# A Queueing-Theory-Based Fault Detection Mechanism for SOA-Based Applications

HAO-PENG CHEN
*School of Software, Shanghai Jiao Tong University, Shanghai 200240, China*
*E-mail: chen-hp@sjtu.edu.cn*

CHENG ZHANG
*School of Software, Shanghai Jiao Tong University, Shanghai 200240, China*
*E-mail: darkinparadise@sjtu.edu.cn*

## Abstract

*SOA has become more and more popular, but fault tolerance is not supported in most SOA-based applications yet. Although fault tolerance is a grand challenge for enterprise computing, we can partially resolve this problem by focusing on its some aspect. This paper focuses on fault detection and puts forward a queueing-theory-based fault detection mechanism to detect the services that fail to satisfy performance requirements. This paper also gives a reference service model and reference architecture of fault-tolerance control center of Enterprise Services Bus for SOA-based applications.*

## 1. Introduction

Service-Oriented Architecture (SOA) is an emerging approach of software system development for constructing complex distributed applications. In this approach, application developers identify desired services in service repository which can realize different parts of the functionality of an application and compose the available services into the target application through an appropriate application composition process. Therefore, instead of designing and coding, SOA-based applications are built through service discovery and composing.

One of the important characteristics of SOA-based applications that are different from traditional software is dynamic discovery and composition [1]. It means that SOA-based applications have the ability to discover and compose services into themselves at runtime, which is necessary for the applications to locate the alternative services to replace existing ones that fail to satisfy the functional or performance requirements during their execution. With dynamic discovery and composition, SOA-based applications have the capability of fault-tolerance.

Dynamic discovery and composition requires four capabilities: (a) to identify existing services that fail to satisfy the functional or performance requirements for SOA-based applications; (b) to generate queries to locate alternative services that could replace existing ones; (c) to efficiently execute these queries at runtime; (d) to dynamically replace existing services during application execution [2]. Among these capabilities, fault detection, the capability to identify the failed existing services, takes precedence over other three capabilities because successful fault detection is the guarantee for them to be employed properly. We can conclude, therefore, fault detection is the most important capability for SOA-based applications.

The research on SOA falls into two major categories:

1. Service matching, that is, discovering services that meet the requirements. The requirements are on either functionality or quality properties.
2. Service discovery efficiency that focuses on discovering services more efficiently.

In the research of service matching, semantic matching is a widely adopted approach to identify the consistency of functional requirement. [3] presents efficient service discovery and composition algorithms that exploit both syntactic and semantic service descriptions of web services. [4] introduces a semantic matching algorithm that exploits the possibility to compose multiple services in order to satisfy a service request. [5] proposes an automatic composition mechanism by using pre and post-conditions of Web services. As for the aspect of quality properties, existing approaches are mainly based on extension of service description. [6] presents an agent-based framework for web services composition, introduces a web services QoS model, and gives an algorithm of selecting services from the perspective of composition. [7] presents a generalized design and implementation of a QoS enabled Web Service discovery mechanism. The mechanism groups web services into the

The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 

predefined categories in UDDI according to the QoS description of services, and search for the desired services in the categories with key words. It brings the benefit of QoS characteristics without affecting the existing UDDI search facilities.

In the research of service discovery efficiency, some effort is put on the design of service registry center, for example, [8] presents a reliable, flexible and scalable distributed web service discovery architecture that is based on the concept of distributed shared space and intelligent search among a subset of spaces. Some research work focuses on the detail of service invocation, for example, [9] proposes the Short Circuit Switch (SCS), which is an intelligent Web services discovery and invocation engine that can be attached to deployed Web services in order to enable direct invocation instead of communication via Web services interface, when two web services happen to reside on the same runtime system.

These approaches or solutions, however, have not emphasized the capability of fault tolerance. This fact shows fault tolerance is an important issue for IT industry, but it has not completely resolved yet. There are quite a few of reasons for this situation. For example, we need an automatic verifier to verify the services dynamically discovered, but as a grand challenge, there is no automatic verifier which has been designed and developed for commercial use. Despite these reasons, some aspects of fault tolerance should and could be resolved, such as fault detection.

As we mentioned, the purpose of fault detection is to identify two kinds of services: the services that fail to satisfy functional requirements and the services fail to satisfy performance requirements. We focused on the latter kind of services, because the failure of the former kind of services may be caused by semantic errors which have to be manually identified and corrected, meanwhile, the failure of the former kind of services can be automatically detected via appropriate mechanism.

This paper puts forward a fault detection mechanism, which is based on the queueing theory, to detect the services that fail to satisfy performance requirements. This paper also gives a reference service model and reference architecture of fault-tolerance control center of ESB (Enterprise Services Bus).

## 2. Rationale of this mechanism

The purpose of our mechanism is to detect the services which have not enough capability to satisfy the performance requirements specified by application assemblers. Performance, to put it simply, is how quickly the system can respond to a given logical operation from a given individual user. Response time is a measure of the amount of time the system consumes while processing a user request, which is made up of three things: latency, which is the amount of time spent processing overhead just to get to the point of carrying out a service; wait time, which is the time spent waiting for the service, or, once the service is executing, the time spent waiting for resources; and service time, which is the time needed to process the request when no waiting is involved [11].

Since response time is an important measure of service performance, we can use it to determine whether the specified service satisfies its performance requirement or not. A fixed value of response time, however, is not suitable to do it, because response time is a typical random variable. Thus, how should we model the service requests processing? With analysis, we can find out that the service requests processing has the following features:

1. The interarrival times between any two successive service invocation requests are independent of each other and have a common distribution.
2. The clients would receive responses if requests processed by service in time, or receive exceptions due to the timeout of waiting. They even could abort the service invocation requests as their wills
3. The service times needed for every request are not only dependent on the status of services, but also identically distributed. Furthermore, they are independent of interarrival times.
4. The requests can be served in many possible orders, such as first come first served, last come first served, shortest processing time first, random order, round robin, and so on. However, the first come first served is still the predominant way.
5. There may be a single service instance or a group of service instances processing the requests. Thus, there are several possible kinds of service capacity.
6. Since the cache or buffer of the service hosting environment is finite, the number of waiting requests is limited. It means if the waiting room of a service hosting environment is fully occupied, then when extra requests arrive this service, they would be ignored.

The above six feature has shown the model of service requests processing is a typical queueing model,

so we can resort to queueing theory to establish our fault detection mechanism.

Queueing models have some relevant performance measures, such as the distribution of the waiting time and the sojourn time of a request, the distribution of the amount of work in the system, and the distribution of the busy period of the service. Note that the sojourn time is the waiting time plus the service time, which equals to response time. [12]

Let E(N) denote the mean number of requests in the service hosting environment, E(T) denote the mean sojourn time, and$\lambda$denote the average number of requests arriving the service per unit time. Under the assumption that the capacity of the system is sufficient to deal with the requests, according to the Little's law, when a queue reaches a steady state after a long running time, the relation between the three values is:

$$E(N) = \lambda E(T) \tag{1}$$

In formula (1), if we know any two of the three elements, we can compute the third one. We take advantage of this feature. [13]

We set up a monitor, which is triggered once a request is processed, to record $S_k$, the sojourn time of this request. This monitor also records $\lambda_t$ every unit time, which is the number of new arriving requests during the unit time right before time $t$. By the formula group (2), we use a scanner to scan the service instance pool every unit time to gain $L(t)$, the number of requests under processing or waiting at time $t$, and use a calculator to compute $E(N)$, $E(T)$, and $\lambda$ with the data recorded by the monitor.

$$\begin{cases} E(N) = \lim\limits_{t\to\infty} \frac{1}{t} \sum\limits_{x=0}^{t} L(x) \\ E(T) = \lim\limits_{n\to\infty} \frac{1}{n} \sum\limits_{k=1}^{n} S_k \\ \lambda = \lim\limits_{t\to\infty} \frac{1}{t} \sum\limits_{x=0}^{t} \lambda_x \end{cases} \tag{2}$$

Consequently, the values of $E(N)$, $E(T)$and$\lambda$in formula group (2) reflect the real-time status of a service. Meanwhile, $E(T')$, the theoretical value of the mean sojourn time, can be computed by formula (1). It is obviously that $E(T)$ could not always equal to $E(T')$. When the service has not enough capability to deal with all arrived requests, some requests would be ignored, and their clients would receive timeout exception, in contrast, some requests would be aborted by their clients. Therefore, it is not all requests that would be served by the service, which means $E(T)$ would be greater or smaller than $E(T')$. Thus, we can specify an acceptable nonnegative error $e$, and compare $E(T)$ with $E(T')$ to check whether the inequation (3) is true.

$$\left|E(T) - E(T')\right| \le e \tag{3}$$

If $E(T)$ and $E(T')$make inequation (3) false, it means there are too many requests to be refused or aborted in current unit time. Next, we set up a positive integer $n$, which denotes if in successive $n$ unit time, the absolute values of the differences between $E(T)$ and $E(T')$ are greater than $e$, then we consider that the service can not satisfy our performance requirement. As a result, we successfully detect a fault by our criteria.

## 3. Reference service model

According to the rationale of our fault detection mechanism, we provide a reference service model, as shown in Figure 1.

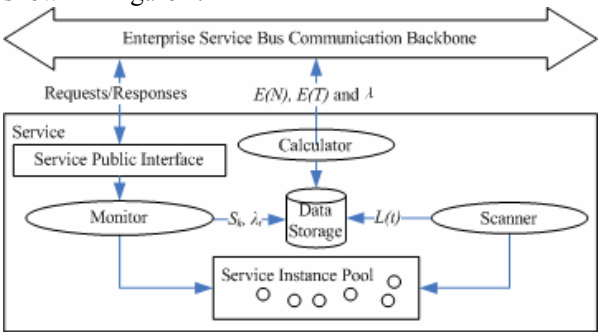


Figure 1. A reference service model

In this reference model, we add a monitor, a scanner, a data storage, and a calculator into the ordinate service model.

The clients of a service generate service invocation requests according to the service public interface, which contains all methods allowed client to invoke. These requests are sent to Enterprise Service Bus (ESB) in form of some kind of messages, such as XML messages.

When these messages arrive at the service hosting environment, the monitor intercepts all of them, records their arrival time into its cache, and updates $\lambda_t$, the number of new arrival requests stored in the data storage. Subsequently, the monitor delegates the requests to service instance pool.

The service instances pool interprets the messages into the method invocations which can be accepted by the programming language used to implement the service. It creates new instances or allocates existing

instances for the method invocations waiting in a queue. If a method invocation is accomplished, there is a response generated and sent back to client. Meanwhile, the service instances pool reclaims the invoked service instance.

When a response is be sending back to client, it is intercepted by the monitor again to read the leaving time. Subsequently, the monitor calculates $S_k$ , the sojourn time of the corresponding request, with the cached arrival time and the leaving time, and stores it into the data storage. Lastly, the response is delivered to ESB.

The scanner scans the service instances pool every unit time to gain *L(t)*, the number of requests under processing or waiting at time *t*. *L(t)* should be the sum of the length of waiting queue and the number of service instance, and stored in the data storage.

The calculator accesses the data storage to retrieve $S_k$ , $\lambda_t$ , and *L(t)*, and uses them to calculate *E(N), E(T)*and$\lambda$by formula group (2) every unit time. Finally, the calculator sends *E(N), E(T)*and$\lambda$onto ESB in order to allow the control center of ESB to receive them and subsequently to determine whether the service is failed to satisfy the performance requirements according to the real-time configured acceptable error *e* and acceptable number of successive failed unit time *n*.

Since the data stored in data storage is just several records, and it needs to be accessed frequently, it should be designed as an in-memory object, such as an in-memory table.

It is obvious that this reference service model achieves our aims in the way that has a side-effect on performance. Since the side-effect is limited, it is worthy to obtain the capability of fault detection at the cost of implementing services in this way.

## 4. Reference architecture of fault-tolerance control center of ESB

In [10], the authors abstracted the architecture of fault-tolerance control center of ESB. We extend this architecture by adding three components into it to support our fault detection mechanism. The reference architecture is shown in Figure. 2.

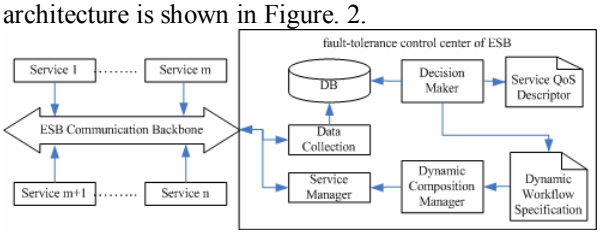


Figure 2. Reference architecture of fault-tolerance control center of ESB

In this reference architecture, the service k (k = 1,2,……,n) is implemented based on the service model in section 4, which periodically sends its *E(N), E(T)*and$\lambda$onto ESB.

The Data Collection Service, Dynamic Composition Manager, and Dynamic Workflow Specification are put forward by the authors of [10]. Data Collection Service will keep on monitoring the behaviors of the participating services and collect data at runtime [10]. In our architecture, the data collected by Data Collection Service is all *E(N), E(T)*and$\lambda$periodically sent by all services.

We add a Database to store the collected data, because the sample spaces for mean response time calculation and comparison of all services are too big to store in memory. This database also could be a lightweight data storage, such as a LDAP storage.

The Decision Maker access the Database to calculate the difference between the actual mean response time *E(T)* and the theoretical mean response time *E(T')*, and access the Service Qos Descriptor, which contains all Qos requirements of services in system, to retrieve the acceptable error and acceptable number of successive failed unit time *n*. If there have been successive *n* times that the absolute value of the difference between *E(T)* and *E(T')*is greater then *e*, a warning message will be send to Dynamic Workflow Specification, which will determine whether the workflow specification should be modified or not [10]. If the workflow specification is modified, the Dynamic Composition Manager will re-composite the workflow at runtime [10].

This architecture is a coarse-grained architecture. when applying to specific SOA-based applications, it needs to be refined and customized.

## 5. Conclusion

In this paper, we put forward a fault detection mechanism, which is based on the queueing theory, to detect the services that fail to satisfy performance requirements. We also give a reference service model and a reference architecture of fault-tolerance control center of ESB based on our fault detection mechanism.

Although queueing theory is mature, and we can prove the correctness of this mechanism, we lack of experiment data yet, because it is difficult to establish a simulation application to validate our mechanism. As we mentioned in section 1, the existing frameworks or platforms have no capability of fault tolerance, some of

The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 © 2007 IEEE

them even have no open APIs provided for us to extend them. With the emergence of open source SOA platform, we can choose a proper one to extend it to have the capability of fault detection, and establish a practical SOA-based application to validate the correctness of this mechanism in future.

## References


[1] W.T. Tsai, Chun Fan, Yinong Chen, R. Paul, and Jen-Yao Chung, "Architecture classification for SOA-based applications", Proc. of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006), April, 2006, pp. 8-15

[2] G. Spanoudakis, A. Zisman, and A. Kozlenkov, "A service discovery framework for service centric systems", Proc. of 2005 IEEE International Conference on Services Computing (SCC 2005), July 2005, pp. 251 - 259

[3] Seog-Chan Oh, Hyunyoung Kil, Dongwon Lee, and Soundar R. T. Kumara, "Algorithms for Web Services Discovery and Composition Based on Syntactic and Semantic Service Descriptions", Proceedings of the 8th IEEE International Conference on E-Commerce Technology and the 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06), June 2006, pp. 66 – 66

[4] Lerina Aversano, Gerardo Canfora, Anna Ciampi , "An algorithm for Web service discovery through their composition", Proceedings of the IEEE International Conference on Web Services (ICWS'04), July 2004, pp.332 – 339

[5] Lin Lin, I. Budak Arpinar, "Discovering Semantic Relations between Web Services Using Their Pre and Post-Conditions", Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05), July 2005, pp.237 - 238 vol.2

[6] Bin Li, Xiao-yan Tang, Jian Lv, "The Research and Implementation of Services Discovery Agent in Web Services Composition Framework", Proceedings of the Fourth International Conference on Machine Learning and Cybernetics, Guangzhou (ICMLC'05), 18-21 August 2005, Volume 1, pp.78 – 84

[7] Yannis Makripoulias, Christos Makris, Yiannis Panagis, Evangelos Sakkopoulos, Poulia Adamopoulou , Athanasios Tsakalidis, "Web Service discovery based on Quality of Service", IEEE International Conference on Computer Systems and Applications (ICCSA'06), March 2006, pp.196 – 199

[8] Brahmananda Sapkota, Dumitru Roman, Sebastian Ryszard Kruk, Dieter Fensel, "Distributed Web Service Discovery Architecture", Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), Feb. 2006, pp.136 – 136

[9] Anand Sangtani, Ravinder Pal, Jia Zhang, "Short Circuit Switch –An Intelligent Web Services Discovery and Invocation Engine", Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05), July 2005, pp.:241 - 242 vol.2

[10] S. Simmons, "Introducing the WebSphere Integration Reference Architecture: A Service-based Foundation for Enterprise-Level Business Integration", IBM WebSphere Developer Technical Journal, Aug. 17, 2005, available at: http://www-128.ibm.com/developerworks/websphere/techjournal/0508_simmons/0508_simmons.html.

[11] Ted Neward, Effective Enterprise Java, Addison Wesley Professional, Boston, August 26, 2004

[12] Ivo Adan, Jacques Resing, Queueing Theory, February 28, 2002, available at: www.win.tue.nl/~iadan/queueing.pdf

[13] AndreasWillig, A Short Introduction to Queueing Theory, July 21, 1999, available at: www.tkn.tu-berlin.de/curricula/ws0203/ue-kn/qt.pdf




0-7695-2913-5/07 $25.00 © 2007 IEEE