Component Ranking for Fault-Tolerant Cloud Applications

Zibin Zheng, *Member*, *IEEE*, Tom Chao Zhou, *Student Member*, *IEEE*, Michael R. Lyu, *Fellow*, *IEEE*, and Irwin King, *Senior Member*, *IEEE*

Abstract—Cloud computing is becoming a mainstream aspect of information technology. More and more enterprises deploy their software systems in the cloud environment. The cloud applications are usually large scale and include a lot of distributed cloud components. Building highly reliable cloud applications is a challenging and critical research problem. To attack this challenge, we propose a component ranking framework, named FTCloud, for building fault-tolerant cloud applications. FTCloud includes two ranking algorithms. The first algorithm employs component invocation structures and invocation frequencies for making significant component ranking. The second ranking algorithm systematically fuses the system structure information as well as the application designers' wisdom to identify the significant components in a cloud application. After the component ranking phase, an algorithm is proposed to automatically determine an optimal fault-tolerance strategy for the significant cloud components. The experimental results show that by tolerating faults of a small part of the most significant components, the reliability of cloud applications can be greatly improved.

Index Terms—Cloud application, component ranking, fault tolerance, software reliability

1 Introduction

Cloud computing is a style of Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand, like the electricity grid [4], [8]. Promoted by the leading industrial companies (e.g., Amazon, Google, IBM, Microsoft, etc.), cloud computing is becoming increasingly popular in recent years. The software systems in the cloud (named as cloud applications) typically involve multiple cloud components communicating with each other [1]. The cloud applications are usually large scale and very complex. Before enterprises transfer their critical systems to the cloud environment, one question they ask is: *Can cloud become as reliable as the power grid achieving 99.999 percent uptime?* Unfortunately, the reliability of the cloud applications is still far from perfect in reality. Nowadays, the demand for highly reliable cloud applications is becoming unprecedentedly strong. Building highly reliable clouds becomes a critical, challenging, and urgently required research problem.

In traditional software reliability engineering, there are four main approaches to build reliable software systems, i.e., fault prevention, fault removal, fault tolerance, and fault forecasting [21]. The trend toward large-scale complex cloud applications makes developing fault-free systems by only employing fault-prevention techniques (e.g., by rigorous development process) and fault-removal techniques (e.g., by testing and debugging techniques) exceedingly difficult. Another approach for building reliable systems, software fault tolerance [20], makes the system more robust by masking faults instead of removing faults. One of the most well-known software fault-tolerance techniques, also known as *design diversity*, is to employ functionally equivalent yet independently designed components to tolerate faults [5]. Due to the cost of developing and maintaining redundant components, software fault tolerance is usually only employed for critical systems (e.g., airplane flight control systems, nuclear power station management systems, etc.). Different from traditional software systems, there are a lot of redundant resources in the cloud environment, making software fault tolerance a possible approach for building highly reliable cloud applications.

Since cloud applications usually involve a large number of components, it is still too expensive to provide alternative components for all the cloud components. Moreover, there is probably no need to provide fault-tolerance mechanisms for the noncritical components, whose failures have limited impact on the systems. To reduce the cost so as to develop highly reliable cloud applications within a limited budget, a small set of critical components needs to be identified from the cloud applications. Microsoft reported that by fixing the top 20 percent of the most reported bugs of Windows and Office, 80 percent of the failures and crashes would be eliminated [26]. Our idea is also based on this well-known 80-20 rules, i.e., by tolerating faults of a small part of the most important cloud components, the cloud application reliability can be greatly improved. Based on this idea, we propose FTCloud, which is a component ranking framework for building fault-tolerant cloud applications. FTCloud identifies the most significant components and suggests

- *Z. Zheng is with the Shenzhen Research Institute and the Department of Computer Science and Engineering, The Chinese University of Hong Kong, and also with State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. E-mail: zbzheng@cse.cuhk.edu.hk.*
- *T.C. Zhou and I. King are with the Shenzhen Research Institute and the Department of Computer Science and Engineering, The Chinese University of Hong Kong. E-mail: {czhou, king}@cse.cuhk.edu.hk.*
- *M.R. Lyu is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, and the School of Computer Science, National University of Defence Technology, Hunan, China. E-mail: lyu@cse.cuhk.edu.hk.*

Manuscript received 1 Dec. 2010; revised 15 May 2011; accepted 11 June 2011; published online 13 July 2011.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org and reference IEEECS Log Number TSCSI-2010-12-0155.
Digital Object Identifier no. 10.1109/TSC.2011.42.

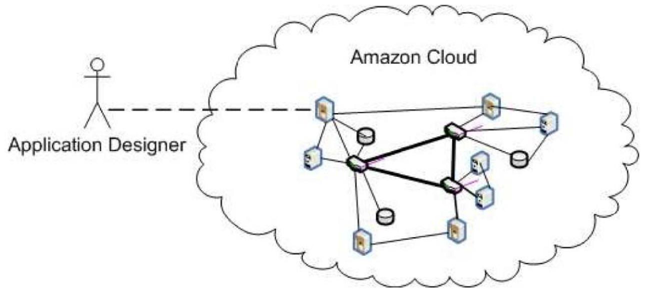


Fig. 1. Motivating example.

the optimal fault-tolerance strategies for these significant components automatically. FTCloud can be employed by designers of cloud applications to design more reliable and robust cloud applications efficiently and effectively.

The contribution of this paper is twofold:

- This paper identifies the critical problem of locating significant components in complex cloud applications and proposes a ranking-based framework, named FTCloud, to build fault-tolerant cloud applications. We first propose two ranking algorithms to identify significant components from the huge amount of cloud components. Then, we present an optimal fault-tolerance strategy selection algorithm to determine the most suitable fault-tolerance strategy for each significant component. We consider FTCloud as the first ranking-based framework for developing fault-tolerant cloud applications.
- We provide extensive experiments to evaluate the impact of significant components on the reliability of cloud applications.

The rest of this paper is organized as follows: Section 2 introduces a motivating example and system architecture, Section 3 proposes two ranking algorithms for discovering significant components, Section 4 presents an optimal fault-tolerance strategy selection algorithm, Section 5 shows experiments, Section 6 introduces related work, and Section 7 concludes the paper.

2 Preliminaries

2.1 Motivating Example

We begin by using a motivating example to show the research problem of this paper. Amazon[1] delivers computer infrastructure as a service to the cloud users (known as Infrastructure as a Service (IaaS)). As shown in Fig. 1, there are a number of distributed components in the Amazon cloud, including Amazon Elastic Compute Cloud (EC2)[2] for providing computing capability, Amazon Simple Storage Service (S3)[3] for providing content storage, and so on. These cloud components are employed by application designers to implement and deploy their own cloud applications.

To enhance the system reliability, the cloud application designer wants to provide fault-tolerance mechanisms by replicating the components. When designing fault-tolerance mechanisms for the cloud application, the cloud application designer encounters the following problems:

- There are a lot of components within the cloud application. It is too expensive to provide alternative components for all the cloud components, since there is a charge for using the Amazon cloud components (e.g., the virtual machines of EC2 or S3). To save money, the designer wants to only tolerate faults of the most important components, whose failures have great impact on the whole system. However, since the cloud applications are usually very complex, it is not easy to identify the important components for a cloud application.
- There are a number of fault-tolerance strategies. The designer may not be an expert on software fault tolerance. It is a challenging task for the application designer to find out the optimal fault-tolerance strategies for the significant cloud components.

To address the above problems, we first propose a component ranking framework at Section 3, which ranks the component automatically for the application designer. After that, we present an optimal fault-tolerance strategy selection algorithm at Section 4, which suggests optimal fault-tolerance strategies for application designers.

1. http://www.amazon.com.
2. http://aws.amazon.com/ec2.
3. http://aws.amazon.com/s3.

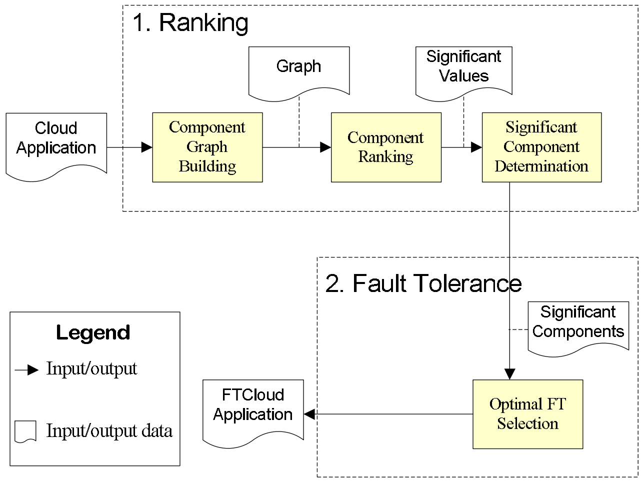


Fig. 2. System architecture of FTCloud.

2.2 System Architecture

Fig. 2 shows the system architecture of our fault-tolerance framework (named FTCloud), which includes two parts: 1) ranking and 2) optimal fault-tolerance selection. The procedures of FTCloud are as follows:

1. The system designer provides the initial architecture design of a cloud application to FTCloud. A component graph is built for the cloud application based on the component invocation relationships.
2. Significance values of the cloud components are calculated by employing component ranking algorithms. Based on the significance values, the components can be ranked.
3. The most significant components in the cloud application are identified based on the ranking results.
4. The performance of various fault-tolerance strategy candidates is calculated and the most suitable

fault-tolerance strategy is selected for each significant component.

5. The component ranking results and the selected fault-tolerance strategies for the significant components are returned to the system designer for building a reliable cloud application.

Section 3 will introduce the technique details of the component ranking algorithm and Section 4 will present the optimal fault-tolerance strategy selection algorithm.

3 Significant Component Ranking

The target of our significant component ranking algorithm is to measure the importance of cloud components based on available information (e.g., application structure, component invocation relationships, component characteristics, etc.). As shown in Fig. 2, our significant component ranking includes three steps (i.e., component graph building, component ranking, and significant component determination), which will be described in Sections 3.1 to 3.3, respectively.

3.1 Component Graph Building

A cloud application can be modeled as a weighted directed graph G, where a node c_i in the graph represents a component and a directed edge e_{ij} from node c_i to node c_j represents a component invocation relationship, i.e., c_i invokes c_j. Each node c_i in the graph G has a nonnegative significance value $V(c_i)$, which is in the range of (0,1). Each edge e_{ij} in the graph has a nonnegative weight value $W(e_{ij})$, which is in the range of [0,1]. The weight value of an edge e_{ij} can be calculated by

$$W(e_{ij}) = \frac{frq_{ij}}{\sum_{j=1}^{n} frq_{ij}}, \tag{1}$$

where frq_{ij} is the invocation frequency of component c_j by component c_i, n is the number of components, and $frq_{ij} = 0$ if component c_i does not invoke c_j. In this way, the edge e_{ij} has a larger weight value if component c_j is invoked more frequently by component c_i compared with other components invoked by c_i.

For a component graph which contains n components, an $n \times n$ transition probability matrix W can be obtained by employing (1) to calculate the invocation weight values of the edges. Each entry w_{ij} in the matrix is the value of $W(e_{ij})$. $w_{ij} = 0$ if there is no edge from c_i to c_j, which means that c_i does not invoke c_j. If a component does not invoke itself, $w_{ii} = 0$. Otherwise, the value of w_{ii} can be calculated by (1). In the case that a node c_i has no outgoing edge, $w_{ij} = \frac{1}{n}$. For $\forall i$, the transition probability matrix W satisfies:

$$\forall i, \sum_{j=1}^{n} w_{ij} = 1. \tag{2}$$

3.2 Component Ranking

Based on the component graph, two component ranking algorithms, named as FTCloud1 and FTCloud2, are proposed in this section. The first approach employs the system structure information (i.e., the component invocation relationships and frequencies) for making component ranking. The second approach not only considers the system structure, but also considers the component characteristics (i.e., critical components or noncritical components) for making component ranking.

3.2.1 *FTCloud1: Structure-Based Component Ranking*

In a cloud application, some components are frequently invoked by a lot of other components. These components are considered to be more important, since their failures will have greater impact on the system compared with other components. Intuitively, the *significant components* in a cloud application are the ones which have many invocation links coming in from the other important components. Inspired by the PageRank algorithm [7], we propose an algorithm to calculate the significance values of the cloud components employing the component invocation relationships and frequencies. The procedure of this structure-based component ranking algorithm is shown in the following steps:

1. Randomly assign initial numerical scores between 0 and 1 to the components in the graph.
2. Compute the significance value for a component c_i by:

$$V(c_i) = \frac{1-d}{n} + d \sum_{k \in N(c_i)} V(c_k) W(e_{ki}), \tag{3}$$

where n is the number of components and $N(c_i)$ is a set of components that invoke component c_i. The parameter d ($0 \le d \le 1$) in (3) is employed to adjust the significance values derived from other components, so that the significance value of c_i is composed of the basic value of itself (i.e., $\frac{1-d}{n}$) and the derived values from the components that invoked c_i. By (3), a component c_i has 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 vector form, (3) can be written as

$$\begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix} = \begin{bmatrix} (1-d)/n \\ \vdots \\ (1-d)/n \end{bmatrix} + dW^t \begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix}, \tag{4}$$

where W^t is the transposed matrix of the transition probability matrix W.

3. Solve (4) by computing the eigenvector with eigenvalue 1 or by repeating the computation until all significance values become stable.

With the above approach, the significance values of the cloud components can be obtained. A component is considered to be more significant, if it has a larger significance value. The failure of such these significant components will have great impact on other components and the whole system.

3.2.2 *FTCloud2: Hybrid Component Ranking*

The structure-based approach ranks the components by only employing the information of component invocation relationships and frequencies. It does not consider the characteristics of the components. For example, some components fulfill critical tasks (e.g., payment), while other components accomplish noncritical tasks (e.g., providing advertisements to display in the web page). Failures of the critical components have great impact on the system and thus have higher fault-tolerance requirement. On the other hand, failures of the noncritical components have smaller impact on the system. Fault-tolerance requirement for the noncritical components is therefore not high.

In order to rank the components as accurate as possible, we propose a hybrid component ranking approach, which considers both the system structure as well as the component characteristics (i.e., critical components or noncritical components) as follow:

1. Randomly assign initial numerical scores between 0 and 1 to the components in the graph. Divide the components in the graph into two component sets, critical components C and noncritical components NC, employing the prior knowledge provided by the system designers.
2. If a component c_i is a critical component ($c_i \in C$), compute the significance value for component c_i by

$$V(c_i) = (1-d)\frac{\beta}{|C|} + d \sum_{k \in N(c_i)} V(c_k)W(e_{ki}), \quad (5)$$

and if a component c_i is a noncritical component ($c_i \in NC$), compute the significance value for component c_i by

$$V(c_i) = (1-d)\frac{1-\beta}{|NC|} + d \sum_{k \in N(c_i)} V(c_k)W(e_{ki}), \quad (6)$$

where $|C|$ and $|NC|$ are the numbers of critical components and noncritical components, respectively, $|C| + |NC| = n$, and $N(c_i)$ is a set of components that invoke component c_i. The parameter d ($0 \leq d \leq 1$) in (5) is employed to adjust the significance values derived from other components, so that the significance value of c_i is composed of the basic value of itself and the derived values from the components that invoke c_i. The parameter β ($\frac{|C|}{n} \leq \beta \leq 1$) is employed to determine how much the hybrid approach relies on the critical components and the noncritical components. When $\beta = \frac{|C|}{n}$, the hybrid approach degrades to the structure-based approach, which treats the critical components and the noncritical components equally. When $\frac{|C|}{n} < \beta \leq 1$, the hybrid approach bias toward critical components. In other word, the basic value of the critical component $(1-d)\frac{\beta}{|C|}$ is larger than that of the noncritical components $(1-d)\frac{1-\beta}{|NC|}$. When $\beta = 1$, the basic value of the noncritical components is equal to 0, leading to smaller significance values of the noncritical components.

In vector form, (3) can be written as

$$\begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + dW^t \begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix}, \quad (7)$$

where

$$x_i = \begin{cases} (1-d)\frac{\beta}{|C|}, & if\ c_i \in C \\ (1-d)\frac{1-\beta}{|NC|}, & if\ c_i \in NC. \end{cases} \quad (8)$$

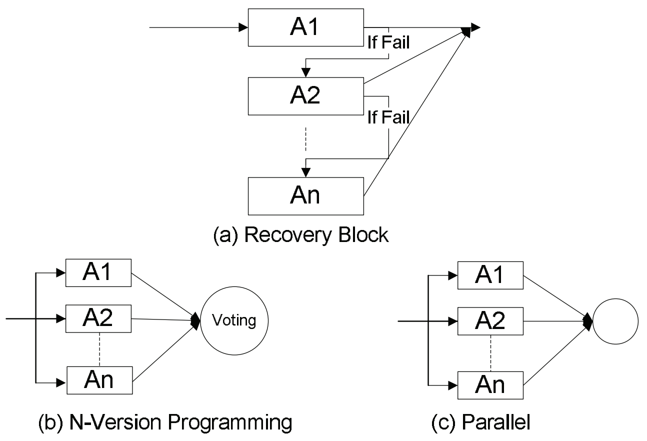


Fig. 3. Fault-tolerance strategies.

3. Solve (7) by computing the eigenvector with eigenvalue 1 or by repeating the computation until all significance values become stable.

With the above approach, the significance values of the cloud components can be calculated by employing both the system structure information and the component characteristics.

3.3 Significant Component Determination

Based on the obtained significance values of the components in the cloud application, the components can be ranked and the top k ($1 \leq k \leq n$) most significant components can be returned to the designer of the cloud application. In this way, the application designer can identify significant components early at the architecture design time and can employ various techniques (e.g., fault-tolerance techniques which will be introduced in Section 4) to improve the reliability of these significant components.

4 Fault-Tolerance Strategy Selection

4.1 Fault-Tolerance Strategies

Software fault tolerance is widely adopted to increase the overall system reliability in critical applications. System reliability can be improved by employing functionally equivalent components to tolerate component failures. Three well-known fault-tolerance strategies are introduced in the following with formulas for calculating the failure probabilities of the fault-tolerant modules. In this paper, failure probability of a cloud component is defined as the probability that an invocation to this component will fail. The value of failure probability is in the range of [0,1].

- **Recovery block (RB)**. Recovery block [25] is a well-known mechanism employed in software fault tolerance. As shown in Fig. 3a, a recovery block is a means of structuring redundant program modules, where standby components will be invoked sequentially if the primary component fails. A recovery block fails only if all the redundant components fail. The failure probability f of a recovery block can be calculated by:

$$f = \prod_{i=1}^{n} f_i, \quad (9)$$

TABLE 1
Fault-Tolerance Strategy Comparison

	RB	NVP	Parallel
Response-time	Middle	Middle	Good
Required resources	Middle	High	High
Fault tolerance	Crash	Crash, Value	Crash

where n is the number of redundant components and f_i is the failure probability of the ith component.

- **N-version programming (NVP)**. N-version programming, also known as multiversion programming, is a software fault-tolerance method where multiple functionally equivalent programs (named as *versions*) are independently generated from the same initial specifications [5]. As shown in Fig. 3b, When applying the NVP approach to the cloud applications, the independently implemented functionally equivalent cloud components are invoked in parallel and the final result is determined by majority voting. The failure probability f of an NVP module can be computed by

$$f = \sum_{i=\frac{n+1}{2}}^{n} F(i), \tag{10}$$

where n is the number of functionally equivalent components (n is usually an odd number in NVP) and $F(i)$ is probability that i alternative components from all the n components fail. For example, when $n = 3$, then $f = F(2) + F(3)$, where $F(2) = f_1f_2(1 - f_3) + f_1(1 - f_2)f_3 + (1 - f_1)f_2f_3$ and $F(3) = f_1f_2f_3$. In other words, an NVP module fails only if more than half of the redundant components fail.

- **Parallel**. As shown in Fig. 3c, parallel strategy invokes all the n functional equivalent components in parallel and the first returned response will be employed as the final result. An parallel module fails only if all the redundant components fail. The failure probability f of a parallel module can be computed by

$$f = \prod_{i=1}^{n} f_i, \tag{11}$$

where n is the number of redundant components and f_i is the failure probability of the ith component.

Different fault-tolerance strategies have different features. As shown in Table 1, the response time performance of RB and NVP strategies is not good compared with Parallel strategy, since RB strategy invokes standby component sequentially when the primary component fails, NVP strategy needs to wait for all the n responses from the parallel invocations for determining the final result, while Parallel strategy employs the first returned response as the final result. The required resources of NVP and Parallel are much higher than those of RB since parallel component invocations consume a lot of networking and computing resources. All RB, NVP, and Parallel strategies can tolerate crash faults (e.g., component crash, communication link crash, etc.). NVP strategy can also mask value faults (e.g., data corruption), since majority voting is employed for determining the final results in NVP.

Employing a suitable fault-tolerance strategy for the cloud components is important to achieve optimal cloud application design. For example, RB strategy for the resource-constrained components, NVP strategy for the components with value faults and Parallel strategy for the components which have restrict real-time requirements. Since cloud applications usually include a large number of distributed components, automatic optimal fault-tolerance strategy selection reduces the workload of system designers and helps achieve optimal allocation of resources.

Employing the component ranking algorithm in Section 3, a set of significant components can be identified from the cloud application. The optimal fault-tolerance strategies can be determined for these significant components employing the approach proposed in Section 4.2.

4.2 Optimal FT Strategy Selection

The fault-tolerance strategies have a number of variations based on different configurations. For example, both RB and Parallel strategies have $n - 1$ variations (i.e., configured with $2, 3, \ldots, n$ redundant components), where n is the maximal number of redundant components. NVP strategy has $(n - 1)/2$ variations (i.e., NVP with $3, 5, \ldots, n$ redundant components), where n is an odd number. For each significant component in a cloud application, these fault-tolerance strategy variations are candidates and the optimal one needs to be identified.

For each significant component that requires a fault-tolerance strategy, the designer can specify constraints (e.g., *response time of the component has to be smaller than 1,000 milliseconds*, etc.). Two user constraints are considered: one for *response time* and one for *cost*. The optimal fault-tolerance strategy selection problem for a cloud component with user constraints can then be formulated mathematically as

Problem 1. ***Minimize:*** $\sum_{i=1}^{m} f_i \times x_i$
Subject to:

- $\sum_{i=1}^{m} s_i \times x_i \leq u_1$,
- $\sum_{i=1}^{m} t_i \times x_i \leq u_2$,
- $\sum_{i=1}^{m} x_i = 1$,
- $x_i \in \{0, 1\}$.

In Problem 1, x_i is set to 1 if the ith candidate is selected for the component and 0 otherwise. Moreover, f_i, s_i, and t_i are the failure probability, cost, and response time of the strategy candidates, respectively, m is the number of fault-tolerance strategy candidates for the component, and u_1 and u_2 are the user constraints for cost and response time, respectively. Problem 1 is extensible, where more user constraints can be readily included in the future.

To solve Problem 1, we first calculate the cost, response-time, and the aggregated failure probability values of different fault-tolerance strategy candidates employing the equations presented in Section 4.1. Then, Algorithm 1 is designed to select the optimal candidate. First, the candidates which cannot meet the user constraints are excluded. After that, the fault-tolerance candidate with the best failure probability performance will be selected as the optimal

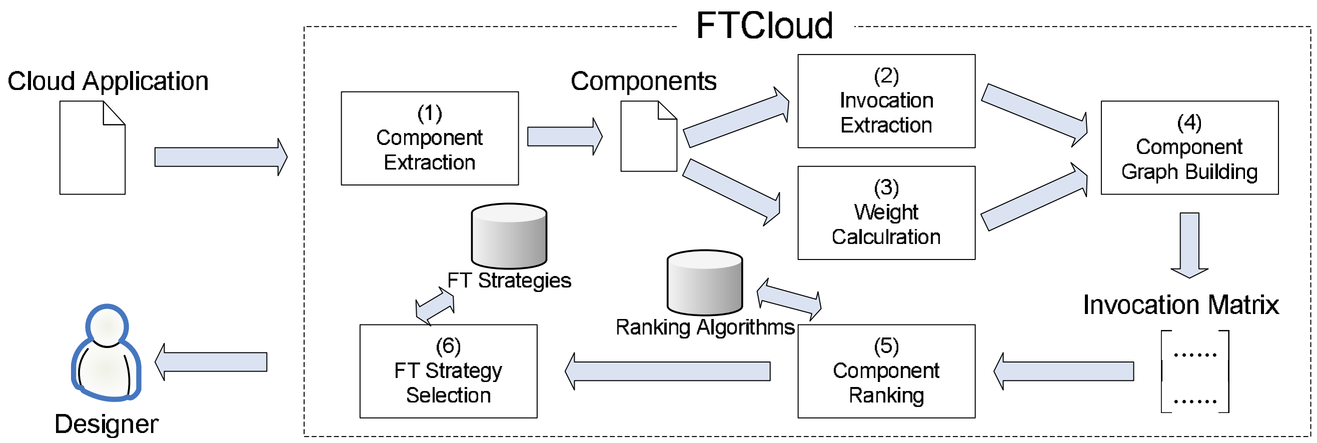


Fig. 4. Implementation of FTCloud.

strategy for component i. By the above approach, the optimal fault-tolerance strategy, which has the best failure probability performance and meets all the user constraints, can be identified.

Algorithm 1. Optimal FT Strategy Selection

```
Input: $s_i$, $t_i$, and $f_i$ values of candidates; user constraints
        $u_1$, $u_2$;
Output: Optimal candidate index $\rho$.
$m$: number of candidates;
for ($i = 1; i \leq m; i++$) do
    if ($s_i \leq u_1 \&\& t_i \leq u_2$) then
        $v_i = f_i$;
    end
end
if no candidate meet user constraints then
    Throw exception;
end
Select $v_x$ which has minimal value from all the $v_i$;
$\rho = x$;
```

5 Experiments

5.1 Prototype Implementation

A prototype of FTCloud is implemented. As shown in Fig. 4, our FTCloud implementation includes several modules:

- *Component extraction*. The components are extracted from a cloud application.
- *Invocation extraction*. The invocation links of different components are extracted from a cloud application.
- *Weight calculation*. The weight values of the invocation links are calculated by (1), which has been introduced in Section 3.1.
- *Component graph building*. Based on the components and the invocation links, a component graph is built for a cloud application.
- *Component ranking*. The significant component ranking algorithms in Section 3.2 are implemented and encapsulated in this module. The input of this module is the component invocation probability matrix and the output is a list of ranked components based on their significance values. Our component ranking framework is extensible. Most component ranking algorithms can be added easily in the future.
- *FT strategy selection*. The optimal fault-tolerance strategy selection algorithm presented in Section 4.2 is implemented in this module. This module calculates failure probabilities of various fault-tolerance strategy candidates and selects the most suitable one for each significant component.
- *FT strategies*. This module defines different fault-tolerance strategies. The design of this module makes our fault-tolerance model extensible, where more fault-tolerance strategy candidates can be added easily.

5.2 Experimental Setup

Our significant component ranking algorithms are implemented by C++ language. To study the performance of reliability improvement, we compare five approaches, which are as follows:

- **NoFT**. No fault-tolerance strategies are employed for the components in the cloud application.
- **RandomFT**. Fault-tolerance strategies are employed to mask faults of K percent components, which are randomly selected.
- **FTCloud1**. Fault-tolerance strategies are employed to mask faults of the Top-K percent significant components. The components are ranked by employing the structure information of the cloud application.
- **FTCloud2**. Fault-tolerance strategies are employed to mask faults of the Top-K percent significant components. The components are ranked by employing the hybrid component ranking algorithm, which considers the structure information as well as the prior knowledge of component characteristics.
- **AllFT**. Fault-tolerance strategies are employed for all the cloud components.

A scale-free graph is a graph whose degree distribution follows a power law. Many empirically observed networks appear to be scale free, including the protein networks, citation networks, and some social networks. Several previous work [10], [15] show that the internal structures of software programs (e.g., class collaboration graphs, call graphs for procedural code, inter package dependency of

TABLE 2
Performance Comparison of Failure Probability

Node Numbers	Methods	Component FP = 0.1%			Component FP = 0.5%			Component FP = 1%		
		Top1%	Top5%	Top10%	Top1%	Top5%	Top10%	Top1%	Top5%	Top10%
100	NoFT	0.090	0.090	0.090	0.335	0.335	0.335	0.507	0.507	0.507
	RandomFT	0.088	0.085	0.085	0.330	0.323	0.309	0.500	0.493	0.472
	FTCloud1	0.087	0.074	0.061	0.327	0.289	0.247	0.495	0.451	0.398
	FTCloud2	0.083	0.071	0.057	0.317	0.277	0.236	0.485	0.442	0.386
	AllFT	0.000	0.000	0.000	0.002	0.002	0.002	0.009	0.009	0.009
1000	NoFT	0.496	0.496	0.496	0.833	0.833	0.833	0.910	0.910	0.910
	RandomFT	0.495	0.482	0.477	0.832	0.824	0.817	0.909	0.905	0.904
	FTCloud1	0.466	0.413	0.369	0.816	0.780	0.749	0.900	0.878	0.858
	FTCloud2	0.454	0.402	0.357	0.808	0.773	0.737	0.895	0.872	0.851
	AllFT	0.001	0.001	0.001	0.025	0.025	0.025	0.089	0.089	0.089
10000	NoFT	0.886	0.886	0.886	0.975	0.975	0.975	0.989	0.989	0.989
	RandomFT	0.883	0.879	0.873	0.975	0.974	0.972	0.989	0.988	0.988
	FTCloud1	0.871	0.845	0.817	0.972	0.963	0.954	0.987	0.983	0.979
	FTCloud2	0.866	0.838	0.811	0.970	0.961	0.952	0.987	0.982	0.978
	AllFT	0.008	0.008	0.008	0.165	0.165	0.165	0.443	0.443	0.443

applications, etc.) exhibit approximate scale-free properties. We use Pajek [6] to generate scale free directed component graphs for making experimental studies and comparing the performance of different approaches.

For a cloud component, we employ the fault-tolerance strategy determination algorithm to automatically select the optimal fault-tolerance strategy for tolerating faults. During the execution of the cloud application, the execution is considered as failed if an invoked component is failed and there is no fault-tolerance strategy for this component. If a fault-tolerance strategy is applied for this component, the component fails only when the whole fault-tolerance strategy fails. In the FTCloud1 and FTCloud2 approaches, the parameter d balances the significance value derived from the other components and the basic value of the component itself. In our experiment, the component ranks are fairly stable when we change the parameter d of (3) from 0.75 to 0.95. Therefore, similar to the work [7], [16], we also set the parameter d to be 0.85 in all the experiments.

5.3 Performance Comparison

We employ random walk to simulate the invocation behavior in cloud applications. Specifically, Pajek [6] is employed to generate scale free directed component graphs and the generated edge weights are used to present the component invocation probability. To start a random walk, a node in the invocation graph is randomly selected as the start node. A very small stop rate is used for the random walk to guarantee the invocation coverage of all nodes in the graph. In our experiments, 10,000 invocation sequences are generated for each setting of node numbers (e.g., 100, 1,000, and 10,000 nodes). Five types of fault-tolerance mechanisms (i.e., NoFT, RandomFT, FTCloud1, FTCloud2, AllFT) are applied on these invocation sequences, and the average results are reported in Table 2.

In Table 2, *Component FP* represents the failure probability of the cloud components, Top-K ($\mathrm{K} = 1, 5, 10$, and 20 percent) indicates that fault-tolerance mechanisms are applied for the K percent components (K percent most significant components in FTCloud1 and FTCloud2, and K percent randomly selected components in RandomFT). The experimental results in Table 2 show that,

- Among the four approaches, AllFT provides the best failure probability performance (smallest failure probability values) while NoFT provides the worst failure probability performance. Because AllFT employs fault-tolerance strategies for all the components while NoFT provides no fault-tolerance strategies for the components.
- Compared with RandomFT, FTCloud1, and FTCloud2 obtains better failure probability performance in all experimental settings. This experimental result indicates that tolerating failures of the significant components can achieve better system reliability than tolerating failures of randomly selected components. This is because the significant components identified by FTCloud1 and FTCloud2 are invoked more frequently and their failures have greater impact on the whole system.
- Compared with FTCoud1 and FTCloud2 obtain better failure probability performance in all experimental settings. This experimental result indicates that the component ranking achieves more accurate results when fusing the prior knowledge of critical components as well as the system structure information.
- When the Top-K value increases from 5 to 20 percent, the failure probabilities performance of FTCloud1 and FTCloud2 decrease monotonically, while failure probability of RandomFT may or may not decrease. This observation indicates that by tolerating failures of more components (set Top-K to be a larger value), the system reliability can be improved by employing FTCloud1 and FTCloud2 approaches.
- With the increase of the node number from 100 to 10,000, the failure probability performance of NoFT increases, since a larger system is easier to fail in error-prone environments than a smaller one. FTCloud1 and FTCloud2 can consistently provide better performance compared with RandomFT with different node numbers, indicating that by tolerating faults of the important components, the system reliability can be greatly improved for different sizes of cloud applications.
- With the increase of the component failure probability ranging from 0.1 to 1 percent, the failure

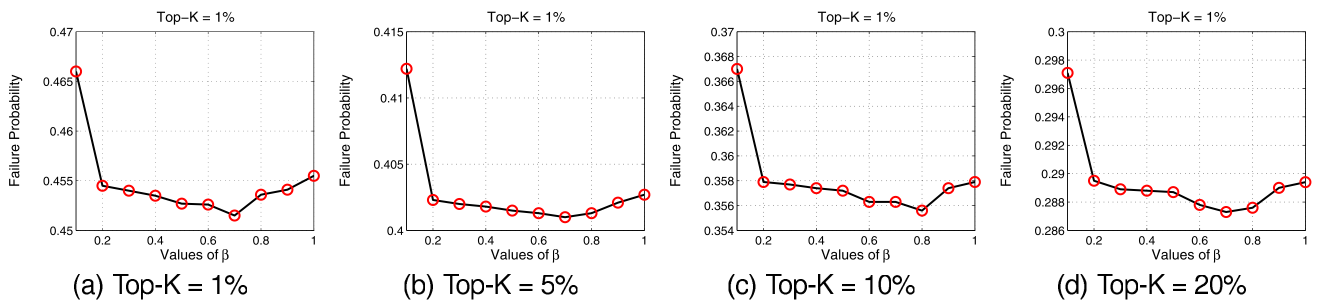


Fig. 5. Impact of β in FTCloud2.

probability for all the approaches is greatly increased. Because 1 percent component failure probability makes the application execution fail easily. In this case, we need to tolerate more significant components to provide highly reliable systems.

5.4 Impact of β in FTCloud2

In the component ranking approach FTCloud2, the parameter β determines how we incorporate the basic value of each component based on characteristics of critical or noncritical component. When $\beta = \frac{|C|}{n}$, FTCloud2 is equal to FTCloud1, which ranks the components by only employing the system structure information. To study the impact of parameter β on the ranking results, we employ 1,000 components, which includes 100 critical components and 900 noncritical components. We study the fault-tolerance performance of FTCloud2 under different Top-K values and β values ($\frac{|C|}{n} \leq \beta \leq 1$ and $\frac{|C|}{n} = 0.1$ in this experiment).

The experimental results in Fig. 5 show that

- With the increase of β value from 0.1 to 1, the system failure probability first decreases and then increases slightly, indicating that a suitable β value setting can achieve better component ranking and system fault-tolerance performance. In this experiment, optimal fault-tolerance performance can be achieved when the value of β is around 0.7.
- In all the four figures with different Top-K value settings, the failure probability of $\beta > 0.1$ is better than $\beta = 0.1$, indicating that considering prior knowledge on critical and noncritical components can obtain better fault-tolerance performance.
- With the increase of Top-K value from 1 to 20 percent, the system failure probability becomes smaller, since faults of more significant system components are masked by the fault-tolerance mechanism.

5.5 Impact of Top-K

To study the impact of the parameter Top-K on the system reliability, we compare RandomFT, FTCloud1, FTCloud2 with different Top-K value settings. The node number in this experiment is 1,000. Table 3 shows the experimental results of application failure probabilities under different Top-K value settings.

Table 3 shows that

- Under different component failure probability settings (i.e., 0.1, 0.5, and 1 percent), FTCloud1, and FTCloud2 consistently outperform RandomFT from $\text{Top-K} = 1\%$ to $\text{Top-K} = 90\%$. The performance of FTCloud1, FTCloud2, and RandomFT is the same in $\text{Top-K} = 100\%$, since fault-tolerance strategies are applied to all the components in all the three approaches in this experimental setting.
- With the increase of Top-K value, the failure probability of FTCloud1 and FTCloud2 decreases much faster than that of RandomFT, indicating that the system reliability can be improved by tolerating faults of the significant components suggested by our component ranking approaches.
- FTCloud2 obtains a smaller failure probability than FTCloud1 consistently under different Top-K value settings.
- With the increase of component failure probability from 0.1 to 1 percent, the system failure probability becomes larger, which is mainly caused by the failures of the components without any fault-tolerance strategies. Larger Top-K value is required to achieve good application failure probability

TABLE 3
Impact of Top-K on Application Failure Probability

Component FP	Methods	Values of Top-K									
		10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.1%	RandomFT	0.477	0.441	0.417	0.356	0.308	0.276	0.234	0.162	0.096	0.001
	FTCloud1	0.369	0.299	0.243	0.192	0.147	0.107	0.070	0.042	0.018	0.001
	FTCloud2	0.357	0.289	0.233	0.184	0.141	0.102	0.069	0.039	0.017	0.001
0.5%	RandomFT	0.817	0.799	0.781	0.751	0.705	0.665	0.589	0.511	0.338	0.025
	FTCloud1	0.749	0.684	0.620	0.547	0.467	0.379	0.285	0.193	0.099	0.025
	FTCloud2	0.737	0.673	0.607	0.534	0.455	0.367	0.276	0.186	0.099	0.025
1%	RandomFT	0.904	0.889	0.876	0.859	0.832	0.808	0.767	0.699	0.528	0.089
	FTCloud1	0.858	0.816	0.769	0.713	0.643	0.561	0.461	0.344	0.214	0.089
	FTCloud2	0.851	0.808	0.760	0.703	0.633	0.549	0.447	0.334	0.208	0.089

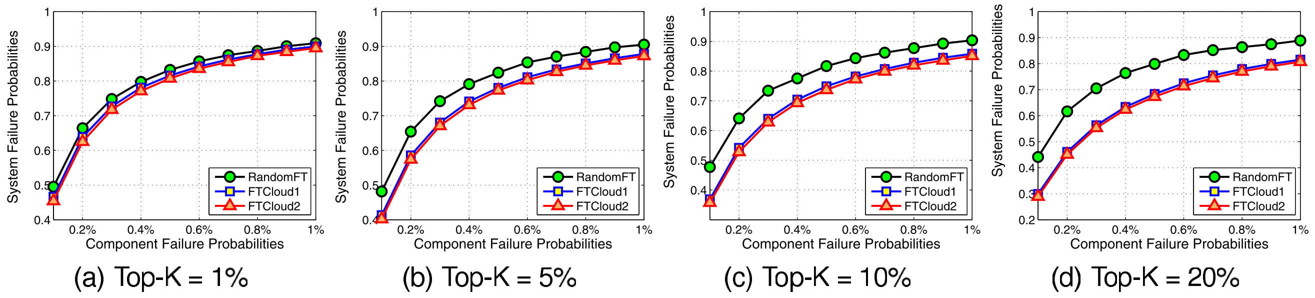


Fig. 6. Impact of component failure probability.

performance under large failure probability settings. The experimental results show that the optimal Top-K value is influenced by the component failure probability.

5.6 Impact of Component Failure Probability

To study the impact of the component failure probability on the system reliability, we compare RandomFT, FTCloud1, and FTCloud2 under failure probability settings of 0.1 to 1 percent with a step value of 0.1 percent. The node number in this experiment is 1,000. Fig. 6 shows the experimental results of cloud application failure probabilities under different Top-K settings.

Fig. 6 illustrates that

- As shown in Figs. 6a, 6b, 6c, and 6d, under different Top-K values, FTCloud1, and FTCloud2 outperform RandomFT in all the component failure probability settings from 0.1 to 1 percent consistently.
- With the increase of component failure probability from 0.1 to 1 percent, the application failure probabilities of all the three approaches become larger. A larger Top-K value is required to build reliable cloud applications under larger component failure probability settings.
- With the increase of Top-K value, the application failure probability of FTCloud1 and FTCloud2 approach decreases much faster than that of RandomFT, indicating that FTCloud1 and FTCloud2 have a more effective use of the redundant components than RandomFT.

The above experimental results show, again, that FTCloud2 achieves the best failure probability performance under different experimental settings.

6 Related Work and Discussion

The main approaches to build reliable software systems include fault prevention, fault removal [33], fault tolerance [20], [39], and fault forecasting [12], [36]. *Software fault tolerance* is widely employed for building reliable distributed systems [13]. The major software fault-tolerance techniques include recovery block [25], N-Version Programming (NVP) [5], N self-checking programming [18], distributed recovery block [17], and so on. The major fault-tolerance strategies can be divided into passive strategies and active strategies [40]. Passive strategies have been discussed in FT-SOAP [11] and FT-CORBA [31], while active strategies have been investigated in FTWeb [29], Thema [23], WS-Replication [28], SWS [19], and Perpetual [24]. In the cloud-computing environment, the abundant resources make software fault tolerance a feasible approach for building reliable cloud applications. Complementary to the previous research efforts which are mainly focused on the design of fault-tolerance strategies, this paper proposes an extensible framework for building fault-tolerant cloud applications.

Component ranking is an important research problem in cloud computing [41], [42]. The component ranking approaches of this paper are based on the intuition that components which are invoked frequently by other important components are more important. Similar ranking approaches include Google PageRank [7] (a ranking algorithm for web page searching) and SPARS-J [16] (software product retrieving system for Java). Different from the PageRank and SPARS-J models, component invocation frequencies as well as component characteristics are explored in our approaches. Moreover, the target of our approach is identifying significant components for cloud applications instead of web page searching (PageRank) or reusable code searching (SPARS-J).

Cloud computing [3] is becoming popular. A number of works have been carried out on cloud computing, including storage management [32], resource allocation [9], workload balance [34], dynamic selection [14], etc. In recent years, a great number of research efforts have been performed in the area of service component selection and composition [30]. Various approaches, such as QoS-aware middleware [38], adaptive service composition [2], efficient service selection algorithms [37], reputation conceptual model [22], and Bayesian network-based assessment model [35] have been proposed. Some recent efforts also take subjective information (e.g., provider reputations, user requirements, etc.) to enable more accurate component selection [27]. Instead of employing nonfunctional performance (e.g., QoS values) or functional capabilities, our approaches rank the cloud components considering component invocation relationship, invocation frequencies, and component characteristics.

The FTCloud framework is mainly designed for cloud applications, since

1. Cloud applications are usually large scale, involving a large number of distributed components. It is time consuming to identify significant components manually. Automatically component ranking approaches become important, which provide valuable information for application designers.
2. Resources (e.g., virtual machines) scale up and scale down dynamically on demand are main feature of cloud computing. Therefore, the cloud application structures may be dynamically updated at runtime.

In this highly dynamic context, automatically significant component identification and fault tolerance become necessary.

3. There are a lot of software/hardware resources in the cloud which can be used on demand. Redundant components are easier to be obtained in the cloud environment. Therefore software fault tolerance becomes a feasible approach for building reliable cloud application.
4. The global information of component invocation structures and invocation frequencies can be obtained since the components are all running on the same cloud.

7 Conclusion

This paper proposes a component ranking framework for fault-tolerant cloud applications. In our proposed component ranking algorithms, the significance value of a component is determined by the number of components that invoke this component, the significance values of these components, how often the current component is invoked by other components, and the component characteristics. After finding out the significant components, we propose an optimal fault-tolerance strategy selection algorithm to provide optimal fault-tolerance strategies to the significant components automatically, based on the user constraints. The experimental results show that our FTCloud1 and FTCloud2 approaches outperform other approaches.

Our current FTCloud framework can be employed to tolerate crash and value faults. In the future, we will investigate more types of faults, such as Byzantine faults. Different types of fault-tolerance mechanisms can be added into our FTCloud framework easily without fundamental changes. We will also investigate more component ranking algorithms and add them to the FTCloud framework. Moreover, we will extended and applied our FTCloud framework to other component-based systems.

In this paper, we only study the most representative type of software component graph, i.e., scale-free graph. Since different applications may have different system structures, we will investigate more types of graph models (e.g., small-world model, random-graph model, etc.) in our future work.

Our future work also includes

1. considering more factors (such as invocation latency, throughput, etc.) when computing the weights of invocations links;
2. investigating the component reliability itself besides the invocation structures and invocation frequencies;
3. more experimental analysis on real-world cloud applications (e.g., employing the NASA NPB benchmark);
4. more investigations on the component failure correlations; and
5. more experimental studies on impact of incorrectness of prior knowledge on the invocation frequencies and critical components.

Acknowledgments

The work described in this paper was fully supported by the National Basic Research Program of China (973 Project No. 2011CB302603), the National Natural Science Foundation of China (Project No. 61100078), and the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 415311 of the General Research Fund and Project No. N CUHK405/11 of the NSFC/RGC Joint Research Scheme).

References

[1] "Cloud Computing in Wikipedia," http://en.wikipedia.org/wiki/Cloud_computing, 2012.
[2] D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 369-384, June 2007.
[3] M. Armbrust et al., "A View of Cloud Computing," *Comm. ACM*, vol. 53, no. 4, pp. 50-58, 2010.
[4] M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," Technical Report EECS-2009-28, Electrical Eng. and Computer Science Dept., Univ. of California, Berkeley, 2009.
[5] A. Avizienis, "The Methodology of N-Version Programming," *Software Fault Tolerance,* M.R. Lyu, ed., pp. 23-46, Wiley, 1995.
[6] V. Batagelj and A. Mrvar, "Pajek - Program for Large Network Analysis," *Connections,* vol. 21, pp. 47-57, 1998.
[7] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Proc. Int'l Conf. World Wide Web,* 1998.
[8] M. Creeger, "Cloud Computing: An Overview," *ACM Queue,* vol. 7, no. 5, p. 2, June 2009.
[9] A. Danak and S. Mannor, "Resource Allocation with Supply Adjustment in Distributed Computing Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '10),* pp. 498-506, 2010.
[10] A.P.S. de Moura, Y.-C. Lai, and A.E. Motter, "Signatures of Small-World and Scale-Free Properties in Large Computer Programs," *Physical Rev. E,* vol. 68, pp. 017102.1-017102.4, 2003.
[11] C.-L. Fang, D. Liang, F. Lin, and C.-C. Lin, "Fault Tolerant Web Services," *J. System Architecture,* vol. 53, no. 1, pp. 21-38, 2007.
[12] S.S. Gokhale and K.S. Trivedi, "Reliability Prediction and Sensitivity Analysis Based on Software Architecture," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '02),* pp. 64-78, 2002.
[13] S. Gorender, R.J. de Araujo Macedo, and M. Raynal, "An Adaptive Programming Model for Fault-Tolerant Distributed Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 4, no. 1, pp. 18-31, Jan.-Mar. 2007.
[14] A. Goscinski and M. Brock, "Toward Dynamic and Attribute-Based Publication, Discovery and Selection for Cloud Computing," *Future Generation Computer Systems,* vol. 26, no. 7, pp. 947-970, 2010.
[15] D. Hyland-Wood, D. Carrington, and Y. Kaplan, "Scale-Free Nature of Java Software Package, Class and Method Collaboration Graphs," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 439-446, 2005.
[16] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 213-225, Mar. 2005.
[17] K. Kim and H. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," *IEEE Trans. Computers,* vol. 38, no. 5, pp. 626-636, May 1989.
[18] J. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *Computer,* vol. 23, no. 7, pp. 39-51, July 1990.
[19] W. Li, J. He, Q. Ma, I.-L. Yen, F. Bastani, and R. Paul, "A Framework to Support Survivable Web Services," *Proc. IEEE 19th Int'l Symp. Parallel and Distributed Processing,* p. 93b, 2005.
[20] M.R. Lyu, *Software Fault Tolerance,* Wiley, 1995.
[21] M.R. Lyu, *Handbook of Software Reliability Engineering.* McGraw-Hill, 1996.
[22] E. Maximilien and M. Singh, "Conceptual Model of Web Service Reputation," *ACM SIGMOD Record,* vol. 31, no. 4, pp. 36-41, 2002.
[23] M.G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-Fault-Tolerant Middleware Forweb-Service Applications," *Proc. IEEE 24th Symp. Reliable Distributed Systems (SRDS '05),* pp. 131-142, 2005.
[24] S.L. Pallemulle, H.D. Thorvaldsson, and K.J. Goldman, "Byzantine Fault-Tolerant Web Services for N-Tier and Service Oriented Architectures," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS '08),* pp. 260-268, 2008.

[25] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," *Software Fault Tolerance,* M.R. Lyu, ed., pp. 1-21, Wiley, 1995.

[26] P. Rooney, "Microsoft's CEO: 80-20 Rule Applies to Bugs, Not Just Features," *ChannelWeb,* Oct. 2002.

[27] S. Rosario, A. Benveniste, S. Haar, and C. Jard, "Probabilistic QoS and Soft Contracts for Transaction-Based Web Services Orchestrations," *IEEE Trans. Services Computing,* vol. 1, no. 4, pp. 187-200, Oct. 2008.

[28] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris, "WS-Replication: A Framework for Highly Available Web Services," *Proc. 15th Int'l Conf. World Wide Web,* pp. 357-366, 2006.

[29] G.T. Santos, L.C. Lung, and C. Montez, "FTWeb: A Fault Tolerant Infrastructure for Web Services," *Proc. IEEE Ninth Int'l Conf. Enterprise Computing,* pp. 95-105, 2005.

[30] Q.Z. Sheng, B. Benatallah, Z. Maamar, and A.H. Ngu, "Configurable Composition and Adaptive Provisioning of Web Services," *IEEE Trans. Services Computing,* vol. 2, no. 1, pp. 34-49, Jan.-Mar. 2009.

[31] G.-W. Sheu, Y.-S. Chang, D. Liang, S.-M. Yuan, and W. Lo, "A Fault-Tolerant Object Service on Corba," *Proc. 17th Int'l Conf. Distributed Computing Systems (ICDCS '97),* pp. 393-400, 1997.

[32] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu, "Storage Management in Virtualized Cloud Environment," *Proc. IEEE Third Int'l Conf. Cloud Computing (Cloud '10),* 2010.

[33] W.-T. Tsai, X. Zhou, Y. Chen, and X. Bai, "On Testing and Evaluating Service-Oriented Software," *Computer,* vol. 41, no. 8, pp. 40-46, Aug. 2008.

[34] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis, "Nefeli: Hint-Based Execution of Workloads in Clouds," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '10),* pp. 74-85, 2010.

[35] G. Wu, J. Wei, X. Qiao, and L. Li, "A Bayesian Network Based QoS Assessment Model for Web Services," *Proc. Int'l Conf. Services Computing (SCC '07),* pp. 498-505, 2007.

[36] S.M. Yacoub, B. Cukic, and H.H. Ammar, "Scenario-Based Reliability Analysis of Component-Based Software," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '99),* pp. 22-31, 1999.

[37] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints," *ACM Trans. Web,* vol. 1, no. 1, pp. 1-26, 2007.

[38] L. Zeng, B. Benatallah, A.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Trans. Software Eng.,* vol. 30, no. 5, pp. 311-327, May 2004.

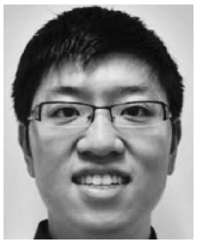
[39] Z. Zheng and M.R. Lyu, "A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services," *Proc. Sixth Int'l Conf. Web Services,* pp. 145-152, 2008.

[40] Z. Zheng and M.R. Lyu, "A QoS-Aware Fault Tolerant Middleware for Dependable Service Composition," *Proc. 39th Int'l Conf. Dependable Systems and Networks (DSN '09),* pp. 239-248, 2009.

[41] Z. Zheng, Y. Zhang, and M.R. Lyu, "CloudRank: A QoS-Driven Component Ranking Framework for Cloud Computing," *Proc. Int'l Symp. Reliable Distributed Systems (SRDS '10),* 2010.

[42] Z. Zheng, T.C. Zhou, M.R. Lyu, and I. King, "FTCloud: A Ranking-Based Framework for Fault Tolerant Cloud Applications," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '10),* 2010.

Zibin Zheng received the PhD degree from The Chinese University of Hong Kong in 2011. Currently, he is a postdoctoral fellow at The Chinese University of Hong Kong. He served as a program committee member of IEEE CLOUD 2009, CLOUDCOMPUTING 2010 and 2011, and SCC 2011. He also served as a reviewer for international journals and conferences, e.g., *IEEE Transactions on Software Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *IJCCBS*, *IJBPIM*, *IJWGS*, *Journal of Systems and Software*, *Journal of Software*, ISFI, WWW, DSN, KDD, WSDM, CloudCom, ICEBE, SCC, HASE, ISAS, P2P, QSIC, PRDC, WCNC, and SEKE. He received the ACM SIGSOFT Distinguished Paper Award at ICSE 2010, the Best Student Paper Award at ICWS 2010, the First Runner-Up Award at the IEEE Hong Kong Postgraduate Research Paper Competition, and the IBM PhD Fellowship Award 2010-2011. His research interests include service computing, cloud computing, and software reliability engineering. He is a member of the IEEE.



Tom Chao Zhou received the BEng degree in computer science and technology from the College of Computer Science, Zhejiang University, in 2008. Currently, he is a PhD candidate in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. He serves as a program committee member for the first Workshop of Machine Learning for Social Computing, held in conjunction with NIPS 2010. He has also served as a reviewer for international journals and conferences, e.g., WSDM, CIKM, QSIC, WWW, ACML, and SocialCom. His research interests include information retrieval, machine learning, and data mining. He is a student member of the IEEE.

Michael R. Lyu received the BS degree in electrical engineering from National Taiwan University, the MS degree in computer engineering from the University of California, Santa Barbara, and the PhD degree in computer science from the University of California, Los Angeles, in 1981, 1985, and 1988, respectively. Currently, he is a professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong, and the director of the Video over Internet and Wireless (VIEW) Technologies Laboratory. He previously worked at the Jet Propulsion Laboratory, the Department of Electrical and Computer Engineering at the University of Iowa, Bell Communications Research (Bellcore), and Bell Laboratories. 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 more than 270 refereed journal and conference papers, has participated in more than 30 industrial projects, and helped develop many commercial systems and software tools. He was the editor of two book volumes: *Software Fault Tolerance* (Wiley, 1995) and *The Handbook of Software Reliability Engineering* (IEEE and New McGraw-Hill, 1996). He received Best Paper Awards at ISSRE 1998 and ISSRE 2003 and initiated the First International Symposium on Software Reliability Engineering (ISSRE) in 1990. He has served as a chair or committee member for many international conferences, is a frequently invited keynote or tutorial speaker, and has served on the editorial boards of several top journals. He is a fellow of the IEEE and AAAS and a Croucher Senior Research Fellow.

Irwin King received the BSc degree in engineering and applied science from the California Institute of Technology in 1984 and the MSc and PhD degrees in computer science from the University of Southern California in 1988 and 1993, respectively. He joined the Chinese University of Hong Kong in 1993. He serves on the editorial boards of the *IEEE Transactions on Neural Networks*, the *Open Information Systems Journal*, *Journal of Nonlinear Analysis and Applied Mathematics*, and the *Neural Information Processing Letters and Reviews Journal.* He also served as special issue guest editor for *Neurocomputing*, the *International Journal of Intelligent Computing and Cybernetics*, the *Journal of Intelligent Information Systems*, and the *International Journal of Computational Intelligent Research.* Currently, he serves on the Neural Network Technical Committee and the Data Mining Technical Committee of the IEEE Computational Intelligence Society. He is also the vice president the Asian Pacific Neural Network Assembly. He has served as a reviewer or committee member for many international conferences and journals. His research interests include machine learning, information retrieval, web intelligence and social computing, and multimedia processing. He has published more than 140 refereed journal and conference papers, as well as 20 book chapters and edited volumes. He has more than 30 research and applied grants. One notable system he developed is CUPIDE (Chinese University Plagiarism IDentification Engine). He is a senior member of the IEEE and a member of the ACM, the International Neural Network Society, and the Asian Pacific Neural Network Assembly.