A Fault Detection Mechanism for Service-Oriented Architecture Based on Queueing Theory

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 is an ideal solution to application building, since it reuses the existing services as many as possible. The fault tolerance is one important capability to ensure the SOA-based applications are high reliable and available. However, fault tolerance is such a complex issue for most SOA providers that they hardly provide this capability in their products. This paper provides a queuing-theory-based algorithm to fault detection, which can be used to detect the services whose performance becomes unsatisfactory at runtime according to the Qos descriptor. Based on this algorithm, this paper also gives the reference models of the extended service and the architecture of fault-tolerance control center of Enterprise Services Bus for SOA-based applications.

Keywords

SOA, Fault Detection, Queuing Theory, Fault-Tolerant, Performance Requirement.

1. Introduction

As a burgeoning approach of software system development, Service-Oriented Architecture (SOA) shows its compelling value in the construction of complex distributed applications. By adopting this approach, developers search on Internet to discover the desired services which can provide different parts of the functionality of an application and integrate them into the target application through an appropriate composition process. Therefore, SOA-based applications are built through service discovery and composition, instead of designing and coding.

SOA resembles CBSD (Component-Based Software Development) in reuse and integration of existing components or services. Nevertheless, it is unnecessary for SOA-based application developers to get copies of all services and integrate them on local server, just as the way CBSD adopts. Instead, the developers only need get the location information of all services, such as their URLs, and the description of the all services, such as their WSDLs, to create a composition configuration file. In the ideal situation, all the services of a SOA-based application should run in their own hosting environment, and be composed at runtime by the service invocation flow described in the configuration file. In this way, different applications share the same service instances on Internet. Consequently, the computing resources are furthest leveraged.

One of the important characteristics of SOA-based applications that are different from traditional software is dynamic discovery and composition of services [1]. 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]. With the enough capability of fault tolerance, 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.

However, most of the current SOA-based applications are built in the same way as that adopted by CBSD, that is, getting copies of all services and integrating them on local server. In these applications, the advantage of SOA has been lost, so they are degraded as component-based applications. The main reason for this situation is that the existing SOA frameworks and solutions do not have enough capability of fault tolerance which is necessary for building real SOA-based applications. Actually, in [3], one of the most popular frameworks, WebSphere Architecture, even was considered as a service-oriented architecture without fault tolerance. This fact shows that fault tolerance is an important issue for IT industry, but it has not completely resolved yet.

If we consider the four capabilities required by dynamic discovery and composition, we would find

0-7695-2983-6/07 $25.00 © 2007 IEEE
DOI 10.1109/CIT.2007.135

that fault detection is the precondition for other three capabilities. Fault detection can be classifies into two kinds: to detect the services that fail to satisfy functional requirements and to detect the services fail to satisfy performance requirements. We focus on the latter kind of detection, because the failure of the former kind of detection may be caused by semantic errors which have to be manually identified and corrected. Meanwhile, the former kind of detection can be automatically executed via appropriate mechanisms.

This paper puts forward a queuing-theory-based algorithm to fault detection, which can be used to detect the services whose performance becomes unsatisfactory at runtime according to the Qos descriptor. 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 parts: 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, while 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 [4].

The response time can be used to determine whether the specified service satisfies its performance requirement or not due to it is an important measure of service performance. The basic nature of response time is that it is a random variable. So, the service requests processing is a stochastic process, which 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 are 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 time needed for every request is not only dependent on the status of services, but also identically distributed. Furthermore, they are independent of interarrival time.
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 order.
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, when extra requests arrive at this service, they would be lost.

The above six features have shown that the model of service requests processing is a typical queuing model, so we can resort to queuing theory to establish our fault detection mechanism.

Queuing 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. [5] Suppose $\xi(t)$ is the number of requests in queue at time t, thus $\xi(t)$ is a random variable, and $\{\xi(t)\}$ is a stochastic process, which can be treated as a birth and death process, in which the arrival of requests are births and the departures of requests are deaths. In theory, $\{\xi(t)\}$ will reach its steady state after running for infinite time. The steady state means that the probability distribution of the state of $\{\xi(t)\}$ will not vary with time, and the probability that its state is j will be a constant P_j. But in practice, for most questions, the corresponding birth and death processes need not to run for infinite time to reach their steady states, they can reach their steady states more quickly. [6]

According to the description above, if a service reaches its steady state, its expected number of requests waiting for serving, expected waiting time of requests, and expected sojourn time of requests will be steady. Furthermore, the distribution of these variables is independent of time; it means in any period or at any time, the distribution of these variables is the same, and the values of these variables before time t would not have impact on the values of these variables at time t. This property is called memoryless property. Thus, we can check the queue periodically to find whether it is steady.

So we add a Service QoS Descriptor in control center of Enterprise Service Bus (EBS), which contains the three variables and acceptable nonnegative error *e* and acceptable number of successive failed periods *n* of each service in system. When the application works, we continuously calculate the mean number of requests waiting for serving, the mean waiting time of requests, and the mean sojourn time of requests in a period which contains certain amount of unit time, and compare these real-time variables with the expected variables stored in Service QoS Descriptor. There must be difference between these two set of variables, especially we focus on the difference between the expected sojourn time and the real-time mean sojourn time. If the difference exceeds *e* for *n* times, we consider the corresponding service deviates its steady state, and as a result, it cannot satisfy the performance requirement any longer.

3. The algorithm this mechanism employs

There are several items to discriminate different queues, including: the distribution of the interarrival times between service invocation requests, the distribution of service time, and the number of service instances. [7]

In SOA-based applications, the number of service invocation requests and the service time has Poisson or exponential distribution, so the queues have an important property: memoryless property. By convention, we use *M* to respectively indicate the number of service invocation requests and the service time.

For the services, there are several instances to serve the clients. For example, if the service is implemented as a servlet or EJB in J2EE, there is an instance pool in application server to manage its multiple instances. These instances are parallel instances, which mean each instance serves only one client at any time. By convention, we use *S* to indicate the number of service instances.

For any application server, its capacity is limited. So the services hosting in application servers have the upper limitation of the number of clients they could serve. This upper limitation is the sum of the number of service instances and the number of waiting requests the queue could hold. By convention, we use *k* to indicate the service capacity.

Thus, the queue model for the services in SOA-based application is *M/M/S/k*. We can use its features to analyze services and detect their faults. We use the following symbols to indicate the basic conceptions in queuing theory:

1. λ: indicates arrival rate, which denotes the rate at which requests arrive at the service.
2. μ : indicates service completion rate, which denotes the rate at which responses depart from the service.
3. ρ: indicates the occupation rate or server utilization, which denotes the fraction of time the server is working.
4. P_j: indicates the probability that there are *j* requests in a queue when the queue reaches its steady state. In particular, P_0 denotes there is no request in the queue, which means the requests could be served immediately after they arrive in the queue and need not to wait.
5. *L*: indicates the expected length of queue, which equals to the mean number of requests in service and the mean number of requests in the queue.
6. L_q: indicates the expected length of waiting queue.
7. *W:* indicates the expected sojourn time.
8. *Wq:* indicates the expected waiting time.

For all services, the control center gathers their real-time information to compare with the description stored in the control center of ESB in SOA-based application in order to detect any service that fails to satisfy performance requirements. The specific algorithm is described as follows:

1. We set the unit time as 1 second, so in every second, we record the number of requests arriving in the service λ_i, the number of requests departing from each instance of the service $\mu_{i,j}$. For every period, such as every 100 seconds, we calculate the λ and μ by the following formulae:

$$\lambda = \frac{1}{n}\sum_{i=1}^{n}\lambda_i \qquad \mu = \frac{1}{Sn}\sum_{i=1}^{n}\sum_{j=1}^{S}\mu_{i,j} \tag{1}$$

2. Calculate P_0 by the following formula:

$$p_0 = \left(\sum_{n=0}^{S}\frac{(\lambda/\mu)^n}{n!} + \frac{1}{S!}\sum_{n=S+1}^{k}\frac{(\lambda/\mu)^n}{S^{n-s}}\right)^{-1} \tag{2}$$

3. Calculate every P_j by the following formula:

$$p_j = \begin{cases} \dfrac{(\lambda/\mu)^j}{j!}p_0 & j \le S \\ \dfrac{(\lambda/\mu)^j}{S!S^{j-s}}p_0 & S < j \le k \end{cases} \tag{3}$$

4. Calculate *L* by the following formula:

$$L = \sum_{j=0}^{k} jp_j \tag{4}$$

5. Calculate L_q by the following formula:

$$L_q = \sum_{j=0}^{k-S} jp_{S+j} \tag{5}$$

6. Calculate effective arrival rate by the following formula:

$$\lambda_e = \lambda(1 - P_k) \tag{6}$$

7. Calculate W by the following formula:

$$W = \frac{1}{\lambda_e} L \tag{7}$$

8. Calculate W_q by the following formula:

$$W_q = \frac{1}{\lambda_e} L_q \tag{8}$$

9. Compare W or W_q with W' or W_q' stored in service QoS Descriptor (it depends on which one is more important for users) to check whether the following inequation is true:

$$W - W' \le e \quad or \quad W_q - W_q' \le e \tag{9}$$

10. If the inequation (9) is true, than clear the counter C, which counts the number of successive failed periods. Then repeat to step 1.
11. If the inequation (9) is false, increase C by 1, and compare C with n, if C is not greater than n, repeat to step 1; otherwise, we consider that the service can not satisfy our performance requirement. As a result, we successfully detect a fault by our criteria.

Actually, for each service, the probability N_s that all of its S instances are busy could be calculated by the following formula:

$$N_s = \sum_{n=1}^{S} np_n + S\sum_{n=S+1}^{k} p_n = \frac{\sum_{n=0}^{S-1}\frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^S}{(S-1)!}\sum_{n=1}^{k-S}(\lambda/S\mu)^n}{\sum_{n=0}^{S}\frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^S}{S!}\sum_{n=1}^{k-S}(\lambda/S\mu)^n} \tag{10}$$

When all instances are busy, the incoming requests would be lost. So for some critical system in which the loss of requests is forbidden, N_s is also an assistant parameter to determine whether the service works well.

4. Reference service model

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

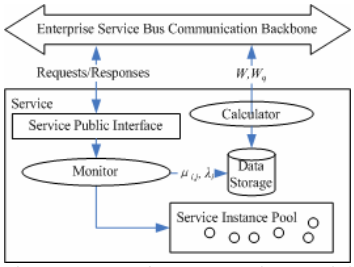

Figure 1. A reference service model

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

The clients of a service generate service invocation requests according to the service public interface, which contains all methods allowed to be invoked by clients. These requests are sent to ESB in form of some kind of messages, such as XML messages.

When the service invocation messages arrive at the service hosting environment, the monitor intercepts all of them, records their arrival times into its cache, and updates λ_i, the number of new arriving requests during the unit time right before time t, which is stored in the data storage. Subsequently, the monitor delegates the requests to the 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. When a method invocation is accomplished, there is a response generated and sent back to client. Meanwhile, the service instances pool reclaims the resource occupied by the invoked service instance.

The monitor intercepts responses to read the departure time before they are sent back to clients. Subsequently, the monitor updates $\mu_{i,j}$, the number of departing requests during the unit time right before time t, which is also stored in the data storage. Lastly, the responses are delivered to ESB.

The calculator accesses the data storage to retrieve λ_i and $\mu_{i,j}$, and uses them to calculate W and W_q by the algorithm described in section 3 every unit time. Finally, the calculator sends W and W_q onto ESB in order to make the control center of ESB 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.

Data storage stores the data that the calculator accesses frequently, so it should be designed as an in-memory object, such as an in-memory table or an in-memory map.

It is obvious that this reference service model achieves our goals 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.

5. Reference architecture of fault-tolerance control center of ESB

In [3], 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.

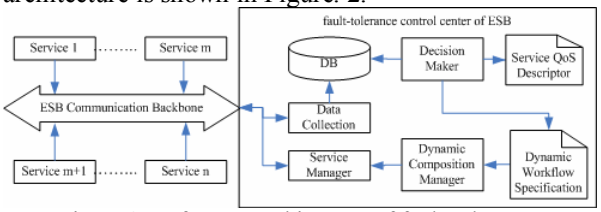

Figure 2. Reference architecture of fault-tolerance control center of ESB

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 W and W_q onto ESB.

The Data Collection Service, Dynamic Composition Manager, and Dynamic Workflow Specification are put forward by the authors of [3]. Data Collection Service will keep on monitoring the behavior of the participating services and collect data at runtime [3]. In our architecture, the data collected by Data Collection Service is all W and W_q 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 lightweight data storage, such as LDAP storage.

For each service, the Decision Maker accesses the Database to compare its W or W_q with its W' or W_q' stored in service QoS Descriptor, and determine whether the service is failed to satisfy the performance requirements by algorithm described in section 3. If the answer is yes, then a warning message will be sent to Dynamic Workflow Specification, which will determine whether the workflow specification should be modified or not [3]. If the workflow specification is modified, the Dynamic Composition Manager will re-composite the workflow at runtime [3].

6. Case study

Suppose we have an SOA-based application, there is a particular service in it. For simplification, we suppose this service is of M/M/6/6 model. We record the number of requests arriving in the system λ_i and the number of requests departing from the system $\mu_{i,j}$ every second for 100 seconds. Both the values of λ_i and $\mu_{i,j}$ range from 0 to 6, and the specific statistic data is shown in table 1:

Table 1: The numbers of various values of λ_i and $\mu_{i,j}$

values	0	1	2	3	4	5	6
Numbers of λ_i	17	25	25	16	11	5	1
Numbers of $\mu_{i,j}$	180	164	137	38	8	2	0

We calculate λ:

$$\lambda = \frac{1}{n}\sum_{i=1}^{n}\lambda_i = \frac{(17\times0+25\times1+25\times2+16\times3+11\times4+5\times5+1\times6)}{100} = 1.98$$

Then, calculate μ:

$$\mu = \frac{1}{Sn}\sum_{i=1}^{n}\sum_{j=1}^{S}\mu_{i,j} = 0.99$$

The occupation rate ρ is:

$$\rho = \frac{\lambda}{S\mu} = \frac{1.98}{6\times0.99} = \frac{1}{3}$$

The P_0 is:

$$p_0 = \left(\sum_{n=0}^{S}\frac{(\lambda/\mu)^n}{n!} + \frac{1}{S!}\sum_{n=S+1}^{k}\frac{(\lambda/\mu)^n}{S^{n-s}}\right)^{-1} = 0.136$$

The P_j is:

$$p_k = p_j = p_6 = \frac{(S\rho)^j}{j!}p_0 = 0.012$$

The λ_e is:

$$\lambda_e = \lambda(1-P_k) = 3.95$$

The L is:

$$L = \sum_{j=0}^{k} jp_j = 0.252$$

Since S and k are all 6, so L_q is 0. The W is:

$$W = \frac{1}{\lambda_e}L = 0.064$$

Now, we can compare W with W' stored in service QoS Descriptor, and determine whether the service is failed to satisfy the performance requirements.

7. Conclusion

In this paper, we put forward a queuing-theory-based algorithm to fault detection, which can be used to detect the services whose performance becomes unsatisfactory at runtime according to the Qos descriptor. Based on this algorithm, we give the reference models of the extended service and the

architecture of fault-tolerance control center of Enterprise Services Bus for SOA-based applications.

We also descript how the fault detection mechanism we put forward works with a study case. Actually, to demonstrate the correctness and practicability of this mechanism, we should develop a simulation application which runs on a SOA platform whose ESB have the fault-tolerance control center that we put forward. It is also the next research step that we will take 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] 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.

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

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

[6] Athanasics Papoulis, S.Unnikrishna Pillai, Probability, Random Variables, and Stochastic Processes, McGraw-Hill Companies, Inc., 2002

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