An Efficient Approach for QoS-Aware Service Selection
Based on A Tree-Based Algorithm

Minhyuk Oh, Jongmoon Baik, Sungwon Kang and Ho-jin Choi
School of Engineering
Information and Communications University
119 Munji-ro, Yuseong-Gu, Daejeon
Republic of Korea
{bread19, jbaik, kangsw, hjchoi}@icu.ac.kr

Abstract

Service Oriented Architecture (SOA) has become a new software development paradigm because it provides a flexible framework that can help reduce development cost and time. SOA also promises loosely coupled, interoperable, and composable services. In order to maximize these benefits, many researchers have focused on service composition techniques, especially in terms of providing quality of services (QoS) to consumers in a dynamic environment. In current literature, many approaches, such as the Genetic Algorithm, Integer Programming, and the Pinninger Algorithm approaches, have been proposed for service compositions to provide quality of services to consumers. However, these approaches are inefficient when any problem occurs during service executions since the problem causes the current execution stop, and service rebounding is required for the unexecuted path. In addition, even if a better service is discovered, it cannot be easily replaced in these approaches because replacing a single service causes a problem on global QoS constraints. In this paper, we focus on these problems by proposing a tree-based algorithm in order to provide optimal services to consumers consistently and efficiently even when the status of selected services have changed. As a result, consumers can always use the optimal services of high qualities.

KEY WORDS
Service Oriented Architecture (SOA), Service Selection, Service Composition, Binary Search Tree (BST), Quality of Service (QoS)

1. Introduction

Service Oriented Architecture (SOA) is a new paradigm for software development that promises loosely coupled, interoperable and composable components or software agents called services. It is also known to provide benefits such as cost-effectiveness, agility, adaptability, and leverage of legacy investment [1]. There are three main parties under SOA: service registry, service provider, and service consumer. A service provider registers its own services with detailed information such as their capabilities, interfaces, policies, and supported communication protocols to service registry, which is where the service consumer can discover the registered services [2]. Since services in SOA are implemented using XML-based standards such as SOAP, WSDL, SLA and UDDI [3], service integration, which has enormous potential in streamlining business-to-business or enterprise application integration [4], can easily be accomplished. Thanks to these benefits, a lot of researches are currently focusing on service composition techniques.

It is one of most important issues to providing good quality of services efficiently for service consumers with service composition. Many approaches are proposed such as using different selection algorithms [5, 6]. However, these approaches show to be inefficient when the status of selected services have changed, which might occur frequently in a dynamic environment like the Web. For example, if a selected service become unavailable or an incorrect estimated QoS is provided, this will cause each approach to stop the service execution, identify the workflow slice still to be executed and perform re-binding on that portion rather than re-binding only the stopped service [7]. In addition, since these approaches consider the set of the selected services as the one solution, a single service will not be exchanged even when a better service has been discovered from the registry.

In this paper, we address the problems mentioned above by proposing a tree-based algorithmic approach with considering QoS information. The tree-based algorithm helps provide optimal services consistently and efficiently to consumers even when the status of services has changed, enabling the consumers to use the quality of services consistently.

The rest of paper is organized as follows. In section 2, some related works are described. In section 3, a service
selection scenario based on a tree algorithm is described. In section 4, A QoS model is presented with QoS criteria and a normalization method. In section 5, a tree-based selection algorithm is presented. Section 6 presents the validation of our proposed approach. Lastly, the conclusion and future works are drawn.

2. Related Works

Many research papers proposed QoS-Aware service compositions and introduced the approaches such Genetic Algorithm (GA) approach, Pisinger Algorithm (PA) approach and Integer Programming (IP) approach to help select the optimal services. These approaches are briefly described below.

G. Canfora et. al. [5] proposed a service composition approach based on a genetic algorithm (GA). The genetic algorithm is used for finding a solution of the service composition while considering global constraints. Once the GA finds services that are requested in a service composition flow, a fitness function is applied. The fitness function computes the QoS value of the selected services and compares the results of the fitness function with the global constraints. If the QoS is satisfied and the stop criteria of the GA are met, the service composition stops with the found solution. However, if a problem occurs while executing the found solution such as the selected service not being available, or the estimated QoS differing from the actual QoS, the re-composition procedure will be performed for the unexecuted path.

L. Zeng et. al. [6] proposed a service composition approach by using the Integer Programming (IP) problem. The IP is used to select an optimal execution plan without generating all possible execution plans. The problem of selecting an execution plan is mapped into an IP problem, allowing the IP to find the appropriate execution plan with concrete services. This approach requires replanning when the QoS is updated, or if a service fails during the execution of services because the approach computes overall score for the execution plan. This implies that if one service is replaced, the overall score could possibly cause problems on global constraints.

In a dynamic environment like the Web, variations in service qualities such as reliability and availability are inevitable because of several factors. For instance, a loss of the Internet connection for service providers might be one of the factors. Thus, the service composition approach should consider these circumstances. However, this paper already identified that the related works do not much consider these issues. First, the above-mentioned approaches perform service re-composition for the remaining unexecuted services when the changes on the status of services are made. For example, the selected service becomes unavailable or the estimated QoS values are different from the actual QoS values. This happens because the above-mentioned approaches cannot simply target the failed service because they are designed to consider the global QoS values. In addition, the existing approaches cannot react to a new service efficiently because when they face a better quality service compared to the existing one, all of the unexecuted services are re-selected to ensure that the set of selected services does not deviate from the global QOS constraints.

3. Service Selection Scenarios

This section describes a service selection scenario using a tree-based algorithmic approach. In this scenario, a new service consumer and a service registry that are different from the normal consumer and registry are introduced. The new service registry contains another repository that keeps the consumer information about which services are being used by which consumers. This information is maintained for two reasons: (1) to retrieve the consumer’s service trees and enable other service consumers to share them; and (2) to keep information consistent between service consumers and service registry. The new service consumer contains service trees in its cache in case of service failure, changes due to the changes in QoS values or the discovery of new service. Therefore, the service consumer re-selects the optimal service node to use the service consistently and dynamically.

![Figure 1: Service selection scenario](image)

With the new service consumer and service registry, a service selection can be performed according to the following process:

1) Service providers register their own services to a service registry. Each registered service in the service registry has information about the capability, QoS data of service, interfaces, and supported communication protocols.
2) The service consumer requests services based on the functionalities and the QoS information.
3) The service registry searches for consumers who use the requested services in a repository.
3-1) If there is a match, the service registry copies the service trees from the consumer and sends them to the requester.
3-2) If there is no match, the service registry discovers the requested services and sends the set of discovered services to the requester.
4) After the requester receives service trees or constructs service trees with received the set of services, the right most service node in the tree is selected as the optimal node. This is because the BST always places the greatest value on that side of the tree.
5) After the services are executed, the service consumer provides a feedback on the service usage such as QoS information.
6) Next, the service registry identifies service consumers who use the service that has been updated recently. Once they are identified, updated information is sent to them.
7) Finally, service consumers who receive changes from the service registry update their service trees and reselect the optimal services if necessary.

As described in this scenario, the service consumer contains service trees, which are generated according to the score of services. The score can be measured from the normalized QoS values with weights that are decided by the consumers. The details will be discussed in section 3.2. The structure of the service tree is similar to a binary search tree (BST). The structure is used because it provides an efficient method of data storage and organization [8, 9].

4. QoS Model

Current service selection and composition are often done through the service registry that only captures the common Quality of Service (QoS), but not the personal service evaluations based on the consumer’s feedback [10, 11], and it does not consider the volatile characteristics of QoS either [12]. Moreover, the published QoS information by the service providers is subjective [13].

In order to consider these issues, we need to monitor the actual QoS while the service is being used and receive the feedback from the service consumers. In this paper, an extensible QoS model is used to deal with dynamic QoS values and various kinds of QoS. The extensible QoS model, QoS = \{Q_1, Q_2, ..., Q_n\}, represents the set of QoS criteria where Q_i presents single QoS information [14]. For example, if there are 6 QoS criteria, such as performance, cost, reliability, availability, reputation and fidelity, it can be represented by using the extensible QoS model as follows:
\[
\text{QoS (S)} = \{Q_{perf}, Q_{cost}, Q_{rel}, Q_{avail}, Q_{repr}, Q_{fid}\}
\]
This model can be easily customized in accordance with services without fundamentally altering the service selection technique [13].

4.1. QoS Criteria for A Service

QoS criteria for different domains may be different. To be more generic and precise, we consider 6 criteria: performance, cost, reliability, availability, reputation and fidelity.

- **Performance**: The performance Q_{perf}(s) is the time duration (turn round time) from a request being sent, to when the results are received.
- **Cost**: It refers to the amount of money that the consumer pays for using a service, Q_{cost}(s).
- **Reliability**: The reliability Q_{rel}(s) is the probability that the requested service is working without a failure within a specified time frame [15].
- **Availability**: The availability Q_{avail}(s) is the quality aspect of whether the service is present or ready for immediate use [15].
- **Reputation**: The reputation Q_{repr}(s) is the criterion in measuring total trustworthiness of a service.
- **Fidelity**: The fidelity Q_{fid}(s) is the average marks that are given by different consumers to the same QoS criterion.

4.2. Normalization

All criteria use the different measurements. For example, the unit of cost is dollars, while the unit of performance is seconds. These units should be normalized in order to compute the single score from the QoS values that have different measurements. One popular methodology, standardizing QoS values between 0 and 1 [13] is used. With the normalized values of quality data, we are able to compute quality scores for services to see which service has a better quality. The formula for computing scores is as follows [14]:
\[
\text{Score (S)} = \sum_{i=1}^{n} W_i \times Q_i
\]

Where n is the number of QoS data for a service, Q_i is the normalized value of QoS data, and W_i is the weight for each QoS data that is decided according to the consumer’s preference. The sum of W_i always becomes 1 in order to set the score to the range between 0 to 1.

5. Tree-Based Selection Algorithm

As described earlier, a binary search tree (BST) is adopted to represent the requested nodes with quality information since it provides an efficient method of data
storage and organization [8]. Unlike other approaches, the BST does not require the global QoS constraints since it provides the optimal services for the set of the abstract services in a service composition.

The following section will show how a service tree can be constructed based on services’ quality scores and how an optimal service is found and section 5.2 presents how the reselection can be performed in the tree.

5.1. Binary Search Tree Construction

In the previous section, the normalized values of QoS data and the computation of quality scores are presented. With the scores, a BST can be created. Figure 2 shows an example of service scores and the constructed service tree based on the values in the table.

<table>
<thead>
<tr>
<th>Service</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
<th>s6</th>
<th>s7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>0.87</td>
<td>0.68</td>
<td>0.92</td>
<td>0.90</td>
<td>0.54</td>
<td>0.97</td>
<td>0.75</td>
</tr>
</tbody>
</table>

![Figure 2: a BST example](image)

When a service tree is constructed, the scores are used to locate the service node’s position. For example, if a service node’s score is bigger than that of the compared node, it is compared with a right child of the node, but if its score is smaller or equal to the node, it is compared with a left child. If there are no nodes to compare to, the location of the service node becomes where it is currently residing.

```
//Purpose: Re-select the optimal service
//Input: the currently selected node node
//Output: the optimal service node

reselectService (node)
start traverse the service nodes from the node position - 1
Traverse as the following order: right, root, left
if current service node value <= node.value then
    return the service node
else
    return null
```

![Figure 4: reselectService() operation](image)

For scenario 3, there are two situations where the current selected service no longer remains as the optimal service. The first situation is where the current or other service’s QoS values are updated.

```
//Purpose: update the value of the service node
//Input: the updated node node, QoS data values
//Output: the tree with updated QoS

updateTree(node, value)
ServiceNode service = Current Optimal Service
delete (node, root)
node.updateQoS(values)
insert(node, root)
if (service == node) then
    if service is not optimal then
        reselect an optimal node
    else if (service.getScore() < node.getScore()) then
        replace the optimal node as node
```

![Figure 5: updateTree() operation](image)

For example, if a service receives a bad feedback, the computed score might not be the best one in the service tree, or if other service receives a good feedback from other consumers, the overall score might become the best one in the service tree. The figure 5 shows the pseudo code for updating the service tree according to the changes of the QoS values.

However, when the second situation, the discovery of a
new service, takes place, updateTree operation does not need to be performed since the new service can simply be inserted into the service tree and checked to see whether it is located under the right child of currently selected service. If the newly inserted service is located under the right child, then we know it is the most optimal service in the tree. Otherwise, there is no need to consider it.

6. Validation

In this section, the performance of our approach is analyzed by comparing our approach with other approaches such as Genetic Algorithm and Integer Programming approach. To give fairness, their experiment scenarios and the performance results for service composition are used. In addition, equal weights to the different QoS attributes are assigned in the score function for simplicity, and individual QoS values are generated randomly in order not to make the service tree(s) that are optimized for the new approach.

The experiments adopt pseudo codes of a normal BST algorithm from [16, 17, 18], implemented them in Java, and the proposed pseudo codes described in the previous section, designed the consumer to have BST data structure for each set of services. The experiments are also designed to let Service Registry to have the consumer's information so that it can reuse the constructed service trees among service consumers.

All experiments were performed on a 3.0 GHz Intel Pentium™, 512 Mb of RAM, Microsoft Windows XP™ and J2SDK 1.5 to construct the same environment for GA and IP for fairness reasons.

6.1. Service Re-Composition for GA, IP with BST

To compare performances of GA, IP and the BST approaches for service re-composition in case of service failure, let us assume that service failure occurs in the first service. Under this assumption, it is ensured that service composition time would be the same as the service re-composition time for GA and IP, because if any problems occur during execution, GA and IP would stop the current execution, identify unexecuted path and perform rebinding for unexecuted path. Hence, if a problem occurs in the first service in the execution path, the unexecuted path would be the same as the original execution path for service composition.

Figure 6 shows the results of the experiment for the three different approaches. When a number of concrete services increase, the performance for re-composition of IP becomes slower, but GA and the BST approach consistently provide the similar performances no matter how many concrete services are available for the abstract service. However, since GA re-selects all services for unexecuted path while the BST approach re-selects only the failed service, the BST approach provides much better performances on service re-composition.

![Figure 6: Performance comparison between GA, IP and BST for reselection](image)

In addition to figure 6, another experiment was performed under the assumption that a service failure occurs randomly through all composed services in order to show that the BST approach is not only efficient when a service failure occurs in the first service, but also for any other services because the previous experiment only analyzed the BST approach for service re-composition when a failure occurs in the first service.

![Figure 7: Performance analysis for service failure](image)

Figure 7 shows the result of this experiment. Both results show the efficiency of the BST approach compared to others. This result is significant since a failure described above could occur often in the SOA environment.

7. Conclusion and Future Works

The QoS-aware service selection for service composition is an active research area in Service-Oriented
Architecture (SOA). The primary drawback of current works in dynamic service composition is the inefficiency against the service changes. This paper has proposed a tree-based algorithmic approach to solve this problem efficiently. For example, when a selected service is failed, an estimated QoS value is different from the actual one, QoS values are updated according to consumers' feedback, or a better service is discovered, the tree-based approach can simply replace the service with new one promptly, so that consumers can receive the quality of services consistently.

In addition to that, the new approach always provides the optimal service from the set of existing services according to consumer's preference, because the new approach constructs the service tree for each requested service and selects the service with the best capability. Consumer's preference is a portion of weight for each of QoS criteria, so there is no need to consider the global QoS constraints for service composition. The global QoS constraints are hard to set, because if they are set too narrowly, appropriate services might not be found, or if they are set too broadly, selected services' quality might be very low.

The proposed approach guarantees that consumers receive the quality of services within average of O(log N) time when the currently selected services are not the optimal ones by updating the service rather than stopping the execution and rebinding for the rest of unexecuted path. Therefore, service consumers receive the services consistently, efficiently and promptly.

However, as a number of candidate services per abstract services in a workflow increases, the amount of memory allocation spaces will be increased as well as the service composition time, as shown in the figure 6. Therefore, our future works will focus on a way to reduce the service set, because by reducing service set, the new approach can improve the service composition time and reduce the allocated memory space. Additionally, we will perform more experiments to show that the approach always selects the best quality of service within the given services.

References


