of retries m, d will become smaller and smaller, reducing the priority of *Retry* strategy and raising the probability that RB will be selected.

If the primary replica fails, the above procedure will be repeated until either a success or the time expires ($RTT \geq t_{max}$).

5.3.3 Dynamic Parallel Strategy Determination

If the value of the attribute *mode* provided by the service user is equal to *parallel*, or the parallel strategy is selected by the middleware, we need to determine the optimal parallel replica number n and the NVP number v ($v \leq n$) for the dynamic parallel strategy.

By employing the RTT prediction algorithm presented in Section 5.2, we can predict the RTT performance of various combinations of the value v and n. The number of all combinations can be calculated by $C_n^v = \frac{n!}{v! \times (n-v)!}$, and the failure-rate can be calculated with (6). By employing (11), the performance of different n and v combination can be calculated and compared. The combination with the minimal p value will be selected and employed as the optimal strategy.

6 Experiments

A series of experiments is designed and performed for illustrating the QoS-aware middleware and the dynamic fault tolerance selection algorithm. In the experiments, we compare the performance of our dynamic fault tolerance strategy (denoted as *Dynamic*) with other four traditional fault tolerance strategies *Retry*, RB, NVP, and *Active*.

6.1 Experimental Setup

Our experimental system is implemented and deployed with JDK6.0, Eclipse3.3, Axis2.0 (Apache 2008), and Tomcat6.0. We develop six Web services following an identical interface to simulate replicas in a service community. These replicas

Table 1 Requirements of service users

Users	t_{max}	f_{max}	r_{max}	Focus
User 1	1000	0.1	50	RTT
User 2	2000	0.01	20	RTT, Fail
User 3	4000	0.03	2	RTT, Fail, Res
User 4	10000	0.02	1	Res
User 5	15000	0.005	3	Fail, Res
User 6	20000	0.0001	80	Fail

are employed for evaluating the performance of various fault tolerance strategies under different situations. The service community coordinator is implemented by *Java Servlet*. The six Web services and the community coordinator are deployed on seven PCs. All PCs have the same configuration: Pentium(R) 4 CPU 2.8 GHz, 1G RAM, 100Mbits/sec Ethernet card and a Windows XP operating system. In the experiments, we simulate network-related faults and logic-related faults. All the faults are further divided into permanent faults (service is down permanently) and temporary faults (faults occur randomly). The fault injection techniques are similar to the ones proposed in Looker and Xu (2003), Vieira et al. (2007).

In our experimental system, service users, who will invoke the six Web service replicas, are implemented as *Java applications*. We first provide six service users with representative requirement settings as typical examples for investigating performance of different fault tolerance strategies in different situations. The detailed user requirements are shown in Table 1. We then study the influence of parameters of the user requirements and report the experimental results.

In the experiments, failures are counted when service users cannot get a proper response. For each service request, if the response time is larger than t_{max}, a *timeout* failure is counted.

Our experimental environment is defined by a set of parameters, which are shown in Table 2. The *permanent fault probability* means the probability of permanent faults among all the faults, which includes *network-related faults* and *logic-related faults*. The *performance degradation threshold* is employed by the dynamic strategy selection algorithm, which has been introduced in Section 5.3. *Dynamic degree* is used to control the QoS changing of replicas in our experimental system, where a larger number means more serious changing of QoS properties.

Table 2 Parameters of experiments

	Parameters	Setting
1	Number of replicas	6
2	Network fault probability	0.01
3	Logic fault probability	0.0025
4	Permanent fault probability	0.05
5	Number of time slots	20
6	Performance degradation threshold (e)	2
7	Dynamic degree	20
8	w_1	1/3
9	w_2	1/3
10	w_3	1/3

6.2 Studies of the Typical Examples

The experimental results of the six service users employing different types of fault tolerance strategies are shown in Tables 3, 4, 5, 6, 7 and 8. The results include the employed fault tolerance strategy (*Strategies*), the number of all requests (*All*), the average RTT of all requests (*RTT*), the number of failure (*Fail*), the average consumed resource (*Res*), and the overall performance (*Perf*, calculated by (11)). The time units of RTT is in milliseconds (ms).

In the following, we provide detailed explanation on the experimental results of Service User 1. As shown in Table 1, the requirements provided by User 1 are: $t_{max} = 1000$, $f_{max} = 0.1$ and $r_{max} = 50$. These requirement settings indicate that User 1 cares more on the response time than the failure-rate and resources, because 1000 ms maximal response time setting is tight in the high dynamic Internet environment, and the settings of failure-rate and the resource consumption are loose. As shown in Table 3, among all the strategies, the RTT performance of the NVP strategy is the worst since it needs to wait for all parallel responses before voting; the RTT performance of the *Active* strategy is the best, since it employs the first properly-returned response as the final result. The *Dynamic* strategy can provide good RTT performance, which is near the performance of the *Active* strategy.

The *Fail* column in Table 3 shows the fault tolerance performance of different strategies. The failure-rates of the *Retry* and RB strategies are not good, because these strategies are sequential and the setting of $t_{max} = 1000ms$ leads to a lot of timeout failures. Among all the strategies, NVP obtains the best fault tolerance performance. This is not only because NVP can tolerate logic-related faults by majority voting, but also because NVP invokes 5 replicas in parallel in our experiments, which greatly reduces the number of *timeout* failures. For example, if one replica does not respond within the required time period t_{max}, NVP can still get the correct result by conducting majority voting using the remaining responses. The fault tolerance performance of the *Dynamic* strategy is not good comparing with NVP. However, this fault tolerance performance is already good enough for User 1, who does not care so much about the failure-rate by setting $f_{max} = 0.1$.

The *Res* column in Table 3 shows the resource consumption information of different fault tolerance strategies. We can see that the resource consumption of *Retry* and RB strategies are equal to 1, because these two strategies invoke only one replica at the same time. In our experiments, the version number of NVP strategy is set to be 5 and the parallel invocation number of *Active* strategy is set to be 6. Therefore, the *Res* of these two strategies are 5 and 6, respectively. The *Dynamic* strategy invokes 2.34 replicas in parallel on average. The $Perf$ column shows the overall performance of different strategies calculated by (11). We can see that the *Dynamic* strategy achieves the best overall performance among all the strategies (smaller value for better performance). Although the *Active* strategy also achieves good performance for User 1, in the following experiments, we can see that it cannot always provide good overall performance under different environments.

As shown in Tables 4–8, for other service users, the *Dynamic* strategy can also provide a suitable strategy dynamically to achieve good performance. As shown in Fig. 5, the *Dynamic* strategy provides the best overall performance among all the fault tolerance strategies for all the six service users. This is because the *Dynamic* strategy considers user requirements and can adjust itself for optimal strategy

Table 3 Experimental results of user 1

U	Strategies	All	RTT	Fail	Res	Perf
1	Retry	50000	420	2853	1	1.011
	RB	50000	420	2808	1	1.002
	NVP	50000	839	2	5	0.939
	Active	50000	251	110	6	0.393
	Dynamic	50000	266	298	2.34	0.372

Table 4 Experimental results of user 2

U	Strategies	All	RTT	Fail	Res	Perf
2	Retry	50000	471	285	1	5.985
	RB	50000	469	283	1	5.944
	NVP	50000	855	0	5	0.677
	Active	50000	253	126	6	2.946
	Dynamic	50000	395	3	4.03	0.459

Table 5 Experimental results of user 3

U	Strategies	All	RTT	Fail	Res	Perf
3	Retry	50000	458	155	1	0.717
	RB	50000	457	149	1	0.713
	NVP	50000	845	1	5	2.712
	Active	50000	248	138	6	3.154
	Dynamic	50000	456	141	1	0.708

Table 6 Experimental results of user 4

U	Strategies	All	RTT	Fail	Res	Perf
4	Retry	50000	498	145	1	1.194
	RB	50000	493	131	1	1.180
	NVP	50000	868	1	5	5.087
	Active	50000	251	119	6	6.144
	Dynamic	50000	494	109	1	1.158

Table 7 Experimental results of user 5

U	Strategies	All	RTT	Fail	Res	Perf
5	Retry	50000	454	115	1	0.823
	RB	50000	450	121	1	0.847
	NVP	50000	779	0	5	1.718
	Active	50000	249	125	6	2.516
	Dynamic	50000	489	60	1.46	0.759

Table 8 Experimental results of user 6

U	Strategies	All	RTT	Fail	Res	Perf
6	Retry	50000	470	146	1	29.236
	RB	50000	468	119	1	23.835
	NVP	50000	839	1	5	0.304
	Active	50000	249	132	6	26.487
	Dynamic	50000	473	1	3.56	0.268

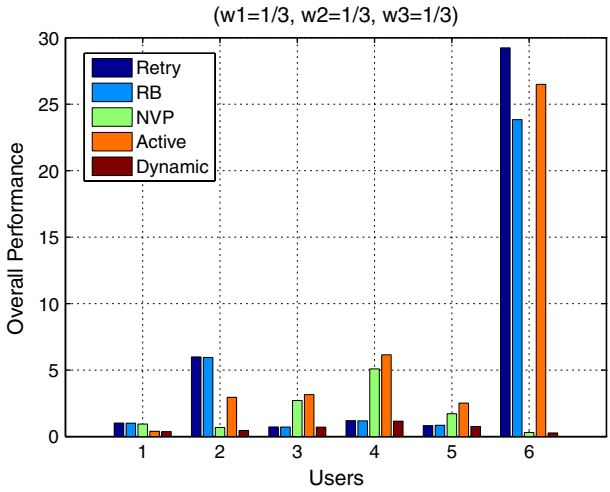


Fig. 5 Overall performance of different fault tolerance strategies

dynamically according to the change of QoS values of the replicas. The other four traditional fault tolerance strategies perform well in some situations; however, they perform badly in other situations, because they are too static. Our experimental results indicate that the traditional fault tolerance strategies may not be good choices in the field of service-oriented computing, which is highly dynamic. The experimental results also indicate that our proposed *Dynamic* fault tolerance strategy is more adaptable and can achieve better overall performance compared with traditional fault tolerance strategies.

6.3 Studies of Different User Requirements

In this section, we conduct experiments with different user requirement settings to study the influence of different requirement parameters (t_{max}, f_{max} and r_{max}). Each experiment is run for 5000 times and the experimental results are shown in Fig. 6.

Figure 6a shows the influence of the user requirement t_{max}, where the x-axis shows the different t_{max} settings (1000–10000 ms) and y-axis is the performance of different fault tolerance strategies calculated by (11). The settings of f_{max} and r_{max} are: $f_{max} = 0.1$, $r_{max} = 6$. Figure 6a shows that: 1) the performance of the sequential strategies *Retry* and RB are worse than the parallel strategies (NVP and *Active*) when the t_{max} is small (e.g., $t_{max} = 1000$), since the response-time performance of the sequential strategies are not good; 2) when $t_{max} > 2000$ ms, sequential fault tolerance strategies achieve better performance than the parallel strategies, since the user requirement on response-time is not tight; and 3) the *Dynamic* strategy, which is more adaptable, can provide the best performance under all the different t_{max} settings in our experiments.

Figure 6b shows the influence of the user requirement f_{max}, where the x-axis shows the different f_{max} settings (0.05–0.5). The settings of t_{max} and r_{max} are: $t_{max} = 1000$ and $r_{max} = 6$. Figure 6(b) shows that: 1) the performance of the sequential strategies *Retry* and RB are not good when f_{max} is small, since the sequential strategies have a lot of *time out* failures caused by the setting of $t_{max} = 1000$; 2) the performance of the sequential strategies increases with the increasing of f_{max}, since large f_{max} value

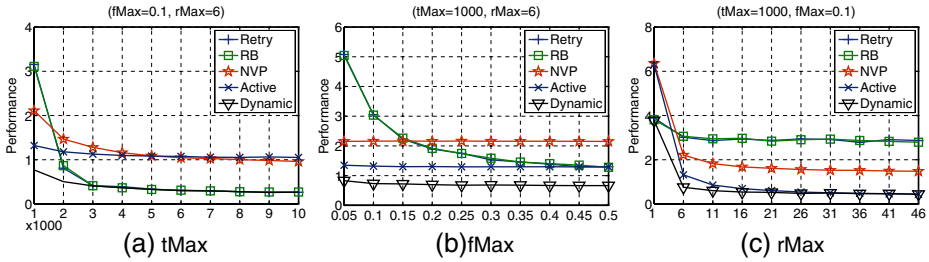


Fig. 6 Strategy performance with different tMax

indicates that the user requirement on the failure-rate is loose; 3) parallel strategies can provide steady performance in our experiments; and 4) the *Dynamic* strategy can provide the best performance under all the different f_{max} settings.

Figure 6c shows the influence of the user requirement r_{max}, where the x-axis shows different r_{max} settings (1–46). The settings of t_{max} and f_{max} are: $t_{max} = 1000$ and $r_{max} = 0.1$. Figure 6c shows that: 1) the performance of the parallel strategies enhance with the increasing of r_{max}, since the user can afford more resource consuming; and 2) the *Dynamic* strategy provides the best performance under all the different f_{max} settings.

The above experimental results show that the traditional fault tolerance strategies can provide good performance in some environments. However, with the changing of user requirements, the performance of traditional fault tolerance strategies cannot be guaranteed since these strategies cannot be auto-adapted to different environments. The *Dynamic* fault tolerance strategy, on the other hand, provides the best overall performance with different t_{max}, f_{max} and r_{max} settings in our experiments.

6.4 Studies of Different Faults

In this section, we study the performance of different fault tolerance strategies under various faults. The user requirements in these experiments are: $t_{max} = 2000$, $f_{max} = 0.1$, $r_{max} = 6$. The experimental results are shown in Fig. 7.

Figure 7a shows the performance of different fault tolerance strategies under different level of network faults (the x-axis), which is from 1%–10%. Figure 7a shows that: 1) the performance of the *NVP* strategy is not good, since the user requirement on the resource is tight ($r_{max} = 6$); 2) the performance of the sequential

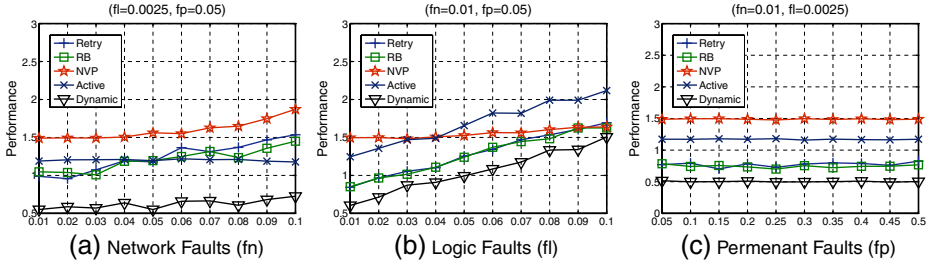


Fig. 7 Strategy performance under different level of faults

strategies degrades with the increasing of network faults, since more *timeout* errors occur (response time larger than t_{max}); and 3) the *Dynamic* strategy can provide the best performance under different levels of network faults.

Figure 7b shows the performance of different fault tolerance strategies under different level of logic faults (1%–10%). Figure 7b shows that: 1) with the increasing of the logic faults, the performance of the *Active* strategy degrades, since *Active* cannot tolerate logic faults; 2) *NVP* can tolerate logic faults; however, it invokes 5 replicas in parallel in our experiments, which consumes a lot of resource; and 3)the *Dynamic* strategy can provide the best performance under different levels of logic faults.

Figure 7c shows the performance of different fault tolerance strategies under different levels of permanent faults (5%–50%). Figure 7c shows that the *Dynamic* strategy can steadily provide the best performance under different levels of permanent faults.

The above experimental results show that the *Dynamic* fault tolerance strategy can provide the best overall performance under different levels of network faults, logic faults and permanent faults.

7 Related Work and Discussion

A number of fault tolerance strategies for Web services have been proposed in the recent literature (Chan et al. 2007; Foster et al. 2003; Salatge and Fabre 2007; Zheng and Lyu 2009). The major approaches can be divided into two types: 1) sequential strategies, where a primary service is invoked to process the request and some backup services are invoked only when the primary service fails. Sequential strategies have been employed in FT-SOAP (Fang et al. 2007), FT-CORBA (Sheu et al. 1997), and work (Chen and Lyu 2003). 2) parallel strategies, where all the candidates are invoked at the same time. Parallel strategies have been employed in FTWeb (Santos et al. 2005), Thema (Merideth et al. 2005) and WS-Replication (Salas et al. 2006). However, these fault tolerance strategies are too static and cannot auto-adapt to different environments. In this paper, we propose a context-aware dynamic fault tolerance strategy for achieving optimal fault tolerance for Web services.

QoS models for Web services have been discussed in a number of recent literature (Ardagna and Pernici 2007; Jaeger et al. 2004; Menasce 2002; O'Sullivan et al. 2002; Ouzzani and Bouguettaya 2004; Thio and Karunasekera 2005; Zheng et al. 2009b). The QoS data of Web services can be measured from either the service user's perspective (e.g., response-time, success-rate, ect.) or the service provider's perspective (e.g., price, availability, etc.). In this paper, we discuss the most representative QoS properties (RTT, failure-rate, and resources) and introduce the key concept of Web 2.0, *user-collaboration*, into our QoS model. QoS measurement of Web services has been used in the Service Level Agreement (SLA) (Ludwig et al. 2003), such as IBMs WSLA framework (Keller and Ludwig 2002) and the work from HP (Sahai et al. 2002). In SLA, the QoS data are mainly for the service providers to maintain a certain level of service to their clients and the QoS data are not available to others. On the other hand, we mainly focus on encouraging the service users to share their individually-obtained QoS data of the Web services, making efficient and effective Web service evaluation and selection.

The WS-ReliableMessaging (OASIS 2005b) can be employed in our middleware framework for enabling reliable message communication. WSRF (OASIS 2005a), which describes the state as XML data-sheets, can be employed for transferring states between different replicas. The proposed middleware can be integrated into the SOA runtime governance framework (Kavianpour 2007) and applied to industrial projects.

8 Conclusion

This paper proposes a dynamic adaptive fault tolerance strategy for Web services, which employs both objective replica QoS information as well as subjective user requirements for optimal strategy configuration determination. Based on a QoS-aware middleware, service users share their individually-obtained Web service QoS information with each other via a service community coordinator. Experiments are conducted and the performances of various fault tolerance strategies under different environments are compared. The experimental results indicate that the proposed *Dynamic* strategy can obtain better overall performance for various service users compared with traditional fault tolerance strategies.

More QoS properties will be involved in our QoS model for Web services in the future. More investigations are needed for the fault tolerance of stateful Web services, which need to maintain states across multiple tasks.

Acknowledgements The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/09E).

References

Apache (2008) Axis2. http://ws.apache.org/axis2

Ardagna D, Pernici B (2007) Adaptive service composition in flexible processes. IEEE Trans Softw Eng 33(6):369–384

Avizienis A (1995) The methodology of n-version programming. Software fault tolerance. Wiley, Chichester, pp 23–46

Benatallah B, Dumas M, Sheng QZ, Ngu AHH (2002) Declarative composition and peer-to-peer provisioning of dynamic web services. In: Proc 18th int'l conf data eng (ICDE'02)

Bram C (2003) Incentives build robustness in bittorrent. In: Proc first workshop economics of peer-to-peer systems, pp 1–5

Chan PP, Lyu MR, Malek M (2007) Reliable web services: methodology, experiment and modeling. In: Proc 5th int'l conf web services (ICWS'07), pp 679–686

Chen X, Lyu MR (2003) Message logging and recovery in wireless corba using access bridge. In: The 6th int'l symp autonomous decentralized systems, pp 107–114

Deora V, Shao J, Gray W, Fiddian N (2003) A quality of service management framework based on user expectations. In: Proc 1st int'l conf service-oriented computing (ICSOC'03), pp 104–114

Fang CL, Liang D, Lin F, Lin CC (2007) Fault tolerant web services. J Syst Archit 53(1):21–38

Foster H, Uchitel S, Magee J, Kramer J (2003) Model-based verification of web service compositions. In: ASE

Jaeger MC, Rojec-Goldmann G, Muhl G (2004) Qos aggregation for web service composition using workflow patterns. In: Proc 8th IEEE int'l enterprise computing conf, pp 149–159

Kavianpour M (2007) Soa and large scale and complex enterprise transformation. In: Proc 5th int'l conf service-oriented computing (ICSOC'07), pp 530–545
Keller A, Ludwig H (2002) The wsla framework: specifying and monitoring service level agreements for web services. In: IBM research division
Leu D, Bastani F, Leiss E (1990) The effect of statically and dynamically replicated components on system reliability. IEEE Trans Reliab 39(2):209–216
Looker N, Xu J (2003) Assessing the dependability of soaprpc-based web services by fault injection. In: Proc of the 9th int'l workshop on object-oriented real-time dependable systems
Ludwig H, Keller A, Dan A, King R, Franck R (2003) A service level agreement language for dynamic electronic services. Electron Commer Res 3(1–2):43–59
Lyu MR (1995) Software fault tolerance. Trends in software. Wiley, New York
Lyu MR (1996) Handbook of software reliability eng. McGraw-Hill, New York
Maximilien E, Singh M (2002) Conceptual model of web service reputation. ACM SIGMOD Record 31(4):36–41
Menasce DA (2002) Qos issues in web services. IEEE Internet Computing 6(6):72–75
Merideth MG, Iyengar A, Mikalsen T, Tai S, Rouvellou I, Narasimhan P (2005) Thema: Byzantine-fault-tolerant middleware forweb-service applications. In: Proc 24th IEEE symp reliable distributed systems (SRDS'05), pp 131–142
OASIS (2005a) Web service resource framework. http://www.oasis-open.org/committees/wsrf/
OASIS (2005b) Web services reliable messaging protocol. http://specs.xmlsoap.org/ws/2005/02/rm
O'Sullivan J, Edmond D, ter Hofstede AHM (2002) What's in a service? Distributed and Parallel Databases 12(2/3):117–133
Ouzzani M, Bouguettaya A (2004) Efficient access to web services. IEEE Internet Computing 8(2):34–44
Randell B, Xu J (1995) The evolution of the recovery block concept. In: Lyu MR (ed) Software fault tolerance. Wiley, Chichester, pp 1–21
Sahai A, Durante A, Machiraju V (2002) Towards automated sla management for web services. In: HP laboratory
Salas J, Perez-Sorrosal F, Marta Pati nM, Jiménez-Peris R (2006) Ws-replication: a framework for highly available web services. In: Proc 15th int'l conf world wide web (WWW'06), pp 357–366
Salatge N, Fabre JC (2007) Fault tolerance connectors for unreliable web services. In: Proc 37th Int'l conf dependable systems and networks (DSN'07), pp 51–60
Santos GT, Lung LC, Montez C (2005) Ftweb: a fault tolerant infrastructure for web services. In: Proc 9th IEEE int'l enterprise computing conf, pp 95–105
Sheu GW, Chang YS, Liang D, Yuan SM, Lo W (1997) A fault-tolerant object service on corba. In: Proc 17th int'l conf distributed computing systems (ICDCS'97), p 393
Thio N, Karunasekera S (2005) Automatic measurement of a qos metric for web service recommendation. In: Proc. Australian software engineering conference, pp 202–211
Tsai W, Paul R, Yu L, Saimi A, Cao Z (2003) Scenario-based web service testing with distributed agents. IEICE Trans Inf Syst E86-D(10):2130–2144
Vieira M, Laranjeiro N, Madeira H (2007) Assessing robustness of web-services infrastructures. In: Proc 37th int'l conf dependable systems and networks (DSN'07), pp 131–136
Wu G, Wei J, Qiao X, Li L (2007) A bayesian network based qos assessment model for web services. In: Proc int'l conf services computing (SCC'07), pp 498–505
Wu J, Wu Z (2005) Similarity-based web service matchmaking. In: Proc int'l conf services computing (SCC'05), pp 287–294
Zeng L, Benatallah B, Ngu AH, Dumas M, Kalagnanam J, Chang H (2004) Qos-aware middleware for web services composition. IEEE Trans Softw Eng 30(5):311–327
Zheng Z, Lyu MR (2008a) A distributed replication strategy evaluation and selection framework for fault tolerant web services. In: Proc 6th int'l conf web services (ICWS'08), pp 145–152
Zheng Z, Lyu MR (2008b) A qos-aware middleware for fault tolerant web services. In: Proc int'l symp software reliability engineering (ISSRE'08), pp 97–106
Zheng Z, Lyu MR (2009) A qos-aware fault tolerant middleware for dependable service composition. In: Proc 39th int'l conf dependable systems and networks (DSN'09), pp 239–248
Zheng W, Lyu MR, Xie T (2009a) Test selection for result inspection via mining predicate rules. In: Companion Proc 31th int'l conf software eng, new ideas and emerging results, pp 219–222
Zheng Z, Ma H, Lyu MR, King I (2009b) Wsrec: a collaborative filtering based web service recommender system. In: Proc 7th int'l conf web services (ICWS'09), pp 437–444

Zibin Zheng received his B.Eng. degree and M.Phil. degree in Computer Science from the Sun Yat-sen University, Guangzhou, China, in 2005 and 2007, respectively. He is currently a Ph.D. candidate in the department of Computer Science and Engineering, The Chinese University of Hong Kong. He served as program committee member of the IEEE CLOUD 2009. He also served as reviewer for international journals as well as conferences including TSE, TPDS, TSC, IJCCBS, IJBPIM, WWW, WSDM, DSN, ISSRE, PRDC, ISAS, HASE, SEKE, P2P, SCC, etc. His research interests include service computing, software reliability engineering, and Web technology.

Michael R. Lyu received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1981; the M.S. degree in computer engineering from University of California, Santa Barbara, in 1985; and the Ph.D. degree in computer science from the University of California, Los Angeles, in 1988. He is currently a Professor in the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, China. He is also Director of the Video over Internet and Wireless (VIEW) Technologies Laboratory. He was with the Jet Propulsion Laboratory as a Technical Staff Member from 1988 to 1990. From 1990 to 1992, he was with the Department of Electrical and Computer Engineering, University of Iowa, Iowa City, as an Assistant Professor. From 1992 to 1995, he was a Member of Technical Staff in the applied research area of Bell Communications Research (Bellcore), Morristown, NJ. From 1995 to 1997, he was a Research Member of Technical Staff at Bell Laboratories, Murray Hill, NJ. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, mobile networks, Web technologies, multimedia information processing, and E-commerce systems. He has published over 270 refereed journal and conference papers in these areas. He has participated in more than 30 industrial projects and helped to develop many commercial systems and software tools. He was the editor of two book volumes: Software Fault Tolerance (New York: Wiley, 1995) and The

Handbook of Software Reliability Engineering (New York: IEEE and New McGraw-Hill, 1996). Dr. Lyu received Best Paper Awards at ISSRE'98 and ISSRE'2003. Dr. Lyu initiated the First International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the Program Chair for ISSRE'96 and General Chair for ISSRE'2001. He was also PRDC'99 Program Co-Chair, WWW10 Program Co-Chair, SRDS'2005 Program Co-Chair, PRDC'2005 General Co-Chair, and ICEBE'2007 Program Co-Chair, and served in program committees for many other conferences including HASE, ICECCS, ISIT, FTCS, DSN, ICDSN, EUROMICRO, APSEC, PRDC, PSAM, ICCCN, ISESE, and WI. He has been frequently invited as a keynote or tutorial speaker to conferences and workshops in the U.S., Europe, and Asia. He has been on the Editorial Board of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, the IEEE TRANSACTIONS ON RELIABILITY, the Journal of Information Science and Engineering, and Software Testing, Verification & Reliability Journal.

Dr. Lyu is an IEEE Fellow and an AAAS Fellow, for his contributions to software reliability engineering and software fault tolerance. He is also a Croucher Senior Research Fellow.