2010 IEEE 21st International Symposium on Software Reliability Engineering

FTCloud: A Component Ranking Framework for Fault-Tolerant Cloud Applications

Zibin Zheng, Tom Chao Zhou, Michael R. Lyu, and Irwin King
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong, China
{zbzheng,czhou,lyu,king}@cse.cuhk.edu.hk

Abstract

Cloud computing is becoming a mainstream aspect of information technology. The cloud applications are usually large-scale, complex, and include a lot of distributed components. Providing highly reliable cloud applications is a challenging and critical research problem. To attack this challenge, we propose FTCloud which is a component ranking based framework for building fault-tolerant cloud applications. FTCloud employs the component invocation structures and the invocation frequencies to identify the significant components in a cloud application. An algorithm is proposed to automatically determine optimal fault tolerance strategy for these significant components. The experimental results show that by tolerating faults of a small part of the most significant components, the reliability of cloud application can be greatly improved.

***Keywords*-Cloud application; fault tolerance; component ranking;**

I. Introduction

Cloud computing is a style of computing, in which resources (e.g., infrastructure, platform, and service) are sharing among the cloud service consumers, cloud partners, and cloud vendors in the cloud value chain [1], [2]. Strongly promoted by the leading industrial companies (e.g., Microsoft, Google, IBM, Amazon, etc.), cloud computing is becoming increasingly popular in recent years. Applications running on such cloud environment are taken as cloud applications. Cloud applications, which involve a number of distributed components, are usually large-scale and very complex. Before enterprises transfer their critical systems to the cloud environment, one question they ask is: *Can clouds become as reliable as the power grid achieving 99.999% uptime?* Unfortunately, the reliability and availability of the cloud applications are still far from perfect in reality. Nowadays, the demand for highly reliable cloud applications is becoming unprecedentedly strong. Building highly reliable and available clouds is 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 [3]. The trend towards 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) exceedingly difficult. Another approach for building reliable systems, software fault tolerance [4], 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 system, there are a lot of redundant resources in the cloud environment, making software fault tolerance a feasible approach for building highly reliable cloud applications.

Since cloud applications usually involve a large number of components, it is too expensive to provide redundant alternative components for all the cloud components. To reduce the cost and to develop highly reliable cloud applications within the limited budget, a small set of critical components need to be identified from the cloud applications. Microsoft reported that by fixing the top 20% of the most reported bugs of Windows and Office, 80% of the failures and crashes would be eliminated [6]. Our idea is also based on this well-known 80-20 rules, i.e.,

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.28

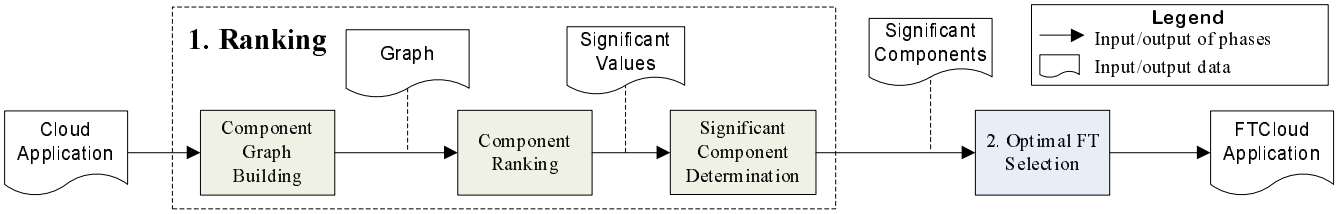

Figure 1. System Architecture of FTCloud

by tolerating faults of a small part of the most significant 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 the optimal fault tolerance strategies for the 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 two-fold:

- This paper identifies the critical problem of locating significant components in complex cloud applications and proposes a ranking based framework to build fault-tolerant cloud applications. We first propose a ranking algorithm 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. To the best of our knowledge, FTCloud is the first systematic 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 the system architecture, Section 3 proposes a ranking algorithm 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.

II. System Architecture

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

(1) The initial architecture design of a cloud application is provided by the system designer and a component graph is built for the cloud application.
(2) A component ranking algorithm is employed to calculate the significance values of the cloud components. Based on the significance values, the components can be ranked.
(3) The most significant components in the cloud application are identified.
(4) The performance of various fault tolerance strategy candidates is calculated and the most suitable strategy is selected for each significant component.
(5) The improved design of the provided cloud application and the component ranking results are returned to the system designer.

Technical details of our significance component ranking algorithm will be introduced in Section III, and the optimal fault tolerance strategy selection algorithm will be introduced in Section IV.

III. Significant Component Ranking Algorithm

The target of our significant component ranking algorithm is to measure the importance of cloud components based on component invocation relationships and invocation frequencies. The design of this ranking algorithm is based on the intuition that components which are invoked frequently by a lot of other important components are more important, since the failures of these components have greater impact to the whole system compared with normal components. As shown in Figure 1, our significant component ranking algorithm includes three steps (i.e., component graph building, component ranking, and significant component determination), which will be described in Section III-A to Section III-C, respectively.

A. 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:

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

where frq_{ij} is the invocation frequency of component c_j by component c_i and n is the number of components invoked by c_i. 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 Eq. (1) to calculate the invocation weight values. 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. In the case that a node c_i has no outgoing edge, $w_{ij} = 1/n$. For $\forall i$, the transition probability matrix W satisfies:

$$\forall i, \sum_{j \in M(c_i)} w_{ij} = 1, \quad (2)$$

where $M(c_i)$ is a set of nodes that c_i invokes.

B. 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 normal components. Intuitively, the *significant components* in a cloud application are the ones which have many invocation links coming in from other important components. Inspired by the PageRank algorithm [7], we propose an algorithm to measure the significance values of the cloud components as follows:

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:

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

where n is the number of components and $N(c_i)$ is a set of components that invoke component c_i. The parameter d ($0 \leq d \leq 1$) in Eq. (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 Eq. (3), a component c_i has larger significant 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, Eq (3) can be written as:

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

where W^t is the transposed matrix of the transition probability matrix W, which has been defined in Section III-A.

4. Solve Eq. (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 (larger significance value) if it is invoked frequently by a lot of other significant components.

C. 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., the fault tolerance techniques which will be introduced in Section IV) to improve the reliability and performance of these significant components.

IV. Optimal Fault Tolerance Strategy Selection

A. 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 [8] is a well-known mechanism employed in software fault tolerance. A recovery block is a means of structuring redundant program modules, where standby components will be invoked sequentially if the primary

Table I. Fault Tolerance Strategy Comparison

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

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:

$$f = \prod_{i=1}^{n} f_i, \tag{5}$$

where n is the number of redundant components and f_i is the failure probability of the i^{th} 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]. 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:

$$f = \sum_{i=\frac{n+1}{2}}^{n} F(i), \tag{6}$$

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 $F(2) = f_1 f_2 (1 - f_3) + f_1 (1 - f_2) f_3 + (1 - f_1) f_2 f_3$ and $F(3) = f_1 f_2 f_3$. In other words, an NVP module fails only if more than half of the redundant components fail.

- **Parallel**. 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:

$$f = \prod_{i=1}^{n} f_i, \tag{7}$$

where n is the number of redundant components and f_i is the failure probability of the i^{th} component.

Different fault tolerance strategies have different features. As shown in Table I, the response time performance of the RB and NVP strategy is not good compared with the Parallel strategies, since RB strategy invokes standby component sequentially when the primary component fails, NVP strategy needs to waiting 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 the RB, NVP, and Parallel strategies can tolerate crash faults (e.g., component crash, communication link crash, etc.). The 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 III, 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 IV-B.

B. Optimal FT Strategy Selection

The fault tolerance strategies have a number of variations based on different configurations. For example, both the RB and Parallel strategy have $n - 1$ variations (i.e., configured with 2, 3, ..., n redundant components), where n is the maximal number of redundant components. The NVP strategy has $(n-1)/2$ variations (i.e., NVP with 3, 5, ..., 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 fault tolerance strategy, the designer can specify constraints (e.g., *response-time of the component has to be smaller than 1000 milli-seconds*, 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:

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

In Problem 1, x_i is set to 1 if the i^{th} 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 added easily in the future.

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 ρ.

```
m: number of candidates;
for (i = 1; i ≤ m; i++) do
    if (s_i ≤ u_1 && t_i ≤ 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;
ρ = x;
```

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 IV-A. 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 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.

V. Experiments

In this section, Section 5.1 shows the prototype implementation, Section 5.2 introduces the experimental setup, Section 5.3 conducts extensive experiments to compare the performance of our approach with other three approaches, Section 5.4 and Section 5.5 investigate the impact of the parameter Top-K and the values of failure probability on the fault tolerance performance, respectively.

A. Implementation

A prototype of FTCloud is implemented. As shown in Figure 2, 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 Equation 1, which has been introduced in Section III-A.
- *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 algorithm in Section III-B is 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.
- *FT strategy selection*: The optimal fault tolerance strategy selection algorithm presented in Section IV-B 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.

B. Experimental Setup

Our significant component ranking algorithm is implemented by C++ language. To study the performance of reliability improvement, we compare four approaches, which are:

- **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 components, which are randomly selected.
- **FTCloud**: Fault tolerance strategies are employed to mask faults of the top k significant components.
- **AllFT**: Fault tolerance strategies are employed for all cloud components.

A scale-free graph is a graph whose degree distribution follows a power law. Many empirically observed networks

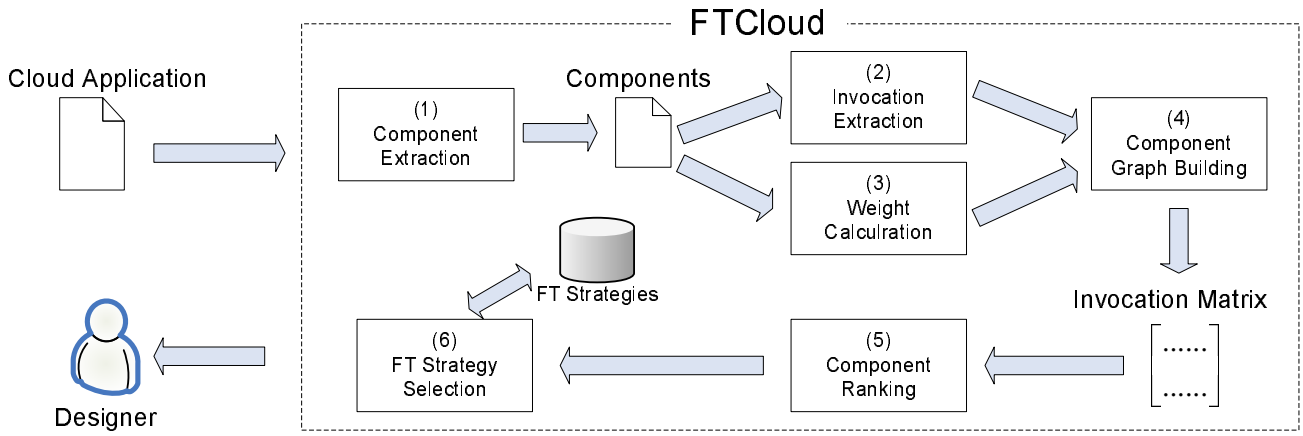

Figure 2. Implementation of FTCloud

Table II. Performance Comparison of Failure Probability

Node Numbers	Methods	Component FP = 1%			Component FP = 5%			Component FP = 10%		
		Top-5	Top-10	Top-20	Top-5	Top-10	Top-20	Top-5	Top-10	Top-20
100	NoFT	0.134	0.134	0.134	0.251	0.251	0.251	0.329	0.329	0.329
	RandomFT	0.133	0.122	0.123	0.246	0.242	0.244	0.322	0.312	0.321
	FTCloud	**0.098**	**0.021**	**0.018**	**0.229**	**0.145**	**0.143**	**0.317**	**0.288**	**0.254**
	AllFT	0.002	0.002	0.002	0.053	0.053	0.053	0.122	0.122	0.122
1000	NoFT	0.258	0.258	0.258	0.448	0.448	0.448	0.516	0.516	0.516
	RandomFT	0.245	0.231	0.227	0.434	0.425	0.429	0.508	0.502	0.499
	FTCloud	**0.177**	**0.039**	**0.032**	**0.397**	**0.196**	**0.187**	**0.493**	**0.390**	**0.379**
	AllFT	0.005	0.005	0.005	0.104	0.104	0.104	0.256	0.256	0.256
10000	NoFT	0.505	0.505	0.505	0.817	0.817	0.817	0.888	0.888	0.888
	RandomFT	0.478	0.479	0.462	0.815	0.798	0.790	0.884	0.872	0.869
	FTCloud	**0.354**	**0.036**	**0.029**	**0.740**	**0.304**	**0.271**	**0.851**	**0.604**	**0.578**
	AllFT	0.009	0.009	0.009	0.218	0.218	0.218	0.505	0.505	0.505

appear to be scale-free, including the protein networks, citation networks, and some social networks. Several previous work [9], [10] show that the internal structures of software programs (e.g., class collaboration graphs, call graphs for procedural code, inter-package dependency of applications, etc.) exhibit approximate scale-free properties. We use Pajek [11] 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 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 our approach, the parameter d balances the significance value derived from 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 Eq. (3) from 0.75 to 0.95. Therefore, similar to the work [7], [12], we also set the parameter d to be 0.85.

C. Performance Comparison

We employ random walk to simulate the invocation behavior in cloud applications. Specifically, Pajek [11] is employed to generate scale-free directed component graphs, and the edge weight is used to simulate the invocation probability. A node in the invocation graph is randomly selected, and a random walk is performed starting from the selected 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 number of nodes (e.g. 100). Four types of fault tolerance mechanisms (i.e. NoFT, RandomFT, FTCloud, AllFT) are applied on these invocation sequences, and the average result is reported in Table II .

In Table II, *Component FP* represents the failure proba-

Table III. Impact of Top-K on Application Failure Probability

Component FP	Methods	Values of Top-K										
		1	2	4	8	16	32	64	128	256	512	1024
1%	RandomFT	0.421	0.412	0.413	0.395	0.407	0.43	0.405	0.398	0.330	0.260	0.011
	FTCloud	0.393	0.367	0.336	0.183	0.053	0.038	0.037	0.024	0.024	0.016	0.011
3%	RandomFT	0.621	0.632	0.613	0.614	0.605	0.577	0.574	0.585	0.577	0.567	0.068
	FTCloud	0.607	0.585	0.553	0.403	0.166	0.125	0.129	0.13	0.106	0.099	0.068
5%	RandomFT	0.677	0.678	0.685	0.681	0.678	0.670	0.671	0.664	0.639	0.579	0.168
	FTCloud	0.673	0.651	0.619	0.487	0.272	0.255	0.233	0.211	0.215	0.191	0.168
10%	RandomFT	0.775	0.770	0.788	0.761	0.766	0.768	0.750	0.737	0.713	0.689	0.396
	FTCloud	0.757	0.756	0.734	0.677	0.567	0.552	0.530	0.510	0.477	0.472	0.396

bility of the cloud components, Top-K (K = 5, 10, and 20) indicates that fault tolerance mechanisms are applied for K components (K most significant components in FTCloud and K randomly selected components in RandomFT). The experimental results in Table II 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, FTCloud 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 FTCloud are invoked more frequently and their failures have greater impact on the whole system.
- When the Top-k value increases from 5 to 20, the failure probability performance of FTCloud decreases monotonically, while RandomFT may or may not decreas the failure probabilities. This observation indicates that the by tolerating failures of more components (set Top-K to be a larger value), the system reliability can be improved by employing the FTCloud approach.
- With the increase of the node number from 100 to 10,000, the failure probability performance of NoFT increases, since larger system is easier to fail in error-prone environments. FTCloud can consistently provide better performance compared with RandomFT with different node numbers, indicating that by tolerating a small part of important components, the system reliability can be greatly improved for different scale cloud applications.
- With the increase of the component failure probability from 1% to 10%, the execution failure probability for all the approaches are greatly increased. Because 10% component failure probability makes the application execution fail easily. In this case, only tolerating faults of the Top-20 significant components is not enough to provide a highly reliable system.

D. Impact of Top-K

To study the impact of the parameter Top-K on the system reliability, we compare FTCloud with RandomFT with different Top-K value settings. The node number in this experiment is 1024. Table III shows the experimental results of application failure probabilities under different Top-K value settings. Table III shows that:

- Under different component failure probability settings (i.e., 1%, 3%, 5%, and 10%), FTCloud consistently outperforms RandomFT in the from Top-K = 1 to Top-K = 512. The performance of FTCloud and RandomFT is the same in Top-K = 1024, since fault tolerance strategies are applied to all the components in both FTCloud and RandomFT in this experimental setting.
- With the increase of Top-K value, the failure probability of FTCloud decreases much faster than RandomFT. For example, when *Component FP = 1%*, FTCloud provides good failure probability results with Top-16 (i.e., 0.053), while RandomFT provides poor failure probability performance (i.e., 0.407).
- With the increase of component failure probability from 1% to 10%, 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 performance under large failure probability settings. The experimental results show that the optimal Top-K value is influenced by the component failure probability.

E. Impact of Failure Probability

To study the impact of the component failure probability on the system reliability, we compare FTCloud with

Table IV. Impact of Component Failure Probability on Application Failure Probability

Top-K	Methods	Component Failure Probability									
		1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
Top-5	RandomFT	0.407	0.550	0.637	0.663	0.684	0.704	0.726	0.757	0.760	0.766
	FTCloud	0.304	0.434	0.523	0.573	0.617	0.664	0.670	0.694	0.718	0.734
Top-10	RandomFT	0.415	0.548	0.612	0.664	0.683	0.712	0.736	0.736	0.756	0.767
	FTCloud	0.046	0.098	0.175	0.236	0.311	0.356	0.442	0.488	0.513	0.585
Top-20	RandomFT	0.413	0.560	0.626	0.671	0.681	0.702	0.730	0.744	0.742	0.741
	FTCloud	0.054	0.097	0.183	0.226	0.286	0.323	0.381	0.465	0.515	0.552
Top-40	RandomFT	0.404	0.523	0.631	0.656	0.675	0.701	0.728	0.742	0.761	0.758
	FTCloud	0.034	0.083	0.133	0.191	0.259	0.334	0.396	0.453	0.493	0.527

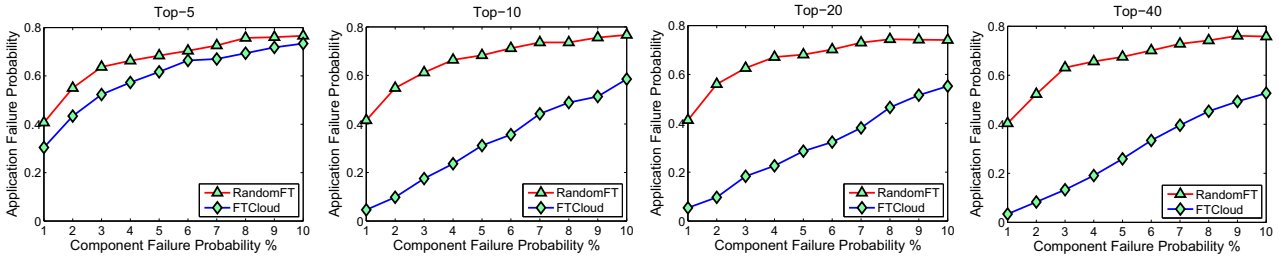

Figure 3. Impact of Component Failure Probability

RandomFT under failure probability settings of 1% to 10% with a step value of 1%. The node number in this experiment is 1024. Table IV and Figure 3 show the experimental results of cloud application failure probabilities under different Top-K settings (i.e., Top-5, Top-10, Top-20, and Top-40). Table IV and Figure 3 show that:

- As shown in Figure 3(a) to Figure 3(d), under different Top-K values, FTCloud outperforms RandomFT in all the component failure probability settings from 1% to 10% consistently.
- With the increase of component failure probability from 1% to 10%, the application failure probabilities of both RandomFT and FTCloud become larger. Larger Top-K value is required to build reliable cloud applications under large component failure probability settings.
- With the increase of Top-K value, the application failure probability of FTCloud approach decreases much faster than RandomFT, indicating that FTCloud has a better effective use of the redundant components than RandomFT.

The above experimental results show, again, that FTCloud achieves better failure probability performance than RandomFT.

VI. Related Work and Discussion

The main approaches to build reliable software systems include fault prevention, fault removal [13], fault tolerance [4], [14], and fault forecasting [15], [16], [17]. *Software fault tolerance* is widely employed for building reliable distributed systems [18], [19]. The major software fault tolerance techniques include recovery block [8], N-Version Programming (NVP) [5], N self-checking programming [20], distributed recovery block [21], and so on. The major fault tolerance strategies can be divided into passive strategies and active strategies [22], [23], [24]. Passive strategies have been discussed in FT-SOAP [25] and FT-CORBA [26], while active strategies have been investigated in FTWeb [27], Thema [28], WS-Replication [29], SWS [30], and Perpetual [31]. In the cloud computing environment, there are a lot of redundant resources available in the cloud. Alternative components are easier to be obtained to build reliable cloud applications. Complementary to the previous research efforts which are mainly focused on the design of fault tolerance strategies, we propose a systematic and extensible framework for building fault-tolerant cloud applications by component ranking.

The component ranking approach of this paper is 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 [12] (software product retrieving system for Java). Different from the PageRank and SPARS-J models, invocation frequencies of the components are explored in our approach. The target of our approach is identifying significant components for cloud applications instead of

Web page searching (PageRank) or reusable code searching (SPARS-J).

A great number of research efforts have been performed in the area of service component selection and composition. Various approaches, such as Qos-aware middleware [32], adaptive service composition [33], efficient service selection algorithms [34], reputation conceptual model [35], and Bayesian network based assessment model [36], have been proposed in recent years. Some recent work also take subjective information (e.g. provider reputations, user requirements, etc.) to enable more accurate service component selection [37], [38]. Instead of employing non-functional performance (e.g., QoS values) or functional capabilities, our approach determines the significant component by employing the component invocation structures as well as the invocation frequencies. At design time, FTCloud can be employed to achieve more reliable system design. At runtime, FTCloud can be employed for dynamically determining significant components using the updated invocation frequency information.

In this paper, we focus on cloud applications, since (1) cloud applications are usually large-scale and include a hug number of components. Identifying significant components provides valuable information to the application designers; (2) redundant components are easier to be obtained in the cloud environment, since there are a lot of software/hardware resources in the cloud which can be used on demand; (3) the global information of component invocation structures and invocation frequencies can be obtained since the components are all running on the same cloud. Nevertheless, beside cloud applications, our FTCloud framework can also be applied to a lot of other component-based systems where the component structure information is available.

VII. Conclusion and Future Work

This paper proposes a component ranking framework for fault-tolerant cloud applications. In our FTCloud approach, the significance value of a component is determined by the number of components that invoke this component, the significance values of these components, and how often the current component is invoked by other components. After finding out the significant components, we propose an optimal fault tolerance selection algorithm to provide optimal fault tolerance strategies to the significant components automatically, based on the user-constraints. The experimental results show that our FTCloud approach significantly outperforms other baseline 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 faults can be added into our FTCloud framework easily without fundamental changes. 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 open-source cloud applications; and (4) more investigations on the component failure correlations.

Acknowledgement

The authors appreciate the reviewers for their extensive and informative comments. The work described in this paper was fully supported by grants (Project No. CUHK4154/09E, CUHK4128/08E) from the Research Grants Council of the Hong Kong Special Administrative Region, China.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," *Technical Report, EECS-2009-28*, University of Texas at Dallas, 2009.

[2] M. Creeger, "Cloud computing: An overview," *ACM Queue*, vol. 7, no. 5, June 2009.

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

[4] M. R. Lyu, *Software Fault Tolerance*. Trends in Software, Wiley, 1995.

[5] A. Avizienis, "The methodology of n-version programming," *Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester*, pp. 23–46, 1995.

[6] P. Rooney, "Microsoft's ceo: 80-20 rule applies to bugs, not just features," *ChannelWeb*, October 2002.

[7] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th Int'l Conf. World Wide Web (WWW'98)*, 1998.

[8] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester*, pp. 1–21, 1995.

[9] 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 Review E*, vol. 68, no. 017102, 2003.

[10] D. Hyland-Wood, D. Carrington, and Y. Kaplan, "Scale-free nature of java software package, class and method collaboration graphs," in *Proc. 5th Int'l Symposium on Empirical Software Engineering*, 2005, pp. 439–446.

[11] V. Batagelj and A. Mrvar, "Pajek - program for large network analysis," *Connections*, vol. 21, pp. 47–57, 1998.

[12] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Trans. Software Engineering*, vol. 31, pp. 213–225, 2005.

[13] W.-T. Tsai, X. Zhou, Y. Chen, and X. Bai, "On testing and evaluating service-oriented software," *IEEE Computer*, vol. 41, no. 8, pp. 40–46, 2008.

[14] Z. Zheng and M. R. Lyu, "A distributed replication strategy evaluation and selection framework for fault tolerant web services," in *Proc. 6th Int'l Conf. Web Services (ICWS'08)*, 2008, pp. 145–152.

[15] S. S. Gokhale and K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *Proc. Int'l Symp. Software Reliability Engineering. (ISSRE'02)*, 2002, pp. 64–78.

[16] S. M. Yacoub, B. Cukic, and H. H. Ammar, "Scenario-based reliability analysis of component-based software," in *Proc. Int'l Symp. Software Reliability Engineering (ISSRE'99)*, 1999, pp. 22–31.

[17] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction for service-oriented systems," in *Proc. IEEE/ACM 32nd Int'l Conf. Software Engineering (ICSE'10)*, 2010, pp. 35–44.

[18] 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, 2007.

[19] Z. Zheng and M. R. Lyu, "Ws-dream: A distributed reliability assessment mechanism for web services," in *Proc. 38th Int'l Conf. Dependable Systems and Networks (DSN'08)*, 2008, pp. 392–397.

[20] 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, Jul 1990.

[21] 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.

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

[23] Z. Zheng and M. R. Lyu, "A qos-aware middleware for fault tolerant web services," in *Proc. Int'l Symp. Software Reliability Engineering (ISSRE'08)*, 2008, pp. 97–106.

[24] Z. Zheng and M. R. Lyu, "A qos-aware fault tolerant middleware for dependable service composition," in *Proc. 39th Int'l Conf. Dependable Systems and Networks (DSN'09)*, 2009, pp. 239–248.

[25] C.-L. Fang, D. Liang, F. Lin, and C.-C. Lin, "Fault tolerant web services," *Journal of System Architure*, vol. 53, no. 1, pp. 21–38, 2007.

[26] G.-W. Sheu, Y.-S. Chang, D. Liang, S.-M. Yuan, and W. Lo, "A fault-tolerant object service on corba," in *Proc. 17th Int'l Conf. Distributed Computing Systems (ICDCS'97)*, 1997, p. 393.

[27] G. T. Santos, L. C. Lung, and C. Montez, "Ftweb: A fault tolerant infrastructure for web services," in *Proc. 9th IEEE Int'l Conf. Enterprise Computing*, 2005, pp. 95–105.

[28] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware forweb-service applications," in *Proc. 24th IEEE Symp. Reliable Distributed Systems (SRDS'05)*, 2005, pp. 131–142.

[29] J. Salas, F. Perez-Sorrosal, n.-M. Marta Pati and R. Jiménez-Peris, "Ws-replication: a framework for highly available web services," in *Proc. 15th Int'l Conf. World Wide Web (WWW'06)*, 2006, pp. 357–366.

[30] W. Li, J. He, Q. Ma, I.-L. Yen, F. Bastani, and R. Paul, "A framework to support survivable web services," in *Proc. 19th IEEE Int'l Symp. Parallel and Distributed Processing*, 2005, p. 93.2.

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

[32] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Trans. Software Engeering*, vol. 30, no. 5, pp. 311–327, 2004.

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

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

[35] E. Maximilien and M. Singh, "Conceptual model of web service reputation," *ACM SIGMOD Record*, vol. 31, no. 4, pp. 36–41, 2002.

[36] G. Wu, J. Wei, X. Qiao, and L. Li, "A bayesian network based qos assessment model for web services," in *Proc. Int'l Conf. Services Computing (SCC'07)*, 2007, pp. 498–505.

[37] V. Deora, J. Shao, W. Gray, and N. Fiddian, "A quality of service management framework based on user expectations," in *Proc. 1st Int'l Conf. Service-Oriented Computing (ICSOC'03)*, 2003, pp. 104–114.

[38] 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, 2008.