Rapid Selection of Services based on QoS

Aimrudee Jongtaveesataporn
Bank of Tokyo Mitsubishi UFJ
Bangkok, Thailand
aimrudee@doi.ics.keio.ac.jp

Shingo Takada
Keio University
Yokohama, Japan
michigan@ics.keio.ac.jp

ABSTRACT

Modularity takes many forms depending on the programming paradigm. We focus on Service Oriented Architecture (SOA), where the primary module unit is a service. Service selection is a key part of SOA, and it is primarily done based on function; but Quality of Service (QoS) is rapidly gaining importance. Existing dynamic service selection approaches recompute the request each time which may be unnecessary. If the same service is chosen as having the "best" QoS for various selections, then that service may end with too much load. We propose the FASICA (FAst service selection for SImilar constraints with CAche) Framework which chooses a service with satisfactory QoS as quickly as possible. The key points are (1) to use a cache which stores previous search results, (2) to use K-Nearest Neighbor (K-NN) algorithm with K-d tree when a satisfactory service does not exist in the cache, and (3) to distribute the service request according to a distribution policy. The results of a simulation show that our approach can rapidly select a service compared to a conventional approach.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Design

Keywords

service selection, service oriented architecture, QoS, K-Nearest Neighbor

1. INTRODUCTION

A wide variety of modules exist. We focus on Service Oriented Architecture (SOA), where the primary module is a service. A system based on SOA will normally consist of multiple services that are connected together. These services are often provided by third parties, i.e., the developers of services are different from developers of SOA-based systems.

The number of published Web services has increased exponentially for a decade [2]. Thus, an efficient mechanism is necessary to select a service from such a set of services that matches both functional and non-functional requirements. Service selection is an important step in service composition, since the selected services are the parts that are to be composed together. If the composition is to be done online and at runtime, an efficient selection algorithm is especially important since we cannot let the client wait too long for a response. Non-functional requirements, or QoS, are key to reducing the number of service candidates.

Current service selection approaches consider one service request at a time and select the service with the best QoS. The selection does not pay attention to multiple requests for services with the same function. The same service (with the best QoS) will be selected even though there may be other usable services leading to possible issues of processing overload and degradation of quality. Indeed, all users do not always need the service with the best QoS. For example, a user may want a service that responds fairly quickly, and do not need the fastest service (which may incur extra fees). Services with high QoS should be served to users who have high QoS requirements.

In this paper, we focus on the selection of services based on QoS constraints that is fast enough for runtime computation and resolves conflicts among requests for the same service. We consider using previous solutions to resolve incoming service requests as it has been estimated that approximately 85% of incoming service requests have been previously resolved by other services [13].

The main contribution of this paper is to propose the *FASICA (FAst service selection for SImilar constraints with CAche)* framework, which manages frequently occuring similar requests and distributes these service requests among similar services. Our approach guarantees that a selected service will satisfy the service requester because the QoS values of the target service will be no worse than the QoS constraints. More specifically, the contributions of this paper are as follows:

1. To present a cache based approach to manage and retrieve previous solutions for speeding up the service selection.
2. To propose an approach based on K-Nearest Neighbor algorithm based on K-d tree that is used when a satisfactory service does not exist in the cache.

BICT 2014, December 01-03, Boston, United States

DOI 10.4108/icst.bict.2014.257855

3. To distribute the service request according to a distribution policy so that requests do not concentrate on a particular service.

4. To show simulation results indicating the validity of our approach.

Our mechanism can be applied to service composition and service substitution when searching for a group of candidate services that can replace a broken service. It can be adapted to use in Enterprise Service Bus (ESB) and cloud service environment. Our FASICA framework can be plugged into an ESB to get a service for each request at runtime. In cloud computing, the performance of Web services, i.e., QoS, may fluctuate due to the dynamic Internet environment. Therefore, our FASICA can also be applied in the cloud environment for quick responses in service selection from a set of functionally equivalent candidates.

The remainder of this paper first discusses related work. The QoS based model is defined in Section 3. Section 4 describes our FASICA framework. Section 5 evaluates our framework. Section 6 makes concluding remarks.

2. RELATED WORK

Many approaches have been proposed to select services based on QoS in service composition with different computation speed. Most work focus on a single request for a Web service. Zeng et al. [19, 20] considered multiple criteria (e.g., price and reliability) for dynamic and quality-driven service selection. They use linear programming to find the best web service for the composition. Ardagna et al. [5] proposed an approach based on mixed integer linear programming for dynamic service selection. They looked for a global optimal solution instead of local optima or suboptima.

Yu et al. [17, 18] proposed algorithms based on a combinatorial model and a graph model. Their objective was to maximize an application-specific utility function under end-to-end QoS constraints. Extensive simulations suggest its practicality, but computation speed could still be improved.

Al-Masri and Mahmoud [1] introduced the Web Service Relevancy Function (WsRF) used for measuring the relevancy ranking of a particular Web service based on QoS metrics and client preferences. The greedy method of Alrifai's work [3] extracts QoS levels from the QoS information of service candidates. They propose a hybrid approach that combines global optimization with local selection in order to find Web services with maximum utility. These work focus on selecting the best web service without considering the problem when multiple users request for the same web service at the same time.

The Skyline technique [7] has been applied to service selection by efficiently pruning candidate services. Alrifai et al. [4] proposed using representative Skyline services for composition. They used a utility function to evaluate the overall, multidimensional QoS of a given service. However, their work still only considers the best utility service without distributing the request load. Pan et al. [14] proposed the Preprune-Refine framework inspired by MapReduce and Skyline techniques for efficient selection from large scale distributed web services. However their framework is suitable for Web service selection in distributed environment only.

There are also work concerned with multiple requests for the same functional service waiting in a queue at the same moment. Dyachuk et al. [9] focus on the request scheduling

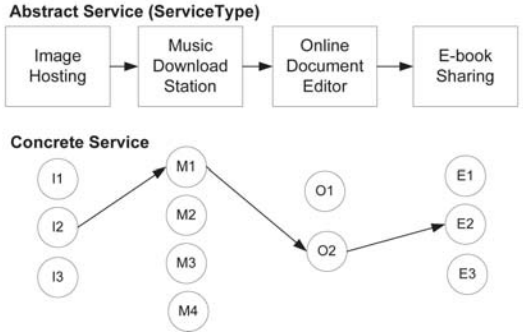

Figure 1: Atomic service selection

for improving composite Web services performance in the service request overload situations by applying scheduling policies such as Shortest Job First (SJF) and Least Work Remaining (LWKR). Shahand's work [15] presents a dynamic web service scheduling and deployment framework (DynaSched). Jongtaveesataporn, et al. [11] enhanced the ESB capabilities by supporting load balancing request of the same Service Type. This work improves the performance of the whole system by preventing the request overload problems but do not consider cases where the same QoS request occurs frequently. Kang and Liu [12] presented the Global Optimal Service Selection for Multiple Requesters (GOSSMR) algorithm with the Skyline technique for selecting the target service and solving the conflicting service requests. However the complexity of GOSSMR is O(N!/(N-M)!) (where N is the number of services and M is the number of requests), the selection time increases rapidly when the number of services becomes large. Even though they try to use the Skyline technique with GOSSMR for reducing the number of candidate services, their mechanism needs recomputation for every incoming request.

3. QOS BASED SERVICE MODEL

In our QoS based service model, each service is either *concrete*, i.e., an actual service, or *abstract*, i.e., essentially a grouping of similar concrete services. An abstract service can also be considered as a *Service Type* [11]. The basic idea is that services of the same Service Type have different QoS values but are interchangeable. For example, in Fig. 1, I_1, I_2, and I_3 are concrete services for the abstract Image Hosting service or Image Hosting Service Type. Our approach assumes that all concrete services are grouped into Service Types.

Each service may have one or more quantitative non-functional property in the form of a *QoS attribute*, such as response time, availability, and cost. A QoS attribute can be positive or negative. A *positive* QoS attribute needs to be maximized (e.g., availability, throughput and reliability). A *negative* QoS attribute needs to be minimized (e.g., response time and cost). The QoS values for a service is represented by the vector $d =< d_1, ..., d_i, ..., d_n >$, where d_i is the value of the ith QoS attribute. This vector is published along with the service itself.

We assume that the actual QoS values of Web services can be obtained dynamically through a QoS broker, service mid-

Algorithm 1 Overview of K-d Tree Construction

1. NormalizedQoSServices = NormalizeQoS(Services)
2. if(S>n) // S = # Services, n = threshold value
3. SkylineServices = SLSelect(NormalizedQoSServices)
4. tree = BuildKdTree(SkylineServices, 1)
5. else
6. tree = BuildKdTree(NormalizedQoSServices, 1)

Algorithm 2 K-d Tree Construction

BuildKdTree(N, depth)
Input: N, depth // A set of nodes (services) and depth
Output: V // The root of a K-d tree storing N

1. if N contains only one node, then return that node
2. qos = depth mod k // choose a QoS attribute to focus on
3. V.data = node with median value for chosen attribute
4. N1 = set of nodes whose QoS value is less than median
5. N2 = set of nodes whose QoS value is greater than median
6. V.left = BuildKdTree(N1, depth+1)
7. V.right = BuildKdTree(N2, depth+1)
8. return V

dleware or an extended module of ESB. This is important as the QoS values of a Web service may change dynamically such as in a cloud environment due to server hardware/software update, workload change of the servers, etc.

From the requester's viewpoint, the non-functional properties in service requests are described as *QoS constraints*, and represented as a vector $r =< r_1, ..., r_i, ..., r_n >$, where r_i is the requested value of the ith QoS attribute.

4. FASICA FRAMEWORK

We propose the FASICA framework for selecting a service. It is based on the following three basic steps, which will be described in detail in the rest of this section.

1. Construct a K-d tree for each Service Type
2. Generate a list of candidate services for a given request
3. Choose the actual service from the candidate list based on runtime policy

We chose K-d tree as the data structure as it is suitable for supporting search based on multi-dimensional data, such as QoS attributes in our case.

4.1 K-d Tree Construction

The first step in the FASICA framework is the construction of a K-d tree for each Service Type. We assume that all concrete services are grouped into Service Types. K-d tree is a multidimensional binary search tree where K is the dimension of the search space [6]. In the FASICA framework, K is the number of QoS attributes and each node corresponds to a service. The K-d tree is reconstructed when services are added/deleted, QoS values change, etc. The basic algorithm is shown in Algorithm 1, which we shall now describe.

Since services will have multiple QoS attributes, each having a completely different scale, we normalize them to a common scale (Line 1). In the scaling process, each QoS attribute value is converted into a value between 0 and 1, by comparing it with the minimum or maximum possible value according to the available information. The formula for normalization is shown in Equation (1). Note that the equation differs for positive and negative QoS attributes. The normalized QoS values of a service is represented as a vector $q =< q_1, ..., q_i..., q_n >$.

$$q_i = \begin{cases} \frac{d_i - min(d_i)}{max(d_i) - min(d_i)} & \text{if } d_i \text{ is a positive attribute} \\ \frac{max(d_i) - d_i}{max(d_i) - min(d_i)} & \text{if } d_i \text{ is a negative attribute} \end{cases} \quad (1)$$

where $d_i =$ value of the i-th QoS attribute for the service under consideration
$min(d_i) =$ minimum value of the i-th QoS attribute of all services
$max(d_i) =$ maximum value of the i-th QoS attribute of all services
$q_i =$ normalized value of the i-th QoS attribute for the service under consideration

If the number of services is greater than a certain threshold value (Line 2), we filter out services with low QoS based on the notion of Skyline to reduce the number of candidate services and improve the speed of future searches (Line 3). We follow the approach proposed by Alrifai, et al. [4] to identify concrete services (Skyline services) that are not dominated by other services in the same Service Type. Service S_i is said to dominate another service S_j, if each of the QoS values of S_i is better than or equal to the corresponding QoS values of S_j, and at least one QoS value is strictly better. For example, in Table 1, response time is a negative attibute. Throughput and availability are positive attributes. Then, services B, C and D are Skyline services because none of the other services dominates them. Even though response time in service A is worse than services C and D, the throughput of service A is better than services C and D. Therefore service A is a Skyline service. On the other hand, services E and F are both dominated by service A; thus services E and F are not Skyline services.

Finally, a K-d tree is constructed for the original services (Line 6) or the Skyline services (Line 4). The actual K-d tree construction is based on the algorithm by [6] and is shown in Algorithm 2. The main idea of K-d tree is to split the space into two subspaces according to the splitting hyperplane of the node. A K-d tree will have K coordinates, with each coordinate splitting the nodes (i.e., services) into two sets. It is possible to show a K-d tree in a two dimensional space [6].

We take Fig. 2 as an example. This figure shows a K-d tree where K=2, i.e. we use two QoS attributes response time and throughput. Each node in the tree shows the node ID and the QoS value, e.g., S9 has 0.25 as the response time value and 0.85 as the throughput value. When showing K-d tree in a two dimensional space, we alternate between each QoS attribute (Line 2 of Algorithm 2). Initially, we search for the median response time value which is 0.25, and then set node S9 as the root node. The rest of the nodes are split into two sets based on the response time; nodes with response time less than 0.25 belong to the left sub-tree, while those greater than 0.25 belong to the right sub-tree. Next, the median throughput value in the left subtree is searched for, resulting in 0.95. S1 is chosen as the root for the left sub-tree and the remaining nodes are split based on the throughput value. This is recursively done until the tree shown in Fig. 2 is obtained.

Table 1: QoS value examples for Skyline, Similar Vector, and QoSDist computation

Service	qos_1 Response Time	qos_2 Throughput	qos_3 Availability	Skyline Service	$QoSDist$
A	0.28	0.71	0.88	✓	$\sqrt{0.01^2 + 0.07^2 + 0.07^2} = 0.099$
B	0.30	0.78	0.83	✓	Infinity
C	0.25	0.65	0.85	✓	$\sqrt{0.04^2 + 0.01^2 + 0.04^2} = 0.057$
D	0.26	0.68	0.92	✓	$\sqrt{0.03^2 + 0.04^2 + 0.11^2} = 0.121$
E	0.38	0.55	0.81	✗	
F	0.29	0.70	0.82	✗	
SV of $SL < A, B, C, D >$	sv_1=$max(qos_1)$ 0.30	sv_2=$min(qos_2)$ 0.65	sv_3=$min(qos_3)$ 0.83		

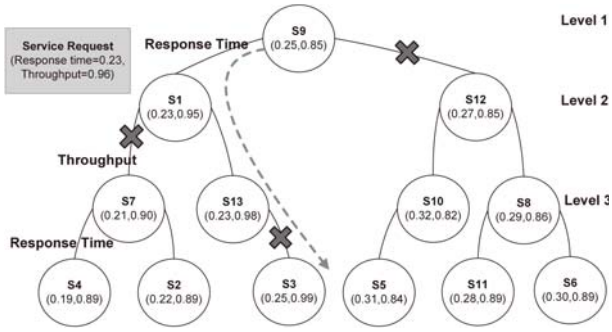

Figure 2: K-d tree example

Algorithm 3 FASICA Candidate List Generation

```
Input: RQoS // QoS constraints of service request
Output: listOfServices // list of candidate services
1. if(similarRequest(RQoS))
2.   listOfServices = AnswerCacheRetrieve(RQoS)
3. else
4.   listOfServices = tree.SearchKNN(RQoS)
5. return listOfServices
```

A K-d tree based approach can handle many QoS attributes efficiently, as searching the tree-based data structure will take O(log(n)) time, where n is the number of services. This is more efficient than GOSSMR method [12] that solves a similar problem. We will describe the details of searching with K-d tree in section 4.2.2.

4.2 Candidate List Generation

When a request arrives, we generate a list of candidate services. It can be generated in two ways (Algorithm 3): (1) from an *Answer Cache* that holds the historical search results, or (2) from the K-d tree.

First, we check the *Answer Cache* which holds past service search results, to see if any past results can be used (Algorithm 3 Line 1). If such an "answer" exists, then a list of candidate services from a previous search is returned. Details will be given in Section 4.2.1. If no such similar request exists, then K-Nearest Neighbor algorithm is applied to the K-d tree to obtain a list of candidate services (Line 4). Details will be given in Section 4.2.2.

4.2.1 Answer Cache

The Answer Cache holds *Service Answers (SAs)* from previous searches, each consisting of a *Similar Vector (SV)* and a *Service List (SL)* as shown in Equations (2) and (3). The Similar Vector and Service List for Service Answer i are represented as $sv =< sv_{i1}, sv_{i2}, ..., sv_{iq} >$and $sl =< sl_{i1}, sl_{i2}, ..., sl_{ir} >$ respectively.

$$AnswerCache = \{sa_1, ..., sa_i, ..., sa_p\} \quad (2)$$

$$sa_i =< SV < sv_{i1}, ..., sv_{iq} >, SL < sl_{i1}, ..., sl_{ir} >> \quad (3)$$

where $sa_i =$ ith Service Answer in the Answer Cache
$sv_{ij} =$ computed QoS value of qos_j (explained below)
$sl_{ik} =$ concrete service whose QoS values satisfy the QoS constraints

The size of the Answer Cache has a maximum value (cache size). When the Answer Cache becomes full, i.e., the number of answers is the same as the cache size, the oldest Service Answer is deleted.

Similar Vector (SV).

A Similar Vector is a list of QoS values each of which corresponds to one QoS attribute qos_j. The value depends on if the QoS is a positive or a negative attribute. If it is positive, then the smallest QoS value of the services in the Service Answer is chosen. If it is negative, then the largest QoS value is chosen. In other words, we take the worst QoS values for the Similar Vector.

For example, suppose services A, B, C, and D in Table 1 are included in a Service List in one of the ServiceAnswers, i.e., $SL =< A, B, C, D >$. Since response time is a negative QoS attribute, the largest response time value (0.30) is chosen as the value for the Similar Vector. Similarly, since throughput and availability are positive QoS attributes, the smallest values are chosen, i.e., $SV =< 0.30, 0.65, 0.83 >$. Thus, $ServiceAnswer =< SV < 0.30, 0.65, 0.83 >, < SL < A, B, C, D >>$.

The reason why we use the worst QoS values for the Similar Vector is so that we can use it as a threshold when searching in the cache for a suitable request. For example, if the service request requires the normalized response time that is less than or equal to 0.33, by setting the Similar Vector response time value to 0.30, we know that (at least for response time) the four services A, B, C, and D can become

candidate services in the current search.

Request Comparison.

The process for finding a candidate service list in Answer Cache consists of two steps. The first step is to filter out any Service Answer whose Similar Vector (SV) value conflicts with the request. For example, if the response time request was 0.21, we can filter out the Service Answer which has the Service List $SL < A, B, C, D >$ in Table 1.

The second step is to compute the difference between the requested QoS and SV, and compare the result against a threshold. The difference is calculated as a distance by using the following Similar Distance Equation (4).

$$similarDist(r, sv) = \sqrt{\sum_{i=1}^{n} w_i \left(r_i - sv_i\right)^2} \tag{4}$$

where $r_i =$ i-th QoS constraint of service request
$sv_i =$ value for i-th QoS in Similar Vector
$w_i =$ weight for i-th QoS
$n =$ number of QoS attributes

The weight w_i is used if a certain QoS is considered to be more important than others. Otherwise, $w_i = 1$.

The first Service Answer whose $similarDist$ value is less than a specified threshold *Similar Distance Threshold (SDT)* (Equation (5)), is taken as the result of the cache search, and the corresponding Service List is returned as the list of candidate services for the request.

$$similarDist(r, sv) \leq SDT \tag{5}$$

Reusing previously found services from the Answer Cache can reduce the computation time for service selction. Note that the SDT is related to the number of QoS attributes. If the number of QoS attributes increases, the SDT value should also increase. We take this greedy approach of returning the first Answer Cache that satisfies Equation (5) to speed up the search.

4.2.2 K-Nearest Neighbor

If there are no suitable list of services within the Answer Cache, we apply the K-Nearest Neighbor (K-NN) algorithm [8] to the K-d tree. We use K-NN because we can guarantee that all candidate services will satisfy the request. Furthermore, operations such as insert and search on a K-d tree using K-NN algorithm is efficient with the time complexity being O(log(n)).

K-NN is a simple machine learning algorithm, with no (or minimal) explicit training phase making it fast. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its K nearest neighbors. K is normally a small positive integer. If $K = 1$, the object is simply assigned to the class of its nearest neighbor. We adapt this algorithm to use in our service selection. Note that the K in K-d tree and the K in K-NN are different; the former being the number of QoS attributes, and the latter being the number of similar services for a given request.

We use QoS Distance (QoSDist; Equation (6)) to compare the similarity of a service request and a service.

$$QoSDist(r, q) = \begin{cases} \sqrt{\sum_{i=1}^{n} w_i \left(r_i - q_i\right)^2} & \forall i \; r_i \geqq q_i \text{ if negative QoS and} \\ & \forall i \; r_i \leqq q_i \text{ if positive QoS} \\ Infinity & otherwise \end{cases} \tag{6}$$

where $r_i =$ value of i-th QoS constraint
$q_i =$ value of the service's i-th QoS attribute
$w_i =$ weight for i-th QoS
$n =$ number of QoS attributes

QoSDist is the distance between the QoS constraints of a request and the QoS values of a service. It is a Euclidean distance similar to Equation (4). Note that when (one of) the QoS value of a service does not satisfy the request, this distance is set to infinity and the service is no longer a candidate. This can help to prune the services which do not satisfy the QoS constraints, reducing the time to search. The weight value of each QoS (w_i) can be changed depending on the importance of the QoS.

For example, we compute the $QoSDist$ between the constraints of request No.1 in Table 2 and QoS values of services A, B, C, and D (Table 1). We handle each QoS attribute equally, i.e., w_i is set to 1. Since response time is a negative QoS attribute, the QoS distance for Service B is set to $infinity$ because its response time 0.30 is greater than the requested 0.29. Table 1 indicates that the QoS values for service C is most similar to the request.

The actual computation is done while traversing the K-d tree (and not in a table) so that we do not need to calculate the QoSDist values for all of the nodes. Starting from the root node of the K-d tree, the QoSDist is calculated while traversing the tree. If a node's QoSDist value is infinity, then the subtree having that node as the root is pruned and not checked. The computation is continued until a set of K services whose QoSDist values are the smallest (i.e., the K services that best match the request) is chosen as the set of candidate services.

We take the example in Fig. 2 again, where the K-d tree is based on two QoS attributes, i.e., response time and throughput. Suppose the service request is specified as $(responsetime, throughput) = (0.22, 0.96)$. At level 1 (i.e., the tree root), we first consider response time. Since all nodes in the right branch have a response time value greater than 0.25, i.e., the QoSDist values are infinity, the right branch can be pruned. Next, the left branch is checked starting with node S1. Since S1 is at level 2, we consider throughput. Since all nodes in the left branch of S1 have a throughput value less than 0.95, the left branch of S1 can be pruned. Next, the right branch of S1 is checked starting with node S13. The right branch of S13, i.e. node S3, has a response time of greater than 0.22, and thus can be pruned. Therefore, S13 is a candidate service for this request.

Although the above example resulted in just one candidate service, the number of candidate services that is returned depends on the value of K in K-NN. For example, when we are conducting a K-NN search where K=3, the three services that best matches the QoS constraints are searched for.

Table 2: Service request example

Request No.	Response Time	Throughput	Availability
1	0.29	0.64	0.81
2	0.31	0.70	0.82
3	0.28	0.63	0.84
4	0.28	0.60	0.85
5	0.30	0.65	0.85

Table 3: Answer Cache example

Service Answer No.	Similar Vector (SV)	Service List (SL)
1	<0.28, 0.65, 0.85 >	<A, C, D>
2	<0.30, 0.71, 0.83>	<A, B>

4.3 Runtime Service Selection

The final decision as to which service to send the request to is chosen from the candidate services based on a specified distribution policy. We call the selected service as a *target service* in this paper.

In our current implementation, we employ a modified round-robin approach, where a service is chosen *in turn* from the candidate service list. An exception is made if that service has been chosen "recently". We prepare a *Sent Queue* to hold information on services that have been chosen. We describe this by using Table 2 which shows an example of service request. The QoS constraints of the first service request are (0.29, 0.64, 0.81). Since this is the first request, there is no Service Answer in the Answer Cache. The K-d tree is searched resulting in services A, C, and D of Table 1 being chosen as candidates, and service A is chosen based on the modified round-robin policy. The Service Answer for the first request is

$$SA1 =< SV < 0.28, 0.65, 0.85 >, SL < A, C, D >>$$

and is stored in the Answer Cache (Table 3). Furthermore, service A is put into the Sent Queue (Fig. 3).

The QoS constraints for the next request is {0.31, 0.70, 0.82}. Since the Similar Vector of $SA1$ does not satisfy the request, the K-d tree is searched. The candidate services of the second request are services A and B of Table 1. In a normal round-robin, service A would be selected again because the list of services for the second request is different from the first request. But in our modified round-robin policy, since service A is in the Sent Queue, we choose service B instead. Then service B is added to the Sent Queue (Fig. 3).

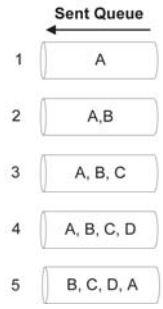

Figure 3: Sent Queue

If the third, fourth, and fifth requests are all found to be similar to SA1 in the Answer Cache, the target service would be services C, D, and A, respectively. The contents of the Sent Queue changes as shown in Fig. 3.

Some other possible distribution policy could be random, minimum, and threshold. A random distribution policy would mean that the service is chosen randomly from the candidate service list. A minimum distribution policy would mean that the service which has served the least number of requests is chosen from the candidate service list. A threshold distribution policy would mean that a particular service is continuously chosen from the candidate service list until a threshold value is reached. Once the threshold value is reached, subsequent requests are sent to another target service in the candidate list, which is chosen based on modified round-robin policy. This next target service will be used until its threshold value is reached. Then the third target service is selected based on modified round-robin policy, and so on.

5. EVALUATION

We evaluate our approach through a simulation, which we conducted on a computer with Intel Core i5 CPU, 4GB RAM running Microsoft Windows 7 (32-bit). For K-d tree library, we used Weka 3 data mining software in Java [16].

5.1 Generation of Services

We generated 3013 Skyline services using the Random Dataset Generator available at the SKYLINE Operator Evaluation Project Home Page [10]. Each service has six attributes each of which represents a QoS value such as response time, throughput, availability, accessibility, cost and security. Note that response time and cost are negative attributes, while the others are positive. Furthermore, as we use six attributes, the K in K-d tree is set to 6. Our evaluation uses several values of K, but this K will refer to the K in K-NN and not K in K-d tree.

5.2 Generation of QoS Constraints for Requests

We implemented a program (Similar QoS Parameter Request Generator; SQPR Generator) to generate the QoS constraints of the requests to be used as input in our simulation. The QoS constraints are in the range between 0 and 1. We weight all QoS attributes of all service constraints equally and use six QoS attributes: response time, throughput, availability, cost, accessibility and security. First the SQPR generator generates a random value between 0 and 1 as a base point for each of the six QoS attributes. This base point is used as the center of a hypersphere. Then the SQPR generator sets the radius of the hypersphere and generates a group of random points inside the hypersphere. We set the radius to 0.1, so that the QoS values will not take a wide range of values, resulting in requests that are similar. Specifically, we generated values for the six attributes, forming 20 similar groups, each group having 500 requests. Therefore there are 10,000 (20×500) similar QoS constraints that we use for service requests in our simulation. All service

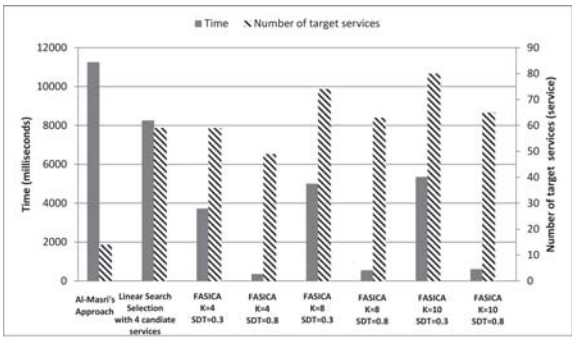

Figure 4: Comparison with other approaches

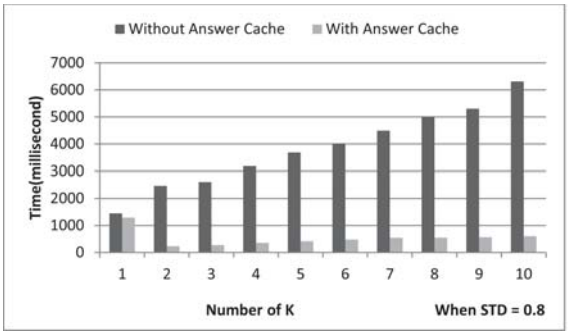

Figure 5: Computation time with and without Answer Cache

requests are sent to our prototype system and processed in sequence one by one.

5.3 Simulation Result and Discussion

We conducted two types of simulations:(1) Comparison with conventional approach and (2) Effect of the Answer Cache. Note that in each simulation, we weight the values of each QoS equally.

5.3.1 Comparison with Conventional Approach

We compare our FASICA Framework with the following two approaches:

1. approach by Al-Masri, et al. [1], which takes the approach of finding the service with the best QoS service.
2. linear sequential search of four candidate services, checking each QoS value until satisfactory services are found.

Fig. 4 compares the total number of distinct target services that were found from the 500 requests and the computation time between our FASICA framework and the above two approaches. It shows that the total number of target services selected by our FASICA Framework is higher than the total number using Al-Masri's framework regardless of K and SDT values. This means that the service requests are distributed to multiple services rather than all requests being sent to one "best" service, thus relieving the overload issue.

Next, for computation time, our FASICA Framework clearly takes less time compared to the two other approaches. The biggest reason for this is because in our framework, a previous solution can be found in the Answer Cache and be used as a solution for a request without wasting time to completely recompute. Moreover, our FASICA Framework creates a K-d tree once and uses this tree to prune the subtrees that will not contain the solution for searching. The overhead for updating the QoS or removing the QoS of an unused service is not too high. The time complexity to update a tree branch for updating QoS is O(log n). This can be done offline or during non-peak time. Therefore, we believe that our FASICA Framework is a solution that can be used for runtime selection.

5.3.2 Effect of Answer Cache

We discuss the benefit of the Answer Cache. Fig. 5 compares the computation time with and without the Answer

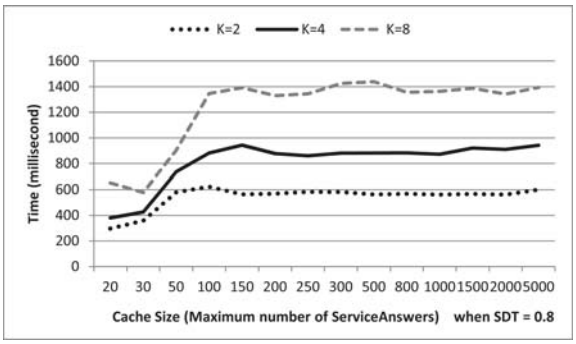

Figure 6: Computation time with different cache size

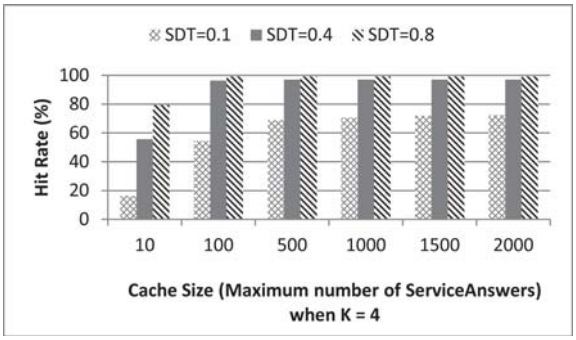

Figure 7: Cache hit rate when K=4

Cache when SDT = 0.8 and the cache size is 1000. The graph clearly shows that the Answer Cache can significantly lower the computation time.

Although the Answer Cache can lower the computation time, we need to consider the cache size. Fig. 6 shows that when we increase the cache size, the computation time will also increase up until the cache size becomes about 100. The increase can be said to be due to more Service Answers needing to be checked. Note that this does not mean that a small cache size is better, because a small cache size would mean that the possibility of an answer existing in the cache becomes lower, and the K-d tree would need to be checked more often.

When the cache size is greater than about 100, the computation time does not change much. This is likely due to the cache hit rate. As Fig. 7 shows, the cache hit rate reaches 100% when the cache size becomes 100, K=4, and SDT=0.8. At that point, since the answer is always contained in the cache, the computation time does not change much.

Note that since we generated requests that have similar constraints using SQPR Generator, the cache hit rate is quite high. Even if the constraints are not similar, i.e., the radius of the hypersphere is set to a value greater than 0.1, the hit rate will be affected by the value of SDT. When the value of SDT becomes large, the hit rate will also become higher. Thus, we still need a way to determine the proper cache size as well as SDT value, because different data will likely have different results. This is part of our future work.

6. CONCLUSION

In this paper, we proposed the FASICA framework that selects target services based on QoS constraints. We employ a cache to speed up the search. When there are no suitable service in the cache, K-NN algorithm is applied to a K-d tree to find candidate services. The actual target service is chosen based on a distribution policy. We evaluated our FASICA framework with a simulation, which showed that our framework uses less computation time than a conventional approach.

An important part of our future work includes how we can determine the cache size as well as the K and SDT values. We also intend to apply our approach to selecting a service in an actual dynamic setting, where for example if a service becomes unavailable, another service can be substituted in its place.

7. REFERENCES

[1] E. Al-Masri and Q. Mahmoud. Discovering the best web service. In *Proc. 16th Intl. Conf. on World Wide Web*, pages 1257–1258, 2007.

[2] E. Al-Masri and Q. Mahmoud. Investigating web services on the world wide web. In *Proc. 17th Intl. Conf. on World Wide Web*, pages 795–804, 2008.

[3] M. Alrifai and T. Risse. Combining globally optimization with local selection for efficient QoS-aware service composition. In *Proc. 18th Intl. Conf. on World Wide Web*, pages 881–890, 2009.

[4] M. Alrifai, D. Skoutas, and T. Risse. Selecting skyline services for QoS-based web service composition. In *Proc. 19th Intl. Conf. on World Wide Web*, pages 11–20, 2010.

[5] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Trans. on Software Engineering*, 33(6):369–384, 2007.

[6] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. 17th Intl. Conf. on Data Engineering*, pages 421–430, 2001.

[8] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, 13(1):21–27, 1967.

[9] D. Dyachuk and R. Deters. Scheduling of composite web services. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 19–20, 2006.

[10] GForge Collaborative Random Development Environment. Random dataset generator for SKYLINE operator evaluation project. `http://randdataset.projects.postgresql.org/`. accessed Aug. 31, 2014.

[11] A. Jongtaveesataporn and S. Takada. Enhancing enterprise service bus capability for load balancing. *WSEAS Trans. on Computers*, 9(3):299–308, 2010.

[12] G. Kang, J. Liu, M. Tang, X. Liu, and K. Fletcher. Web service selection for resolving conflicting service requests. In *Proc. 2011 Intl. Conf. on Web Services*, pages 387–394, 2011.

[13] Oracle. Oracle's Siebel business applications bookshelf documentation library, version 7.8. `http://docs.oracle.com/cd/B31104_02/books/FieldServ/FieldServServSupport44.html#wp1004398`. accessed Aug. 31, 2014.

[14] L. Pan, L. Chen, J. Hui, and J. Wu. QoS-based distributed service selection in large-scale web services. In *Proc. Intl. Conf. on Services Computing*, pages 725–726, 2011.

[15] S. Shahand, S. Turner, W. Cai, and M. Hedayat. DynaSched: a dynamic web service scheduling and deployment framework for data-intensive grid workflows. *Procedia Computer Science*, 1(1):593–602, 2010.

[16] Weka. Weka 3 data mining software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`. accessed Aug. 31, 2014.

[17] T. Yu and K. Lin. Service selection algorithms for web services with end-to-end QoS constraints. In *Proc. of the Intl. Conf on E-Commerce Technology*, pages 129–136, 2004.

[18] T. Yu, Y. Zhang, and K. Lin. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Trans. on the Web*, 1(1):1–26, 2007.

[19] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng. Quality driven web services composition. In *Proc. 12th Intl. Conf. on World Wide Web*, pages 411–421, 2003.

[20] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Trans. on Software Engineering*, 30(5):311–327, 2004.