Performance Aware Service Pool in Dependable Service Oriented Architecture

Gang Huang[1,*] (黄 罡), Li Zhou[1] (周 立), Xuan-Zhe Liu[1] (刘儇哲), Hong Mei[1] (梅 宏), and Shing-Chi Cheung[2] (张成志)

[1]*School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, P.R. China*

[2]*Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong 999077, P.R. China*

E-mail: {huanggang, zhouli04, liuxzh}@sei.pku.edu.cn; meih@pku.edu.cn; sccheung@cs.ust.hk

Revised May 11, 2006.

Abstract As a popular approach to dependable service oriented architecture (SOA), a service pool collects a set of services that provide the same functionality by different service providers for achieving desired reliability. However, if the tradeoff between reliability and other important qualities, e.g., performance, has to be considered, the construction and management of a service pool become much more complex. In this paper, an automated approach to this problem is presented. Based on the investigation of service pools in the typical triangle SOA model, two challenges critical to the effectiveness and efficiency of service pools are identified, including which services should be held by a pool and what order these services are invoked in. A set of algorithms are designed to address the two challenges and then a service pool can be automatically constructed and managed for given reliability and performance requirements in polynomial time. The approach is demonstrated on a J2EE based service platform and the comparison results between different pooling algorithms are evaluated.

Keywords service oriented architecture, service pool, dependability, performance

1 Introduction

In the past few years, service oriented architecture (SOA) has been emerging as a promising paradigm for the development, deployment, integration and management of such distributed systems that reside in the sea of rapid and continuous changes of user requirements and runtime environments[1,2]. One of the most important issues in SOA is dependability. In terms of the concepts and taxonomy built up by Avizienis *et al.*[3], the dependability can be defined as the ability to avoid service failures that are more frequent or more severe than the extent acceptable by any stakeholders of the service. Particularly, the dependability becomes much more difficult to guarantee when using SOA to integrate distributed systems developed and/or operated by different parties.

A widely adopted solution to dependable SOA is to pool multiple services that provide the same functionality by different service providers[4]. For a given request, if a service in the pool fails, another service will be selected to process the request again. Obviously, the failure ratio of a service pool decreases rapidly according to the product of the failure ratio of each service in the pool. Conceptually, service pools are analogous to clusters. Both of them are to satisfy a given requirement for dependability. However, SOA is also subject to other important quality requirements, in particular, the performance. The construction and management of a service pool have to consider multiple quality requirements simultaneously. This makes the problem more difficult than that of the clusters. It is aggravated by the complexities brought about by the Internet. Firstly, Internet catalyses market competition and the personalization of services. This triggers the ever increasing number of services with the same or similar functions, offering a wide spectrum of prices and service qualities. It is impractical and even impossible to put all available services into one pool due to technical and nontechnical factors. Secondly, the invocation order of the services in a pool could impact some qualities, such as response time. As a result, though a subset of services determines the dependability of the pool, the invocation order of the services has to be carefully determined so that some other qualities can be guaranteed in a best effort way, e.g., guaranteeing the desired dependability while keeping the average response time minimal. Thirdly, the extremely open and dynamic nature of Internet makes the qualities of services change from time to time and then makes the construction and management of a service pool become difficult, error-prone and time-consuming.

In this paper, an automated approach to constructing and managing a service pool according to given reliability and performance requirements is presented. In Section 2, we investigate how service pools can guarantee dependability in the well-known triangle SOA model and identify the technical challenges of service pools. For automatically constructing a service pool that satisfies given dependability requirements while keeps the best-of-the-breed performance, we design a set of algorithms in Section 3. We also prove that the algorithms are optimal from the user's perspective and the execu-

Regular Paper

*Corresponding author

Supported by the National Natural Science Foundation of China under Grant Nos. 90412011, 90612011, 60233010, and 60403030, the National Basic Research 973 Program of China under Grant No. 2002CB312003, and the IBM University Joint Study Program.

tion times of the algorithms are polynomial. In Section 4, we demonstrate the approach on PKUAS, a J2EE application server integrating service oriented mechanisms, and evaluate the experimental results between different pooling algorithms. Finally, we introduce some related work in Section 5 and conclude the whole paper in Section 6.

2 Service Pool in Dependable SOA

Since dependability is tightly coupled with customers' satisfaction and enterprises' credits and profits, poor dependability could result in uncompensable damages, such as crash of service instances, crash of hosting servers, hang or deadlock of services, exceptions of messages (e.g., the delay is so long, a request message results in two or more replicated response messages), and so on. However, we find that failures mostly emerge from the elementary roles in the well-known triangle SOA model[1,2], including service consumers, service providers, service brokers and exchanged messages, as shown in Fig.1. Though different roles require different dependability mechanisms, the service pool is a common mechanism for the first three roles ①.

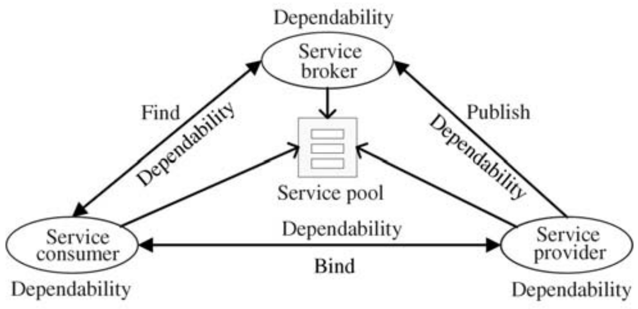

Fig.1. Dependability model of SOA.

2.1 Service Pool for Dependable Service Consumers

A service consumer usually requests a proper service and waits for responses. If a request fails, the consumer can repeat the request until it succeeds or the number of retries exceeds some limit. The consumer may also contact the service broker to locate and request an alternative service. In this case, the consumer itself implements core functions of a service pool, i.e., detecting the failures and selecting a new service.

These dependability mechanisms can be hard-coded in the implementation of service consumers or encapsulated in the client-side middleware. For example, J2EE application servers, such as Weblogic[5], JBoss[6], JonAS[7] and PKUAS[8], usually employ the client-side retrying and re-finding mechanisms for improving the dependability while moving the corresponding performance penalty from servers to clients.

Such dependability mechanisms work well in J2EE due to the relatively static nature of J2EE, such as the number and location of servers are deterministic, the failure of a server is usually temporary, etc. On the contrary, an SOA system is much more dynamic. As a result, the retrying may not succeed for ever because the service provider may disappear permanently, and the re-finding cost may be very expensive because the number and location of services may change rapidly and continuously. Obviously, a common and reusable service pool that deals with the construction and scheduling of service candidates can free service consumers from the dependability problems.

2.2 Service Pool for Dependable Service Brokers

A service broker usually stores a WSDL definition when a service provider publishes its service into the broker and returns a WSDL definition when a service consumer queries a desired service. For guaranteeing the dependability, the service broker may become much more active. It may periodically validate the availability of services specified by stored WSDL definitions. If a service is unavailable, its WSDL definition could be removed from the broker. Even better, the service broker may support ranking of services[9], that is, it can collect the failed and successful history of services and rank these services so that service consumers can always get the most dependable services. In these two cases, the service broker acts as a service pool using two different scheduling algorithms.

Such dependability mechanisms for the naming and directory servers are also common in the classical distributed computing areas. For example, the dependability of HTTP based systems is usually achieved by the way that the DNS server maintains a list of replicated HTTP servers and returns one of them to an HTTP client in terms of some scheduling algorithms.

The effectiveness of such dependability mechanisms depends on the implementation details of the application. For example, if a service consumer invokes the service broker per request to a desired service, it will get the most dependable service. If the consumer invokes the broker one time and requests the service more times, the dependability mechanisms provided by service brokers cannot impact the consumer until it invokes the broker again. Since SOA systems usually belong to the first case, the service pool in service brokers is effective in SOA. According to the experience gained in traditional distributed systems, the scheduling algorithm is the key to the effectiveness and efficiency.

①The dependability supported by service pools has many terms, such as reliability, availability and accessibility in different literature. To our point of view, these different terms just come from different perspectives. In this paper, we use 'reliability' because we focus on how to implement the service pool.

2.3 Service Pool for Dependable Service Providers

There are two types of service providers: one provides a service by assembling a set of existing services while another provides a service by encapsulating existing business logic in legacy systems.

In the first type of services, called composite services, the service pool is a common dependability mechanism. For example, in most service composition approaches, a service type can be implemented by a set of service partners, which will be selected dynamically by some algorithms. Furthermore, a service provider can assemble multiple services into a service pool for providing a more reliable version of the service. Though the service pool is commonly used in service composition, its construction and management are done by hand or automated by some inefficient algorithms. More discussion on their limitation can be found in the section of related work.

The stakeholders of SOA systems usually overlook the dependability mechanisms of the second type of services, called primitive services. Usually, a primitive service is implemented by a set of backend components in legacy systems, in which classical middleware, such as CORBA and J2EE, provides many sophisticated dependability mechanisms. These mechanisms can improve the dependability as well as impact other qualities of primitive services and then make the scheduling of these services in a pool more complex. For example, we employ the recovery-oriented computing mechanisms for repairing the failure of a service, which can improve the dependability of a service by 80% but increase the response time by 10 to 15 seconds[10]. This work also shows that the dependability mechanisms of backend systems can be efficiently utilized if and only if the stakeholders of SOA systems take them into consideration.

3 Pooling Algorithms

From the above study on service pools in the triangle SOA model from the perspective of dependability, it can be concluded that the core of the service pool is the pooling algorithm, which can select a set of services and define the invocation order according to desired qualities. Different qualities have different calculations and we only focus on the reliability and response time, i.e., keeping the response time as little as possible while satisfying a given reliability.

3.1 Basic Definitions

The reliability and response time are defined as follows. (Note that, there are some other definitions of these two qualities, which do not impact our algorithms.)

Reliability is a successful execution rate of a service, which is the probability representing the degree to which a request is correctly served[11]. It can be calculated as follows:

$$V_{reliability} = N(s)/M(s),$$

where $N(s)$ is the number of times that the service has been successfully completed within the maximum expected time frame and $M(s)$ is the total number of invocations.

Response time is the amount of time between sending the request and receiving a response[11] or the guaranteed average time required to complete a service request, which can be represented in the following formula:

$$Responsetime = T_{executiontime} + T_{delay} + T_{waiting},$$

where $T_{executiontime}$ is the time to process the service task, T_{delay} is the time for the transmission, and $T_{waiting}$ is the time of waiting for the result. In our model, the binding time and network latency are not considered as either they can be ignored, compared with the execution time, or they cannot be controlled at all in the unstable or heavy traffic network. In the rest of our paper, we use $\triangleq$ to denote the definition formula.

Suppose that there are n service instances in the pool. Let $S \triangleq \{S_1, S_2, \ldots, S_n\}$ be the pool of service instances, $R_1, R_2, \ldots, R_n$ be the reliability of them with execution time $T_1, T_2, \ldots, T_n$.

Let $I \triangleq \{1, 2 \ldots, n\}$ be the index set. If A is a subset of I, we use $S(A) \triangleq \{S_i | (i \in A)\}$ to denote the selected group of service instances. The MRT (maximum response time) of this group of service instances is denoted as $MRT(A) \triangleq \sum_{i \in A} T_i$. It is easy to prove that the reliability of $S(A)$ denoted by $R(A)$ is $1 - \prod_{i \in A}(1 - R_i)$ ②. The minimum reliability of the service required is denoted by R_0. There are many measurements of the response time, two of which are considered in this paper for simplification. The first measurement is the *minimal maximum response time* of S, denoted as MINMRT, where $MINMRT(S) \triangleq \min_{(A \subseteq S) \wedge (R(A) > R_0)} (MRT(A))$ and $R(A) > R_0$ are the reliability constraint of the services. The second measurement is the *average response time* of S, denoted by ART, which means the mathematical expectation of the response time. To calculate ART, we have to rearrange the selected set A, and denote the result as $\sigma(A) \triangleq (i_1, i_2, \ldots, i_m)$, where $A = \{i_1, i_2, \ldots, i_m\}$. The corresponding service invocation order can be decided by this. In this order, if the instance i_k is missed, the request will be automatically bound to the instance i_{k+1}. Therefore, it is easy to obtain the formula of calculating the expected response time as

$$ART(\sigma(A)) \triangleq E(T_{Execution})$$
$$= R_1 T_1 + (1 - R_1) R_2 (T_1 + T_2)$$

② We assume unreliability is contributed only by services. Note that other components in the service pool management, such as service switching could introduce unreliability, which are omitted since we assume the service pool itself is reliable.

$$\begin{aligned}
&+(1-R_1)(1-R_2)R_3(T_1+T_2+T_3)+\cdots\\
&+(1-R_1)(1-R_2)\cdots(1-R_{m-1})\\
&\cdot R_m(T_1+T_2+\cdots+T_m)+(1-R_1)(1-R_2)\cdots\\
&\cdot(1-R_{m-1})(1-R_m)(T_1+T_2+\cdots+T_m)\\
=&T_1(R_1+(1-R_1)R_2+\cdots+(1-R_1)(1-R_2)\cdots\\
&\cdot(1-R_{m-1})((1-R_m)+R_m))\\
&+T_2((1-R_1)R_2+\cdots+(1-R_1)(1-R_2)\cdots\\
&\cdot(1-R_{m-1})((1-R_m)+R_m))+\cdots\\
&+T_m(1-R_1)(1-R_2)\cdots(1-R_{m-1})(R_m+(1-R_m))\\
=&T_1+T_2(1-R_1)+\cdots+T_m(1-R_1)(1-R_2)\cdots(1-R_{m-1})\\
=&T_1+T_2(1-R_1)+\cdots+T_m\Big(\prod_{l=1}^{m-1}(1-R_{i_l})\Big)\\
=&\sum_{j=1}^{m}\Big(\prod_{l=1}^{j-1}(1-R_{i_l})\Big)T_{i_j}\ \Big(\text{where}\prod_{l=1}^{0}R_{i_l}\triangleq 1\Big),
\end{aligned}$$

where the meaning of A and $\sigma(A)$ is the same as above.

For convenience, we represent the values of reliability measurements as the logarithm of the failure ratio. Let $ER_i \triangleq 1-R_i, ER(A)\triangleq 1-R(A)$ be the failure ratio, $LER_i\triangleq -\lg(ER_i), LER(A)\triangleq -\lg(ER(A))$ be the logarithm of failure ratio, where $\lg(x)$ is the common logarithm function and $i\in I\cup\{0\}=\{0,1,\dots,n\}$. Notice that LER is always positive since ER<1. The reason we choose this measurement is that the requirement of the accuracy of the percentage notation of R_0 will be increased when R_0 is close to 1. For example, the reliability of 90% and 90.1% can be considered almost the same in most cases, while 99.9% and 99.99% are very different. If a server receives 10000 requests per hour, 99.9% means that one error will occur per six minutes in average while 99.99% means that one error will occur per hour in average. So, it would be better to use the scientific notation of the failure ratio to express our requirements, which is proper to be measured by the logarithm instead of the percentage of reliability. Here we make an assumption that every LER has the accuracy of 2 decimal digits. We think this approximation is reasonable since very few users require such a high accuracy that they care the reliability difference between 90% and 90.5%, or between 99.9% and 99.905%. Of course, we can ask for a higher accuracy or lower accuracy. The algorithm would still work when the requirement of accuracy changes, while the accuracy does not impact our algorithm.

3.2 Optimal Analysis

Our optimal analysis on this approximate model includes two kinds of selection problems.

1) MRT-prior selection. It is easy to prove that $MINMRT(S)$ is not related to the order of service instances of a selected set $S(A)$. So an MRT-prior selection can be described as follow. Firstly, we select a subset A of index set I satisfying $MRT(A) = MINMRT(S)$, i.e., $S(A)$ is the set of selected service instances ensuring that the reliability of the service will not be less than R_0, and the MRT of this set is minimum. This subset is denoted by A_{Min}. Secondly, we select an arrangement $\sigma(A_{Min})$ of A_{Min} (denoted by $\sigma(A_{Min})\triangleq(i_1,i_2,\dots,i_m)$, where $A_{Min}=\{i_1,i_2,\dots,i_m\}$), which satisfies that for any arrangement of A_{Min}, say, $\sigma'(A)$, such that $ART(\sigma(A))\leqslant ART(\sigma'(A))$. $\sigma(A_{Min})$ is called an MRT-prior selection.

2) ART-prior selection. Only the first k-th three service instances need to be considered because the contributions of other instances to ART are so small (less than 1/100) that they can be ignored, as we will show later. Therefore, we select the first three service instances in the pool to get an approximately optimized ART and it will be close to the best. For the rest of service instances, we prefer to find an MRT-prior solution. So, an ART-prior selection can be described informally as follow. Firstly, we select three service instances as the first three instances so that the ART of this selection is close to the best. Then we make an MRT-prior selection for other service instances so that the service can still have a good MRT and the reliability requirement is satisfied.

3.3 MRT-Prior Selection Algorithm

Without loss of generality, we assume that S is sorted by descending sort of T_i/R_i $(i=1,2,\dots,n)$. The reason for this assumption will be discussed later. To solve the MRT-prior selection problem, the first step is to select A such that $MRT(A) = MINMRT(S)$. Suppose that $S(A)$ is the selected set. By the above definitions of ER and LER we can easily get that $LER(A)=\sum_{i\in A}LER_i$.

So we have

$$\begin{aligned}R(A)>R_0 &\Leftrightarrow \lg(1-R(A))>-\lg(1-R_0)\\ &\Leftrightarrow LER(A)>LER_0,\end{aligned}$$

then

$$\begin{aligned}&\operatorname*{Min}_{(A\subseteq I)\wedge(R(A)>R_0)}(MRT(A))\\ =&\operatorname*{Min}_{(A\subseteq I)\wedge\left(\sum_{i\in A}LER_i>LER_0\right)}\Big(\sum_{i\in A}T_i\Big).\end{aligned}$$

It can be proved that if all the values were considered as real numbers, this problem would have the same complexity with the 0–1 knapsack problem which is NP-hard. But we have already been on the assumption that the LER values only have 2 decimal digits. This means that the value set of LER is finite instead of infinite. So we can find an algorithm with polynomial time respect to $|I|$ and another parameter about LER. In fact, our algorithm is linear to both $|I|$ and the parameter $P\triangleq\sum_{i\in I}LER_i$ in our approximate model. It is very similar to the integer-value 0–1 knapsack algorithm[11], although we must make some changes.

Let $I_k\triangleq\{1,2,3,\dots,k\}\subseteq I$. We define a function $F(k,e)\triangleq\operatorname*{Max}_{(\overline{A}\subseteq I_k)\wedge(\sum_{i\in\overline{A}}LER_i<e)}(\sum_{i\in\overline{A}}T_i)$, and

$F(k,e) \triangleq -\infty$ if $e < 0$. Note that

$$\begin{aligned}
MINMRT(S) &= \underset{(A\subseteq I)\wedge(R(A)>R_0)}{\mathrm{Min}}(MRT(A))\\
&= \underset{(A\subseteq I)\wedge(\sum_{i\in A} LER_i > LER_0)}{\mathrm{Min}}\Big(\sum_{i\in A} T_i\Big)\\
&= \underset{(\overline{A}\subseteq I)\wedge(\sum_{i\in \overline{A}} LER_i < P - LER_0)}{\mathrm{Min}}\Big(\sum_{i\in I} T_i - \sum_{i\in \overline{A}} T_i\Big)\\
&= \sum_{i\in I} T_i - \underset{(\overline{A}\subseteq I)\wedge(\sum_{i\in \overline{A}} LER_i < P - LER_0)}{\mathrm{Max}}\Big(\sum_{i\in \overline{A}} T_i\Big)\\
&= \sum_{i\in I} T_i - F(n, P - LER_0), \qquad (1)
\end{aligned}$$

where $\overline{A} \triangleq I - A$ is the complement of A. Therefore we can calculate $MINMRT(S)$ by computing $F(k,e)$ recursively from the following equations:

$$\begin{cases}
F(k,e) := \mathrm{Max}\{F(k-1,e), F(k-1,e-LER_k)+T_k\};\\
F(0,e) := 0, \quad 0 \leqslant e \leqslant P;\\
F(k,0) := 0, \quad 0 \leqslant k \leqslant n;\\
F(k,e) = -\infty, \quad (e<0).
\end{cases} \qquad (2)$$

Now we show the explanation of (2). Let $A_0 \subseteq I$ satisfies $MRT(A_0) = F(k,e)$. There are only two cases. If k belongs to A, since $A_0 - \{k\} \subseteq I_{k-1}$, $F(k,e) = MRT(A_0) = \sum_{i\in A_0} T_i = \sum_{i\in A_0-\{k\}} T_i + T_k \leqslant F(k-1, e-T_k) + T_k \leqslant F(k,e)$, and then $F(k,e) = F(k-1, e-LER_k) + T_k$. If k does not belong to A, it is obvious that $F(k,e) = F(k-1,e)$. So the first equation of (2) holds, and other equations of (2) are quite obvious.

We use $A(k,e)$ to denote a subset A which satisfies $MRT(\bar{A}) = F(k,e)$. It will not make any confusion in this paper. From the above discussion, the correctness of the following algorithm is proved.

Algorithm 1. Calculate All $F(k,e)$ and $A(k,e)$

```
1. $F(0,e) := 0$, $0 \leqslant e \leqslant P$, $F(k,0) := 0$, $0 \leqslant k \leqslant n$,
   $F(k,e) := -\infty$ $(e<0)$;
2. $A(0,e) := \emptyset$, $0 \leqslant e \leqslant P$, $A(k,0) := \emptyset$, $0 \leqslant k \leqslant n$,
   $A(k,e) := \emptyset$ $(e<0)$;
3. for $k := 0$ to $n$
   for $e := 0$ to $P$ step 0.01
     $F(k,e) := \mathrm{Max}\{F(k-1,e), F(k-1,e-LER_k)+T_k\}$;
     if $(F(k,e) = F(k-1,e-LER_k)+T_k)$ then
       $A(k,e) := A(k-1,e-LER_k) \cup \{k\}$;
     else
       $A(k,e) := A(k-1,e)$;
4. return $F(k,e), A(k,e)$ for every $(k,e)$.
```

Now we can get $MINMRT(S)$ by returning $\sum_{i\in I} T_i - F(n, P-LER_0)$. And we can get A_{Min} because $A_{\mathrm{Min}} = I - A(n, P-LER_0)$. It is easy to prove the complexity of this algorithm is $O(100Pn) = O(Pn)$.

The MRT-prior selection problem has not been finished yet because we have to decide $\sigma(A_{\mathrm{Min}})$. We need the following lemma to solve this problem.

Lemma 1. *Given an index subset A. If $\sigma(A)$ is an arrangement of A satisfying that for any arrangement $\sigma'(A)$, $ART(\sigma(A)) \leqslant ART(\sigma'(A))$, then, let $\sigma(A) = (i_1, i_2, \ldots, i_m)$, we have $T_{i_j}/R_{i_j} \leqslant T_{i_{j+1}}/R_{i_{j+1}}$, $\forall j < m$.*

Proof. Suppose that there is a J, which makes $k_J \triangleq T_{i_J}/R_{i_J} > k_{J+1} \triangleq T_{i_{J+1}}/R_{i_{J+1}}$. Recall the definition of $ART(\sigma(A))$, i.e. $ART(\sigma(A)) \triangleq \sum_{j=1}^{m}\Big(\prod_{l=1}^{j-1} ER_{i_l}\Big)T_{i_j}\Big(\prod_{l=1}^{0} R_{i_l} \triangleq 1\Big)$. Now, let $\sigma' = (i_1, \ldots, i_{J-1}, i_J, \ldots, i_m)$, $a \triangleq ER_{i_j}, b \triangleq ER_{i_{j+1}}$, and let $\alpha \triangleq \prod_{l=1}^{j-1} ER_{i_l}$. It is obvious that $\alpha > 0$. Then,

$$\begin{aligned}
&ART(\sigma) - ART(\sigma')\\
&= \alpha \cdot ((T_{i_J} + T_{i_{J+1}} ER_{i_J}) - (T_{i_{J+1}} + T_{i_J} ER_{i_{J+1}}))\\
&= \alpha \cdot ((k_J(1-a) + k_{J+1}(1-b)a)\\
&\quad - (k_{J+1}(1-b) + k_J(1-a)b))\\
&= \alpha \cdot (k_J - k_{J+1})(1-a)(1-b) > 0. \qquad (3)
\end{aligned}$$

Since $ART(\sigma(A)) > ART(\sigma'(A))$ is a contradiction, this lemma is proved. □

Because the arrangement set of A is finite, such $\sigma(A)$ in the above lemma exists and $ART(\sigma(A))$ is the minimum solution. So it is easy to get the following lemma by recalling the assumption in the beginning of this subsection, which ensures the descending sort of T_i/R_i $(i = 1, 2, \ldots, n)$.

Lemma 2. *If $\sigma(A_{Min})$ is the descending sort of the index subset A_{Min}, then $\sigma(A_{Min})$ is one of the MRT-prior selections of S.*

With this, the following algorithm makes sense.

Algorithm 2. MRT-Prior Selection

```
1. Re-label the index of $S$ in order that $S$ is sorted by
   the descending sort of $T_i/R_i$ $(i = 1, 2, \ldots, n)$;
2. Calculate all $F(k,e)$ and $A(k,e)$;
3. Return $\sum_{i\in I} T_i - F(n, P-LER_0)$ as $MINMRT(S)$;
4. Return the descending sort of $I - A(n, P-LER_0)$ as
   a MRT-prior selection of $S$.
```

It is not hard to see that the complexities of these four steps are $O(n\log n)$, $O(nP)$, $O(1)$, $O(n)$ respectively. Therefore, the overall complexity of this algorithm is $O(nP)$.

3.4 ART-Prior Selection Algorithm

It is reasonable to assume that the reliability of every service instance in the pool is over 80%. By the definition of ART, the coefficient of T_{i_j} is $\prod_{l=1}^{j-1} ER_{i_l}$. Therefore, the coefficient of T_{i_j} is not greater than 1/125 when $j > 3$, and reduced exponentially. In that sense, it is reasonable to ignore the contributions of the service instances after the first three instances. So after the first three instances are selected, we prefer to use the MRT-prior selection for the other instances. Then we can define a good selection as follow:

Definition. *A minimum selection (MS for short) $\sigma Min(A)$ is a selection such that the reliability constraint is satisfied and $ART(\sigma Min(A)) \leqslant ART(\sigma'(A'))$ for every selection $\sigma'(A')$. Let $\sigma(A)$ be a selection. We call $\sigma(A)$ a good selection if the following statement is true: for every $\sigma Min(A)$ there exist selections $\sigma'(A')$,*

$\sigma''(A'')$ such that $ART(\sigma(A)) \leqslant ART(\sigma'(A'))$, while $\sigma'(A')$ and $\sigma Min(A)$, $\sigma''(A'')$ and $\sigma(A)$ both have the same first three terms, and all of them satisfy the reliability constraint.

Now we prove the following lemma:

Lemma 3. *Suppose every MS has at least three terms. Then any descending sort of $\sigma(A)$, where A contains $\{n, n-1, n-2\}$, is good with the assumption in the beginning of Subsection 3.4.*

Proof. Let $\sigma Min(A)$ be any *MS*. Select a $\sigma'(I)$, an arrangement of the index set $\sigma'(I)$ which has the first $|A|$ terms is the same as $\sigma Min(A)$. From Lemma 1, $ART(\sigma'(I)) \geqslant ART((n, n-1, n-2, \ldots, 1))$. So the lemma is proved. □

If some service instances can satisfy the reliability constraint itself, then we use M to denote the index of the one which has the minimum response time. Similarly, we use (M_1, M_2) to denote the pair that has a minimum ART and satisfies the reliability constraint themselves. It is easy to prove that if the ART of a selection is smaller than a good selection, this selection is good too. And it is also clear that if an *MS* has less than 3 terms, it must be (M_1, M_2) or (M). Now we suppose that $\sigma(A)$ is any selection the first three terms of which are $(n, n-1, n-2)$. Then, from Lemma 3, we can get that $\mathrm{Min}\{ART(\sigma(A)), ART((M_1, M_2)), ART((M))\}$ is the ART of a good selection. We can choose the corresponding arrangement as the good selection.

The following algorithm is a solution to this problem and the correctness of it is ensured by the above discussion.

```
Algorithm 3. ART-Prior Selection
  1 and 2 are the first two steps of the algorithm MRT-
    prior selection.
  3. if $LER_0 > LER_n + LER_{n-1} + ER_{n-2}$ then
       Tag: =true;
       $TM := \sum_{i\in I} T_i - F(n-3, P - (LER_0 - LER_n - LER_{n-1} - LER_{n-2}) + \sum_{i=0}^{2} T_{n-i};$
       $A := I - A(n-3, P - (LER_0 - LER_n - LER_{n-1} - LER_{n-2})) + \{n, n-1, n-2\};$
$\sigma'(A)$ := the descending sort of $A$.
     else
       Tag:= false; $A := \infty$;
  4. Search $M$ and $(M_1, M_2)$.
  5. $T\_\mathrm{min} := \mathrm{Min}\{ART(\sigma'(A)),$
     $ART((M_1, M_2)),\ ART((M))\}$;
     $\sigma(A)$:= the corresponding arrangement of $T\_\mathrm{min}$.
  6. Return $\sigma(A)$, $T\_\mathrm{min}$ as the ART-prior selection and
       its ART.
  7. if $T\_\mathrm{min} = ART(\sigma'(A))$ and Tag= true then
       Return TM as the MRT of this selection
       else Return $MRT(\sigma(A))$ as the MRT.
```

We can sort all T_i before the search of M and (M_1, M_2). Notice $ART((M_1, M_2)) = T_{M_1} + ER_{M_1} T_{M_2}$. Therefore, by using the binary search to the selections of M_2, the complexity of the fourth step is $O(n \log n)$. It is obvious that the first two steps are the bottleneck of the algorithm. Therefore the complexity of this algorithm is also $O(nP)$.

4 Implementation and Experiments

4.1 Prototype on PKUAS

Being a J2EE application server, PKUAS[8] has been widely used in Chinese academia and industry. For supporting some SOA systems, e.g., the national wide component repositories, PKUAS integrates a set of web service mechanisms, including JUDDI[12] as the service broker and Apache AXIS[13] as the SOAP engine. On the basis of PKUAS, we implement a service pooling prototype as a special middleware plug-in, as shown in Fig.2.

The core of the prototype is a Service Pooling Manager, which consists of three major components. The Pool Planner gets the functional requirement from the service type definition that uses WSDL and then queries the UDDI server with the given functions. It also gets the reliability and response time requirements from a WSLA or an extended WSDL of the given service type. After that, the Pool Planner will use the pooling algorithm to generate the selection policy according to the given reliability and response time requirements. On the basis of the selection policy, the Service Selector determines which services will be selected for or removed from the service pool. The Pool Generator finally generates the service pool as an available service queue for service consumers. It maintains the up-to-date states of the service pool, including the execution state and messages triggered by services. Once the pool is unavailable or suffers a significant performance penalty, it requests the Service Selector to do removal operation and re-plan the pool policy.

4.2 Experimental Evaluation

We conducted a series of experiments for two goals, i.e., investigating the improvement of reliability by using the service pool and illustrating the effectiveness of reducing the response time through our pooling algorithms. The experimental environment consists of a server and several client machines. All the machines are Pentium IV 2.8G PC with 512M DDR memory and directly connected with 100 Mbps Ethernet.

Firstly, we deploy 10 services having the same functions in the AXIS server and publish them in the JUDDI registry. Then we search for services, consume every service and provide feedbacks to record the response time and reliability. The three steps are repeated 500 times. Particularly, we randomly interrupt some requests, i.e., throwing exceptions, so that the successful ratio of a service falls into the range of 80% to 90%. Then the test data for each service instance with the concrete values of the response time and reliability is generated.

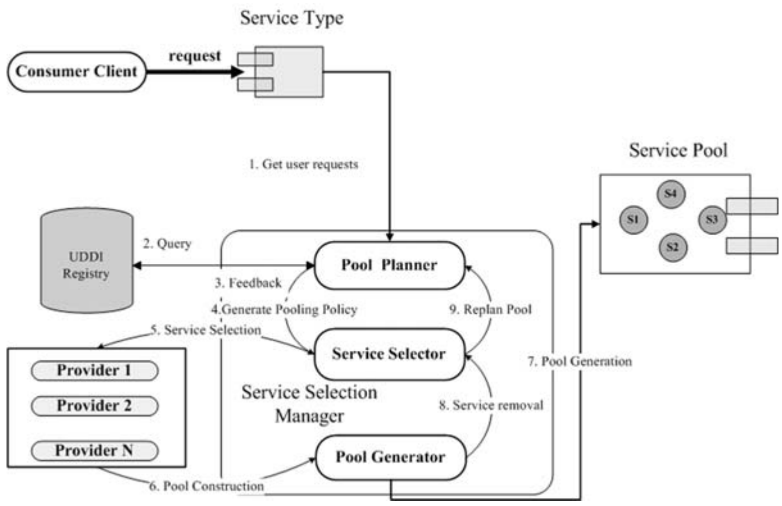

Fig.2. Architecture of service pooling prototype.

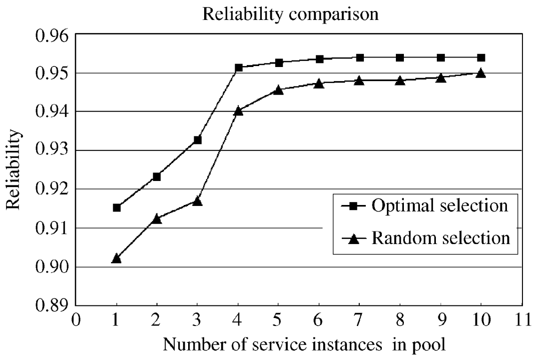

Fig.3. Comparison of reliability (ART Prior).

Next, we consume the service for 1000 times while increasing the size of service instances. Four different algorithms for constructing the service pool tested here are: 1) non-optimal selection in the random sequence; 2) optimal selection with the optimal algorithm; 3) reliability prior selection in the reliability descending order; 4) response time prior selection in the response time ascending order.

Two assumptions are held in the experiment: 1) we do not consider the binding time of requests since all the services are deployed in the same network environment; 2) the sort duration is neglected since we apply Merge-Sort, which costs less than 20 milliseconds and such time cost can be neglected compared with the execution time of a service.

4.2.1 *Simulation 1: ART-Prior Selection*

First we investigate the ART-prior algorithm and we constraint the reliability ranges from 85% to 95%. Fig.3 shows the trend for reliability with the size of service pool. No matter what selection policy is applied, we can see that the reliability increases when the service pool size increases. This gives us the confidence that the reliability will be higher by using more service instances in a certain range. On the other hand, it also shows the effectiveness of different algorithms. Besides the reliability prior selection (it always selects the service having higher reliability than others), the optimal selection can achieve a little higher reliability than the random selection and the response time prior selection. Another conclusion using the optimal selection algorithm we get is that the reliability keeps stable after the pool involves 4 instances (here the reliability is 95.4%). It means the first four instances are reliable enough in most cases, while the random selection needs more instances to reach the same reliability. Therefore, by the optimal policy, it is unnecessary to enlarge the capacity of service pool continuously and then the maintenance penalty can be well controlled.

Fig.4 shows that the response time increases more or less when the pool size extends. Obviously, the increasing response time penalty in the optimal selection is much smaller than the other three algorithms. When the pool generated by our optimal algorithm reaches the capacity of 4 instances, the response time will be very stable (about 9600 milliseconds). The reason is, usually by our algorithm, the first 3 instances can satisfy the reliability requirement, and the remainders can be neglected in almost all cases. On the contrary, the response time in the random selection grows very fast. The response time of the response time prior selection is only less than that of the random selection. It means that only considering the response time may not get the desired little response time. Although the reliability prior selection can achieve much higher reliability, it always leads to longer response time than the optimal selection (increasing 10%–30%), while the reliability difference is very small (95.4% versus 95.7%).

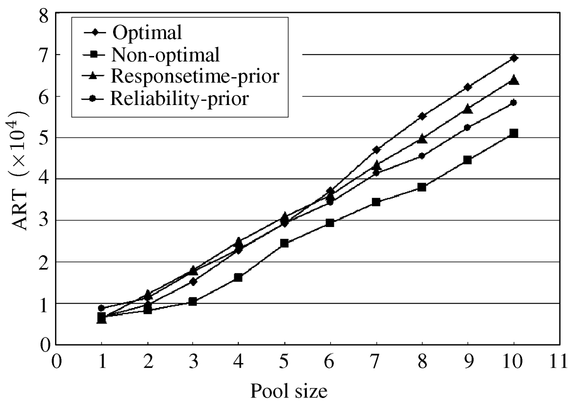

Fig.4. Comparison of response time (ART Prior).

4.2.2 *Simulation 2: MRT-Prior Selection*

To demonstrate the MRT-prior algorithm, we reach much higher reliability, i.e. 98%. As shown in Fig.5, it is obviously that the optimal selection always resides below other three algorithms. It reveals that optimal selection guarantees the lowest time overhead. The random selection has the worst MRT, since it may try all service instances. Another conclusion we get is that the slope of all four algorithms becomes higher when the pool size increases and the smaller the pool size, the better the effectiveness of our optimal algorithm.

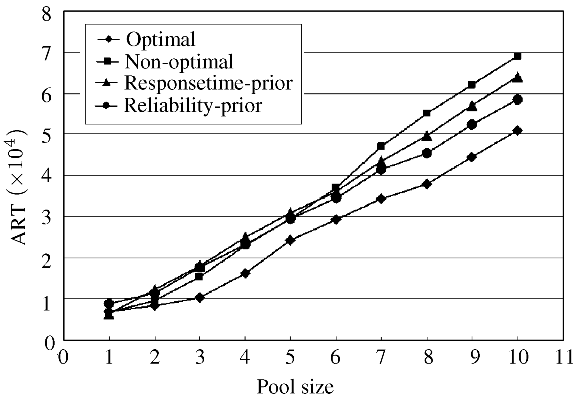

Fig.5. Comparison of response time (MRT Prior).

5 Related Work

Currently, the selection for dependable web services becomes an active research area. Due to the computational complexity of service selection over the Internet, some researchers focus on the optimal way. Bonatti *et al.*[14] proved the service selection problem as NP-hard and proposed three kinds of optimal algorithms. These algorithms are based on cost minimization and two different quality maximization criteria. Their work provides some computational foundation and heuristic solutions for service selection.

In AgFlow[11,15], Zeng *et al.* discuss a global planning way to generate a QoS model for service selection. A multidimensional QoS model involves price, accessibility, response time and reputation. They provide a mechanism for service providers to query their QoS computed by the QoS registry, and update their published services to become more competitive at runtime. For the optimal selection, linear planning is applied to the QoS matrix for maximum QoS values. For a given service flow and assuming each service has multiple implementations, they try to calculate an execution path that satisfies the desired multiple QoS requirements. The most important contribution is to design and compare two methods. One is to select the optimal implementation only considering the individual service. Another is to select the optimal implementation from the global perspective. It should be noted that their global selection is similar to the scenario specific validation in our approach. They optimize the global calculation with Integer Programming (IP), which reveals the fundamental differences with ours.

The algorithm presented in this paper uses a different mathematical model for the above work. In real-world SOA cases, we discovered that users not only neglect the little difference of the response time of different instances, but also do not care the difference of such as 89.9% and 90% reliability. The two-digit is enough to specify the reliability constraint in practice. It also makes our algorithm with precise MRT solution in the approximate model. The discussion in the beginning of Subsection 3.4 explains why we chose the first three instances, and the latter experiment supports this discussion.

The broker is a common technique to improve networking level qualities such as reliability and availability, etc. The broker acts as an intermediary third party to make web services selection and QoS negotiation on behalf of the client. Tsai *et al.*[9] propose a testing approach, CV&V (Collaborative Verification and Validation) on UDDI server to achieve dynamic reconfiguration. They suggest group testing technique that has the ability to evaluate the test scripts, automatically establish the oracle of the each test script, and identify faulty web services in a failed composite one. Erradi *et al.*[16] design a QoS-aware dynamic reconfigurable broker (called wsBus) to ease reliable integration and runtime management of web services by using a broker pattern enriched with various fault-tolerance mechanisms. However, the delegation of selection and negotiation raises performance penalty. Actually, our approach can also be regarded as a broker. We give details on how to reduce the performance penalty and enforce our approach in an objective manner for the service consumers.

Besides the service consumer, broker and provider, the relationships between them can contribute to the dependability, i.e., the dependable message exchange. Since existing popular middleware has been originated for problems of message exchange, e.g., CORBA for interoperation between distributed objects, there are many efforts on how to improve dependability via special mechanisms in message exchange. For example, FlexiNET[17] and MChaRM[18] support asynchronous message exchange, group communication, buffer-and-deliver transmission, etc. PKUAS[8] provides an open interoperability framework so that some special dependable communication protocol, like RTP (Real Time Protocol), can be installed on demand. There are some similar efforts on SOAP. Looker *et al.*[19] have concluded the limitations of current SOAP 1.1 standard on building an FT web service and then propose an FT-SOAP (Fault Tolerant SOAP) for building web services with higher resilience to failure. Obviously, dependability mechanisms for message exchange can deal with dependability threats related to communications with more or less performance penalty. More importantly, these mechanisms are usually specific to the protocol implementation so that it will impact the openness of SOA.

6 Conclusions and Future Work

The dependability and performance are the most important qualities in service oriented architecture (SOA). The service pool succeeds in improving dependability of SOA systems but suffers the significant performance penalty. In this paper, we presented an automated approach to constructing and managing service pools that can achieve the given reliability while keeping the response time (e.g., average response time or maximum response time) minimal. The algorithms in our approach are optimal and spend polynomial time, which are proved not only in theory but also by the experiments on J2EE.

There are many open issues to be addressed in the future. Firstly, service pools should support the trade-off between more QoS attributes. However, it should be noted that the measurements of different QoS attributes may be diverse from one to another. And the interactions between some qualities are hard to investigate in a quantified way. In that sense, the popularization of the service pool is correlated to the QoS modeling of SOA. Secondly, the service pool may not work well in service composition because QoS evaluation of service composition does not rely on individual services. So the role and challenges of the service pool, in service composition, should be investigated carefully. Finally, the service pooling prototype will be improved and released as a plug-in to popular service platform and then more cases can be studied.

References

[1] Kreger H. IBM Web Services Conceptual Architecture. 2001. http://www.ibm.com.

[2] Papazoglou M P, Georgakopoulos D. Service-oriented computing: Introduction. *Communications of ACM,* 2003, 46(10): 24–28.

[3] Avizienis A, Laprie J C, Randell B *et al.* Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable and Secure Computing,* 2004, 1(1): 11–33.

[4] Tsai W T, Song W, Paul R *et al.* Services-oriented dynamic reconfiguration framework for dependable distributed computing. In *28th Annual Int. Computer Software and Applications Conf.* (*COMPSAC*), Hongkong, China, 2004, pp.554–559.

[5] WebLogic Homepage. http://www.bea.com.

[6] JBoss Homepage. http://www.jboss.org.

[7] JonAS Homepage. http://www.objectweb.org.

[8] Mei H, Huang G. PKUAS: An architecture-based reflective component operating platform. In *10th IEEE International Workshop on Future Trends of Distributed Computing Systems* (*FTDCS*), Suzhou, China, 2004, pp.163–169.

[9] Tsai W T, Paul R, Cao Z *et al.* Verification of web services using an enhanced UDDI server. In *The Eighth Workshop on Object-Oriented Real-Time Dependable Systems* (*WORDS*), Guadalajara, Mexico, 2003, pp.131–138.

[10] Huang G, Liu X, Mei H. SOAR: Towards dependable service-oriented architecture via reflective middleware. *Int. J. Simulation and Process Modeling,* Jan. 2007 (to appear).

[11] Zeng L, Benatallah B *et al.* QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, May 2004, 30(5): 311–327.

[12] http://ws.apache.org/juddi/.

[13] http://ws.apache.org/axis.

[14] Bonatti P A, Festa P. On optimal service selection. In *Proc. Int. Conf. World Wide Web*, Japan, 2005, pp.530–538.

[15] Liu Y, Ngu A H, Zeng L J. QoS computation and policing in dynamic web service selection. *ACM Conference on World Wide Web*, New York, USA, 2004, pp.66–73.

[16] Erradi A, Maheshwari P. A broker-based approach for improving web services reliability. In *International Conference of Web Services*, Florida, USA, 2005, pp.355–362.

[17] Hayton R, ANSA Team. FlexiNet Architecture. 1999, http://www.ansa.co.uk.

[18] Cazzola W. Communication-oriented reflection: A way to open up the RMI mechanism [Dissertation]. Milano, Italy, 2001.

[19] Looker N, Jie Xu. Assessing the dependability of SOAP RPC-based web services by fault injection. In *Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (*WORDS*), Guadalajara, Mexico, 2003, pp.163–170.

Gang Huang is an associate professor in the School of Electronics Engineering and Computer Science, Peking University. His research interests are in the area of distributed computing with a focus on middleware, including the construction and management of middleware, and software engineering with a focus on component based development and software architecture.

Li Zhou is a master student in the School of Electronics Engineering and Computer Science, Peking university. His major research interests include software architecture, model checking, service-oriented architecture and agent technique.

Xuan-Zhe Liu is a Ph.D. student in the School of Electronics Engineering and Computer Science, Peking University. His research interests are in the area of service oriented architecture (SOA) with a focus on web services, dependable service delivery and enterprise service bus.

Hong Mei is a professor in Dept. Computer Science and Technology, Peking University. His current research interests include software engineering and software engineering environment, software reuse and software component technology, distributed object technology, software production technology, and programming language.

Shing-Chi Cheung is an associate professor in the Department of Computer Science, Hong Kong University of Science and Technology. His research interests are in the areas of software testing, pervasive computing, RFID based systems.