Chapter 12
Improving the QoS of a Composite Web Service by Pruning its Weak Partners

Kuljit Kaur Chahal, Navinderjit Kaur Kahlon and Sukhleen Bindra Narang

Abstract Quality of Service (QoS)-aware web service composition is based on nonfunctional properties of component (or partner) web services. In a dynamic environment, these properties of partner web services change on the fly. There exist several research proposals that take into account QoS degradation of partner web services at run-time, and propose solutions to maintain the optimality of the service composition in such circumstances. In this paper, we focus on the problem from a different perspective. We take into account the situation when quality (QoS values) of some of the partner web services improves, but for some others it remains the same. With the passage of time, if the quality of these web services does not improve, they act as bottlenecks or the weakest links in an otherwise efficient process. We simulate a framework which identifies such web services, and expands the search domain by sending a selective query to remote/premium service registries/brokers for finding better alternatives of such services. The proposed approach is effective, efficient, and scalable as well.

Keywords Service computing · SOA · Service composition · Web services · Supply-Chain network · Quality of Service · Weakest link

12.1 Introduction

In the Service-Oriented Architecture, a Composite Web Service (CWS) is created to serve functionality which the existing web services are not able to provide. In addition to the functionality, a CWS is also supposed to fulfill the expected non-functional requirements of its end users. QoS attributes of a web service describe its

K.K. Chahal (✉) · N.K. Kahlon
Department of Computer Science, Guru Nanak Dev University, Amritsar, India
e-mail: kuljitchahal@yahoo.com

S.B. Narang
Department of Electronics Technology, Guru Nanak Dev University, Amritsar, India

M. Ramachandran and Z. Mahmood (eds.), *Requirements Engineering for Service and Cloud Computing*, DOI 10.1007/978-3-319-51310-2_12

nonfunctional properties. It encompasses a number of performance metrics of a web service such as availability, reputation, price, execution time, response time, etc. A CWS depends upon its partner web services to fulfill functional as well as nonfunctional requirements of its end users. Successful execution of partner web services contributes to meet user's expected QoS of the CWS. Inability to do so may lead to loss of current as well as future business of the CWS, though the cause of the deficient service lies outside the ambit of the CWS provider.

Several methods have been proposed in the research literature to accurately estimate aggregate QoS value of a CWS given the QoS values of its partner (component) web services [4]. A web composition can be static in which partner web services are decided at design-time [8]. However, such a composition is not suitable for a dynamic environment in which partner web services are controlled by third parties and may become unavailable or their QoS values degrade during run-time [1]. We need solutions which are dynamic as well as proactive to manage QoS degradation of partner web services in such a way that aggregate quality level of the CWS can be maintained.

We conjecture that a suboptimal (QoS aware) solution may not always be due to QoS degradation of some of the partner web services of a CWS. A solution may also become suboptimal when quality of most of the services improves barring a few. In such a situation, there is need to look into the web services whose QoS value is worst and stable (i.e., neither degrades nor improves).

Performance (in terms of QoS) of a CWS is dependent on the performance of its partner web services (in combination). We define weakest link of a CWS as a partner web service that limits the web service in attaining higher efficiency beyond a certain threshold. Therefore, identification and penalization (i.e., substitution) of weakest links becomes imperative to improve performance of the CWS. The idea of identifying a weakest link in a CWS (when comprehended as a value chain) is akin to identification of bottlenecks in a supply-chain network.

We propose to use log analysis of the previous CWS execution traces to identify a weakest link in its execution. A weakest link, in a CWS execution process, is a web service whose QoS value contributes the maximum (or minimum) to the global (aggregate) QoS value of the process to deviate it from attaining a better value. For example, a set of partner web services of a CWS has the following values for the Execution Time (ET) QoS attribute (in ms): $\{w_1 = 0.3, w_2 = 0.4, w_3 = 0.6, w_4 = 0.7, w_5 = 2.5\}$. If ET for the CWS is calculated using the aggregation formula for a serial workflow (i.e., sum of all the values), it is 4.5 ms. In this aggregate value, maximum contribution is of the web service W_5. It has the maximum value for the ET, and the value is significantly different from the corresponding values of the other web services. Now W_5's ET value may be high due to the nature of the task it performs. For a complex task, ET will be high. If nature of the task is not complex, then high ET indicates low quality level of the web service in comparison to other web services in the value chain. Then, such a web service is identified as the weakest link.

We suggest that search space can be expanded on demand. When all other services in a web service composition improve, we should be able to identify a web

service which proves to be a weakest link in the configuration. Such web services can be replaced with alternative web services by exploring remote and/or premium repositories to look for alternatives.

The remainder of the paper is organized as follows: Next section defines the problem with the help of a motivating example. Section 12.3 presents the related work. Section 12.4 explains the research methodology. Section 12.5 shows evaluation of the proposed framework. Section 12.6 mentions limitations of the study. Last section concludes the paper followed by the references.

12.2 Problem Definition

Consider a situation in which execution time of some of the partner web services of a CWS improves over a period of time. There is a partner web service, for example, whose ET neither improves nor degrades. Had its ET degraded, it would have got substituted with a better alternative. Therefore, it continues to be a part of the configuration. With the passage of time, its ET is significantly different from the execution times of other web services in the CWS. As no degradation happens in its execution time over the period of time, it cannot be identified by a framework that may have been used to manage QoS degradation of partner web services [16]. There is need to identify this bottleneck service and find alternatives of this service (may be from global/premium service repositories). However, to the best of our knowledge, there does not exist any proposal that tackles the issues in a dynamic environment of a service oriented solution from this point of view

In an ideal situation, the best web service for a task, wherever it exists on the planet should be searched and included in the composition. But reality is very different from this. A number of service registries exist. It is not possible for a service discovery module to look into all the possible service registries in real time to get the best services in a time-efficient manner when a composite web service is configured from scratch. There is a tradeoff between finding the best services and time complexity of a service discovery process [2, 7]. Moreover, some registries are public, and some are private and available at a premium. Therefore, for time and cost efficiency, a service discovery module should expand the search to premium registries only in exigent cases. Previous work in the service discovery domain [6] acknowledges the need for creating registry federations to carry out the discovery process in multiple registries [3]. Such an approach gives more useful results than a centralized repository. Among the numerous efforts to improve the efficiency of the service discovery module in SOA, Sivashanmugam et al. [17] propose a crawler engine that searches the web services information from multiple registries and other heterogeneous resources and creates a centralized large database called Web Service Storage (WSS). The large data set needs to be updated regularly by the crawler as changes occur in the services in their base locations. Changes in the web services are very frequent in the dynamic Internet based environment. Hence, such

solutions are inefficient. As they not only require high maintenance to reflect latest updates, but also suffer from single point of failure syndrome, and are not scalable as well. In distributed or decentralized approaches of web service registry management, a locally maintained registry is searched first. If the search request is not satisfied by the local registry, then it is sent to global registries. We also propose to involve global registries selectively as retrieval of web service information from global repositories is not cost effective for routine searching in the beginning of the web service composition process [2]. It takes time in tens of seconds [19]. We suggest that search space can be expanded on demand during the web service life cycle. When all other services in a web service composition improve, we should be able to identify a web service which proves to be a weakest link in the configuration. Then explore remote and/or premium repositories to look for an alternative.

We propose, in this paper, to start with a CWS (may be created with local optimization approach criteria alone or along with a global optimization approach) and then improve the solution in the dynamic environment. It is not (economically) feasible to create a CWS with global optimization criteria for every user request especially when the CWS is a long-term solution, and is invoked frequently. Once a CWS is created, it should be self-optimizing after that. We propose a framework which can identify partner web services, not in sync with other web services in the execution plan of the CWS. Such web services can be replaced with alternative web services. The premise of our proposal is that a discovery algorithm is able to search a subset of service repositories for web services required for a CWS. Searching in all the possible (under the sky) service repositories is not cost effective. A CWS may be created with whatever is available at the first instance in a local repository. It can be later improved by expanding the search scope to find alternatives for a few web services which are not in sync with others in the combination. This expanded search scope may encompass service repositories which are remote or are available at a premium.

Next, we analyze the weakest link approach for managing QoS of a CWS with a long life span. There exist a few web service composition approaches that consider life span of a CWS as well to decide the composition process to follow [12, 15]. A CWS with a short life span responds to a few requests for a short period of time. As soon as the business goal is fulfilled, the CWS ceases to exit. For every request, the CWS is created from scratch. In case of a CWS with long life span, configuration of the CWS is created once and then used many times. Jiang et al. [12] distinguish between two composition approaches as one time query, and continuous query. One time query corresponds to a CWS with short span, and is created from scratch for a new request. In continuous query, an old instance of a CWS is (re)used with some adaptation (if required) for the new requests. Liu et al. [15] define a long-term composed service (LCS) as a web service with long-term business goal (or an open-ended life time). It has a stable relationship with its partner web services to serve a continuous stream of requests. In such a situation, creating a CWS from scratch for every request is not right from efficiency point of view.

Table 12.1 Example of a typical scenario

Trace	Service bindings	Remarks
1	WS = {WS_{11}, WS_{21}, WS_{34}, WS_{42}, WS_{53}, WS_{61}} ET = {1000, 1200, 1300, 2000, 1300, 1400}	WS_{42} has the maximum execution time in the set
2	WS = {WS_{11}, WS_{21}, WS_{34}, WS_{42}, WS_{53}, WS_{63}} ET = {1000, 1200, 1300, 2000, 1300, 1200}	WS_{63} replaces WS_{61} as WS_{61}'s ET increases
3	WS = {WS_{11}, WS_{21}, WS_{34}, WS_{42}, WS_{53}, WS_{63}} ET = {1000, 1200, 1300, 2000, 1300, 1200}	=no change=
4	WS = {WS_{11}, WS_{25}, WS_{34}, WS_{42}, WS_{53}, WS_{61}} ET = {1000, 1100, 1300, 2000, 1300, 1200}	WS_{25} replaces WS_{21} as WS_{21}'s ET increases

12.2.1 Motivating Example

An analysis of previous traces of the CWS on client side (see Table 12.1), shows that web service WS_{42} is spending maximum time in execution (measured in ms), and thus delays the overall process. This service is also not being replaced (in the following traces) as its ET remains the same (i.e., does not degrade). We assume that a better alternative of the web service is not available in the primary set of registries that the service discovery module explores while searching for new services. So there is need to extend the search boundary to include a distant repository (may be of a premium category).

Our proposed work contributes in the following ways:

- We define a strategy to identify those web services in a value chain which contribute in making the CWS sluggish. A web service is defined as a weakest link if its QoS values are worst and stable for a long period of time.
- We make a case for the service discovery module to look for candidate web services in premium/remote service repositories only in exigent cases. The search space should expand as per demand, only if a better alternative is not available in the local repository.
- The proposed framework is applied on two types of CWS—one with short term use, and second with long term use.

12.3 Related Work

Major issue in the web service composition process has been to find a QoS-aware optimal solution for the service composition problem. Researchers proposed several methods to search for an optimal solution such as exact algorithms [5], heuristic algorithms (e.g., [14], and meta-Heuristic algorithms [20] using local optimization

or global optimization as the criteria. These solutions look for an optimal or near optimal solution by focusing on two perspectives: reducing time complexity of the algorithm, and limiting the search space. Most of the solutions are applicable in static environments only. However, web services-based solutions are realized using the Internet. The Internet being a dynamic entity, a web service composition configuration should remain optimal in the dynamic environment otherwise the optimal solution is limited to a few instances only. As soon as the environment changes, the solution goes below the optimum level.

Keeping the dynamism of the operating environment in mind, researchers in the past have focused on handling QoS degradation of web services to maintain the optimality of the solution in a dynamic environment. Research in this area has handled QoS degradation of partner web services, and proposed solutions to adapt the configuration of a CWS by replacing the degraded partner web service with a better alternative [16].

We also work toward realizing an optimal solution for the web service composition problem. But we look at this problem from a different point of view. We monitor not only degradation of the (partner) web services, but also the web services which do not degrade themselves but become bottlenecks when all other services in a configuration improve in their QoS values. The questions that we aspire to answer are: what if (rather than degrading) some of the partner web services improve in their quality except a few of them? Would not it lead to the web services (whose quality does not improve) acting as bottlenecks or weakest links in the value chain? As better alternatives for these services are not available in the registries explored by the service-oriented application in routine, solution lies in expanding the search space to include more repositories. We observe that this improves the solution quality in a time-efficient manner as search in this case is for selective web services (a subset of the total set of services) only.

In a dynamic environment, managing QoS degradation of the partner web services requires querying the service resources for latest information about their QoS values. Keeping in mind the running time overhead that is incurred for collecting this information, Harney and Doshi [9] propose to use a selective query approach. In this, full information is not requested from the resource providers for all the services at one time. Only those services whose QoS values may have changed are queried for the information. We also use selective querying, but in a different context. They use selective querying, from monitoring point of view, to collect latest information at the execution time. We too use this approach during execution phase of the CWS but from discovery point of view. Their target of the query is a service registry/broker that is already supporting the application. We target the query at a service registry/broker which has not been yet explored by the application for the service discovery task.

Finding the weakest link in a supply-chain network is of interest for every business. But we could not find any solution from this domain that can be adopted for finding the weakest link in a service network in the context of SOA for

managing the solution in an automated way. We chose a statistical approach to identify the web service which contributes maximum to the aggregate QoS value of the CWS. For the sake of simplicity, we have assumed only one dimension of the QoS value, e.g., execution time. Execution time is a QoS attribute with a negative dimension, i.e., lower is the value, better is the quality. Multiple attributes can also be incorporated easily in this approach by using approaches like Simple Additive Weighting to find a utility score for a web service.

During exploration of the related research literature, we could find only one other research paper which focuses on the same issue of finding a weakest link in a CWS configuration. Research work in [10] focuses on the weak points in a QoS composition to improve it. The approach to identify weak point is similar to ours—a service with biggest impact on the composition with respect to a QoS attribute. Unlike our statistical approach, the authors suggest two approaches for doing so—in brute force method all the services are tried one by one to identify the web service with the biggest impact, and in branch and bound method a branch (in a parallel workflow pattern) with the highest execution time is followed to identify the weak point web service. Unlike our framework, they assume that a better alternative of the weak point web service does not exist in the service repository. Therefore, the solution to replace the weak web service with a set of alternatives is limited to the available web services in the local repository. The solution is then another combination of existing web services realized after analyzing various arrangements for different workflow composition patterns.

We premise that such a solution is more relevant for a CWS with an open-ended life time. A few solutions already exist which focus on managing partner web services of a CWS created for long-term use [12, 15].

Jiang et al. [12] perceive the requirement of a mechanism to support execution of a CWS which responds to a continuous stream of requests. For a one time composition request, a CWS is created from scratch, and it ceases to exist as soon as the response is generated. However, when there is continuous flow of requests for a CWS on the service network, an old CWS instance can be (re)used to respond to the requests. In case of dynamic changes in the partner services of the CWS, only affected services are replaced and not the whole web service space. The continuous query-based approach has good scalability, and is more efficient than creating a CWS from scratch for every request.

Liu et al. [15] propose a solution to manage changes that pertain to top level view of a LCS (Long Composed composite web Service). For example, owners of the LCS may have different functionality (due to business changes) or QoS (due to new competitors in the market) requirements. Therefore, changes are introduced from the top. As requirements change, web services may be added to or removed from the LCS configuration. Unlike them, we manage changes in a LCS from bottom to top. We detect web services whose QoS values are stable but worst in the configuration, then change the LCS configuration by pruning such web services, and replace them with better alternatives.

12.4 Research Methodology

We propose to analyze execution trace of a CWS periodically. Duration of periodicity can be determined by the service owner on the basis of cost/benefit tradeoff of executing the analysis. The execution trace records every partner web service's QoS value (e.g. Execution Time). Web services advertise their processing time or provide methods to inquire about it. Kahlon et al. [13] propose publish-subscribe mechanism based solution to provide web service QoS values to its clients.

12.4.1 The Statistics

This study explores analysis of the extreme values in an execution trace using Interquartile Ranges and Tukey Fences [18] as the statistics. Interquartile range is the statistic to measure variability in a data set. It is the difference between the first Quartile, Q_1, and the third Quartile, Q_3. It gives the range of the middle 50% values in a data set. The formula to calculate is

$$\text{InterQuartile Range (IQR)} = Q_3 - Q_1 \tag{12.1}$$

The Quartiles Q_1, and Q_3 represent respectively the least 25%, and the largest 25% of the values of a data set.

Tukey Fences is a popular method of identifying extreme values in a data set. After calculating the first and third Quartiles for a data set, the Tukey Fences are calculated as follows:

$$\text{Lower limit} = Q_1 - 1.5\,(\text{IQR}) \tag{12.2}$$

$$\text{Upper limit} = Q_3 + 1.5\,(\text{IQR}) \tag{12.3}$$

12.4.2 Identifying the Outlier(s)

In the present case, the data set consists of QoS values of partner web services of a composite web service. A QoS attribute can have a positive or a negative dimension.

For identifying an outlier in the case of a negative QoS attribute, first find the maximum value in the data set. If the maximum value is greater than the upper limit (defined in Eq. 12.3), then the corresponding data item is an extreme value in the data set. Similarly for finding an outlier in the case of a positive attribute, if the minimum value is lower than the lower limit (Eq. 12.2), then that is the extreme value in the data set. For a given data set regarding ET of 7 partner web services that constitute a CWS, let us examine the statistics in Table 12.2.

Table 12.2 Statistics for an example data set

Data set	Median	Q_1	Q_3	IQR	Upper limit
{70, 200, 400, 560, 756, 832, 2200}	560	300	794	494	1301

As per the given values, the maximum value in the data set (i.e., 2200) is greater than the upper limit of the Tukey Fences (i.e., 1301). Therefore, it is an extreme value. We know that a better web service is not available in the service repositories being explored by the service discovery module in the normal routine (otherwise this web service would have got replaced already). There is need to expand the search boundary to find a better web service.

12.4.3 Analyze the Influence of the Outlier

A workflow in a service composition may follow serial, cyclic, or parallel, or a combination of the three execution patterns of partner web services. Aggregate value of a QoS attribute for a CWS is calculated using different formulae for the different workflow patterns [11]. In this paper, we consider that service composition follows a serial workflow pattern. In a serial pattern, the partner web services execute one after another. Output of one web service becomes input of another in a serial order. Aggregate value of the Execution Time QoS for a CWS is the sum of Execution Times of all its partner web services. In our context, the model is an aggregate function Sum, it takes ET values of various web services and gives ET of the CWS as output. We assume the workflow pattern as a simple sequence of service executions. Here, the aggregate (global) value for the QoS attribute ET is sum of the ET value of each partner service.

The influence of a data point on the aggregate is calculated by first finding the difference between the original aggregate (which included the said data point) and the modified aggregate (excluding the said data point). The influence is defined as a ratio between this difference and the number of data points contributing to this change. When there is more than 1 outlier, influence values can be used to order the pruning actions. An outlier with maximum influence is pruned first.

12.5 Results and Analysis

The weakest link analysis approach is analyzed for a CWS with short span of life (in Experiment 1), as well as for a CWS with long span of life (in Experiment 2). A CWS, with short life, is created from scratch for every new request. Whereas a CWS, with long life, is (re)used to respond to forthcoming requests. We use a synthetic data set in the experiment. Values for the QoS attribute ET are generated using a uniform random process. The solution is implemented in C++ using

CodeBlocks 11.0 IDE with gcc as the compiler on an Intel machine with Core 2 Duo CPU, 2 GB RAM, and Windows XP as the operating system.

12.5.1 Experiment 1

Here, we consider a simple situation in which a CWS is created from scratch for every request. A new configuration for the CWS is created by searching the local/global repositories and then executed. Our proposed solution is to analyze execution trace of a CWS to identify the partner web services that contributed the maximum in QoS (parameters with negative dimension such as Execution Time) of that instance of the CWS.

This section presents the evaluation of the framework by comparing it with other two naïve approaches. We use local optimization as the criteria for selecting web services from the candidate set of services.

We create three different cases to analyze the results of the proposed approach. First two cases model two different base (benchmark) situations. In the first case, service discovery is limited to a local repository. By using an exhaustive strategy, best service for each task is selected from the candidate web services in the local repository. In the second case, service discovery is expanded to a global repository, and selection strategy is exhaustive again. In the third case (the proposed approach), service composition is created using candidate web services from the local repository, and then service discovery is expanded to global repository only when the need is felt to manage web services with worst QoS value in the configuration.

In order to compare the cases, we measure

- Efficiency, i.e., the time taken to generate the CWS configuration,
- Quality of the solution in terms of aggregate QoS value of the resultant CWS.

Before we compare the proposed approach with the two basic approaches, we discuss the effectiveness (i.e., the usefulness) of the proposed approach in the next paragraph.

12.6 Effectiveness

Figure 12.1 shows results of the simulation in which ET of all the partner web services improves except one service. ET is increased at different rates at 10% in the second run, at 25% in the third run. Then the web service which does not see any improvement in its ET QoS attribute in the first two runs is identified by the framework as the weakest link. When it is substituted with an alternate web service (with 25% better ET) from a distant repository in the fourth run, aggregate ET of the CWS improves by 33%. It improves only by 5 and 8% in the earlier two runs. It shows that the proposed approach is promising.

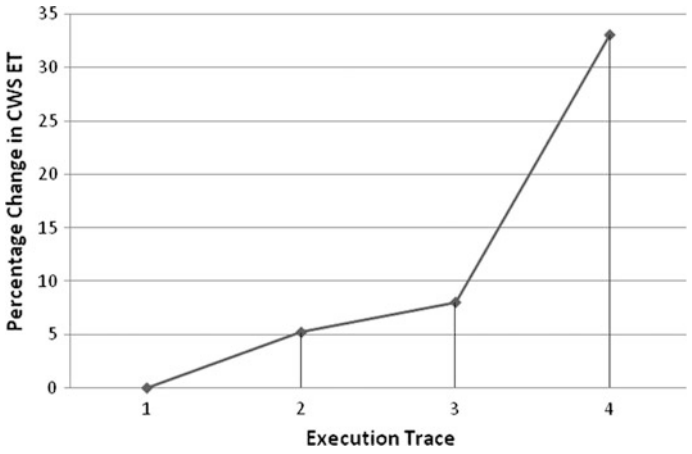

Fig. 12.1 Improvement in CWS ET after pruning and substituting its weakest link WS

12.7 Efficiency

We simulated one service repository at the local machine, and one global repository on a different machine. Network latency value of 101 ms between the two sites was taken on the basis of the monitoring information available on the Dotcom-Monitor cloud network (https://www.dotcom-tools.com/internet-backbone-latency.aspx) on June 10, 2016. The Dotcom-monitor provides standard baseline network latency between different locations that it monitors across the globe. We selected Mumbai (India), and Hongkong (China) as the two locations for simulating the process. Mumbai has the least network latency with Hongkong.

Both the repositories were populated with web services with similar functionalities. A few web services with better QoS values (than the local repositories) were made available in the distant repository only.

For the first case, only the local repository was used in the discovery process. Local optimization was used to create the initial composition configuration. Here, the running time increases at a polynomial rate of growth when number of tasks is five (Fig. 12.2). The best fit equation in this case is $y = 13.45x^2 - 133.8x + 345.6$ with $R^2 = 0.833$ for five tasks. However, as the situation becomes more complex with a higher number of tasks, running time starts following an exponential growth rate (Fig. 12.3). Here, the best fit equation is found to be $y = 41.10e^{0.251x}$ with $R^2 = 0.740$.

In the second case, the local as well the global repository was searched during the service discovery process. This approach is very poor in scalability as the running time curve follows an exponential rise as the number of candidate web services in the registry increases. The best fit equation is $y = 0.347e^{0.596x}$ for 5 tasks with coefficient of determination $R^2 = 0.975$. Similarly, $y = 0.289e^{0.685x}$ for 10 tasks

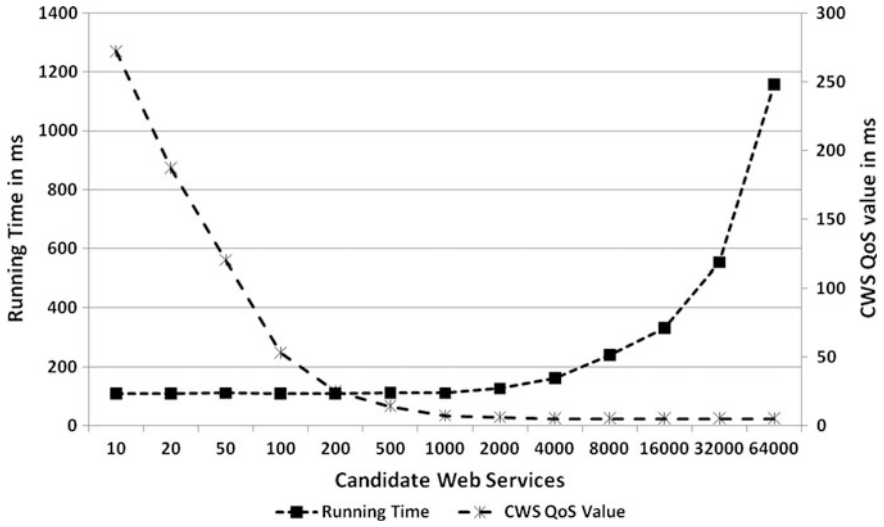

Fig. 12.2 Using local repository with local optimization for number of tasks = 5

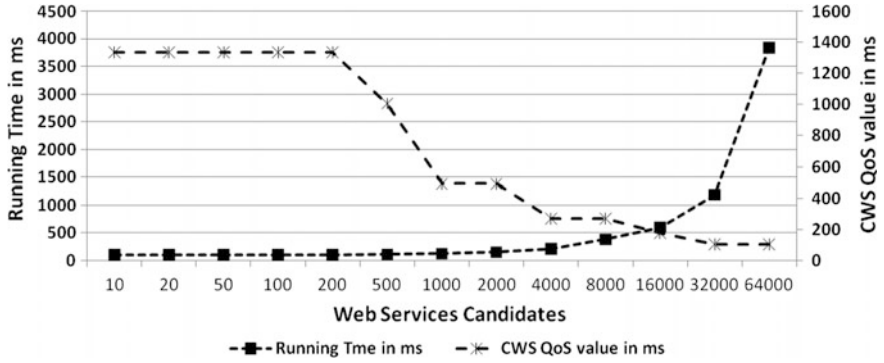

Fig. 12.3 Using local repository with local optimization for number of tasks = 10

with R^2 = 0.985. It happens in both the cases: when number of tasks is five (see Fig. 12.4), or is increased to ten (see Fig. 12.5).

The third case corresponds to the proposed work in this paper. A configuration was analyzed to identify the weakest link in the service sequence, and the global repository was searched only when there was a weakest link to find an alternative web service for the weakest link web service only. In this case, running time follows a polynomial rate of growth represented by the equation $y = 12.71x^2 - 126.7x + 333.1$ with $R^2 = 0.852$ for 5 tasks (Fig. 12.6). When number of tasks was increased to ten, even then the running time followed a polynomial growth rate (Fig. 12.7) with the best fit equation as $y = 31.42x^2 - 315.4x + 673.1$, and coefficient of

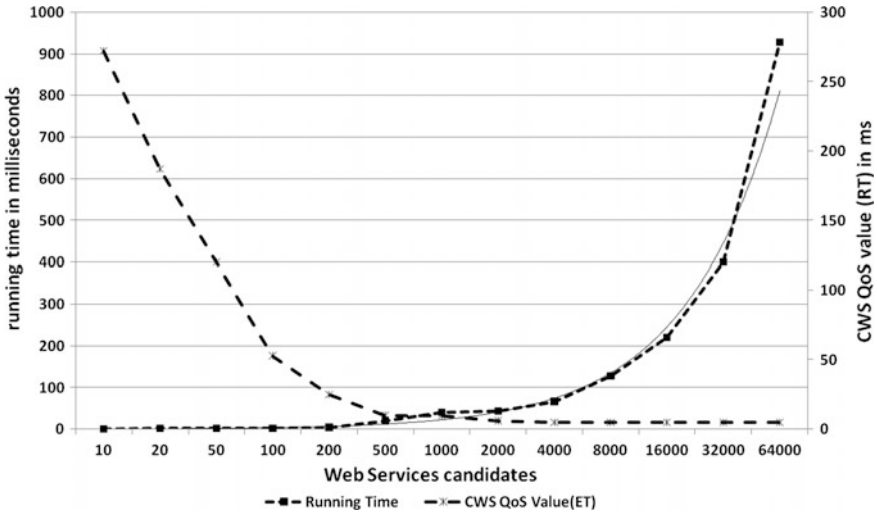

Fig. 12.4 Using local as well as global repository with 5 tasks

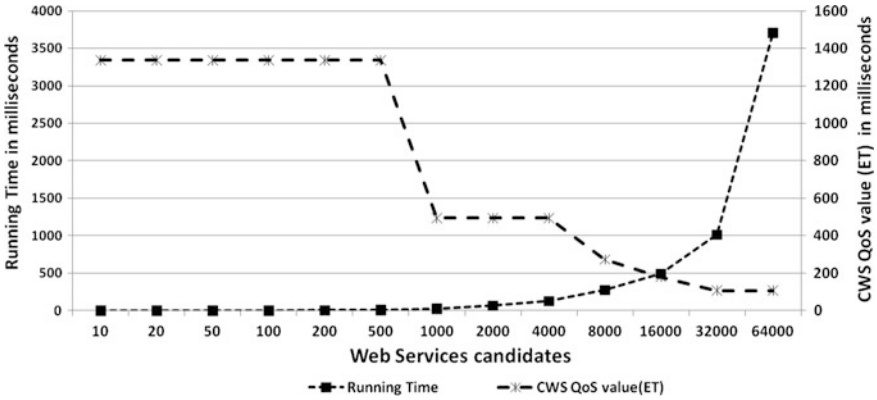

Fig. 12.5 Using local as well as global repository with 10 tasks

determination $R^2 = 0.801$. We analyzed the results for increasing the number of tasks to 20 (Fig. 12.8). The running time is still with polynomial growth rate represented by $y = 54.11x^2 - 533.9x + 1049$ with $R^2 = 0.867$ as the best fit equation.

12.8 Quality of the Solution

We measure quality of solution in terms of the aggregate QoS value for the Execution Time of the CWS. It can be observed (see CWS QoS value in Figs. 12.2, 12.3, 12.4, 12.5, 12.6, 12.7 and 12.8) that quality of the solution improves as the

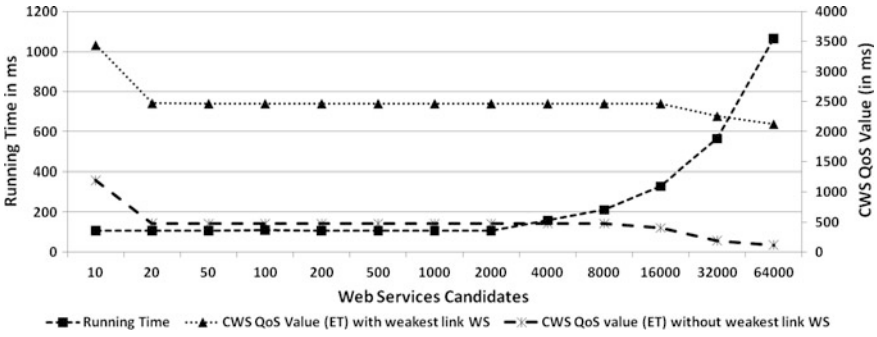

Fig. 12.6 Results for the proposed solution using local as well as a global repository with 5 tasks

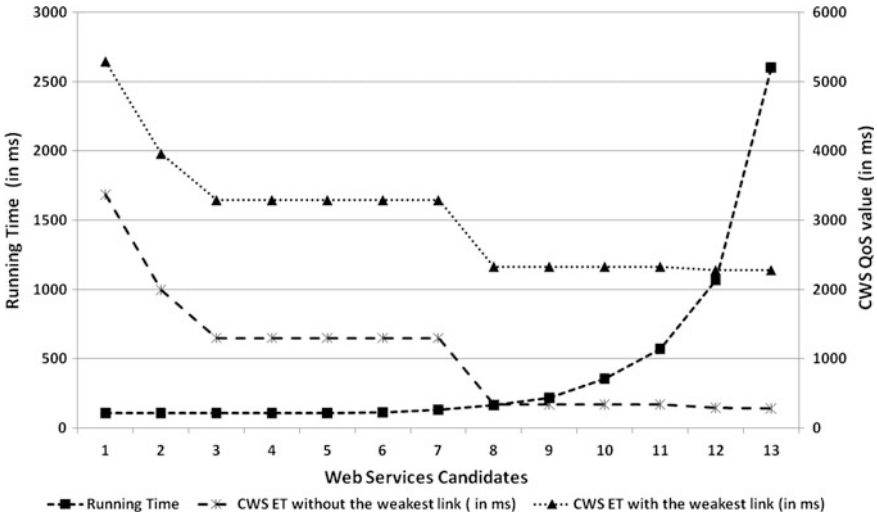

Fig. 12.7 Results for the proposed solution using local as well as a global repository with 10 tasks

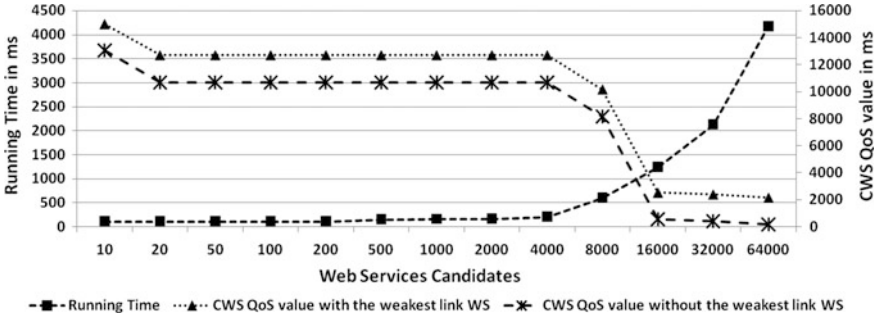

Fig. 12.8 Results for the proposed solution using local as well as a global repository with 20 tasks

search space is expanded in all the cases. Aggregate value of the Execution Time of a CWS decreases as more and more number of services are added to the search domain. As the number of tasks in a CWS increases, the aggregate ET values also increases, and intuition also implies the same. However, the cost of improvement is the least in case of the proposed solution.

12.8.1 Experiment 2

Analysis of the proposed framework for a CWS with long span of life is presented in this section. This section presents the evaluation of the framework by comparing it with a solution that does not use any policy to analyze a CWS execution plan to identify partner web services posing as weakest links in the value chain. In the first case, service discovery is limited to a local repository. By using an exhaustive strategy, best service for each task is selected from the candidate web services in the local repository. In the second case (or for the proposed approach), service composition is created using candidate web services from the local repository, and then service discovery is expanded to global repository only when the need is felt to manage web services with worst QoS value in the configuration.

In order to compare the approaches, we measure

- Quality of the solution in terms of aggregate QoS value of the CWS.
- Efficiency, i.e., the time taken to generate the CWS configuration
- Scalability, i.e., the response of the proposed approach as the problem size scales up.

We use a synthetic data set in the experiment. Values for the QoS attribute ET are generated using a uniform random process. The solution is implemented in C++ using CodeBlocks 11.0 IDE with gcc as the compiler on an Intel machine with Core 2 Duo CPU, 2 GB RAM, and Windows XP as the operating system.

12.9 Quality of the Solution

We measure quality of solution in terms of the aggregate QoS value for the Execution Time of the CWS. In the first case, only the local repository was used in the discovery process. Local optimization was used to create the initial composition configuration. In a static environment, a CWS is created only once, and responds to all the requests that it gets after that. Figure 12.9a shows the CWS QoS value for first and the subsequent requests in case of static composition. It stays almost the same for the 20 requests the CWS was run for. When the proposed framework is used to analyze the CWS execution process for weakest link web services in a static environment, CWS QoS value improves (for request number 2 in the Fig. 12.9).

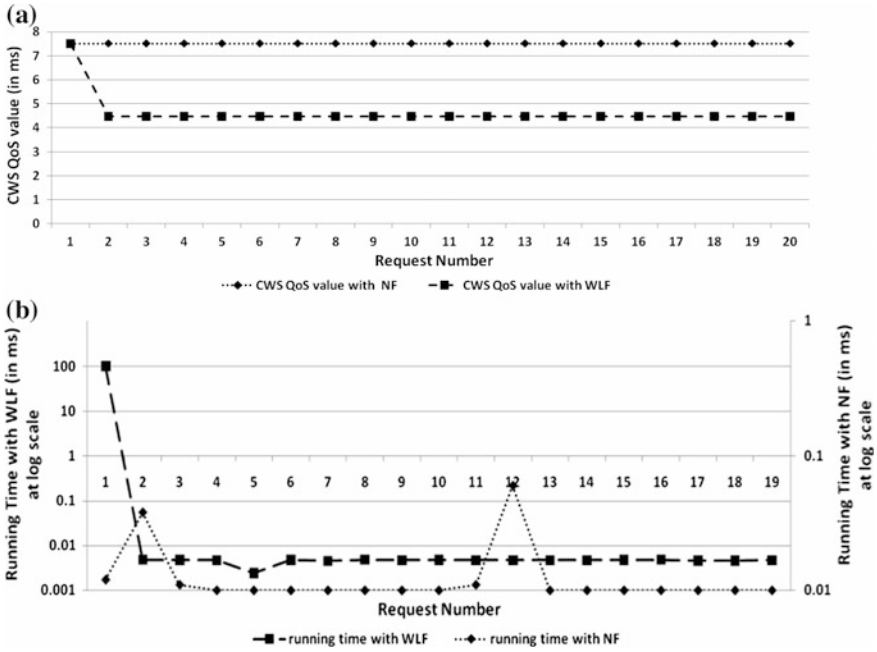

Fig. 12.9 Experimental results for a static environment

We use NF for No Framework, and WLF for the Weakest Link Framework proposed in this paper.

For the proposed solution, we simulated one service repository at the local machine, and one global repository on a different machine. Network latency value of 101 ms between the two sites was taken on the basis of the monitoring information available on the Dotcom-Monitor cloud network (https://www.dotcom-tools.com/internet-backbone-latency.aspx) on June 10, 2016.

In the second case, we considered a dynamic environment in which QoS values of the partner web services change (improve) randomly. The experiment results (in Fig. 12.10a) show that CWS QoS values improve consistently in both the cases (without as well as with the framework). However, improvement in case of the Weakest Link Framework (WLF) is far better than the case when no framework is used.

12.10 Efficiency

Figures 12.9b and 12.10b present the running time of composing a CWS in static and dynamic environments respectively. The running time of the naïve approach (no framework) is better than the proposed approach in the static environment. With

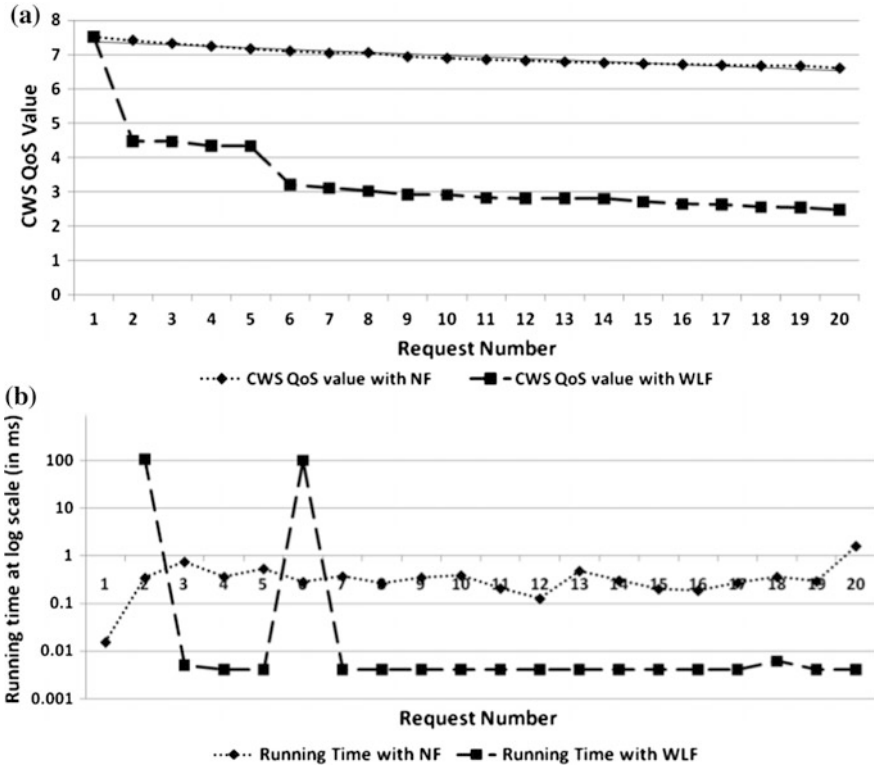

Fig. 12.10 Experimental results for a dynamic environment

the framework, running time is considerably high for the first request. But with support for the weakest link analysis and replacement with a better alternative, the running time decreases significantly for the subsequent requests.

In the dynamic environment, as QoS of the partner web services improve a few web services become weakest links. We can see spikes in the running time for the proposed framework. Otherwise, the running time for the proposed framework is better than the naïve approach.

12.11 Scalability

Figure 12.11a, b gives a comparison of the average CWS QoS value, and average running time for both the approaches. When we scale up the number of requests that invoke the CWS, the average CWS QoS value improves in case the proposed framework is employed. However, it remains almost at the same level throughout for the naïve approach. The average running time is also better (than the naïve

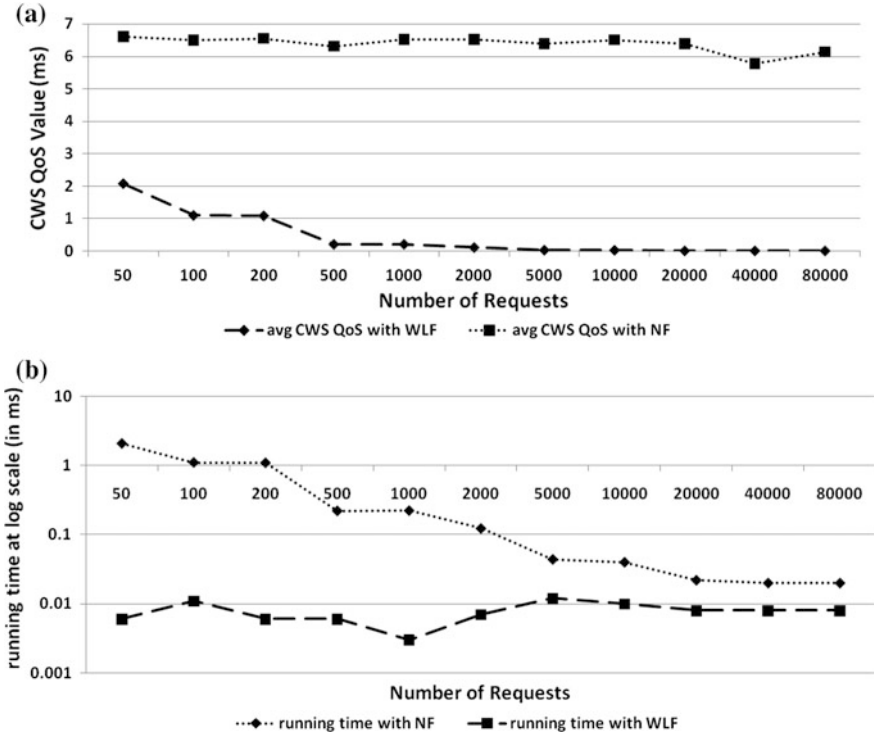

Fig. 12.11 Experiment results to show scalability of the approach

approach) for the proposed framework as the number of requests scale up from 50 to 80,000. In this case running time is almost constant. Actually, extra overhead to deal with the weakest link web services gets distributed in multiple requests.

12.12 Limitations of the Study

Three factors determine optimization of a web service composition problem: number of tasks of the CWS, number of candidate web services for the tasks, and number of QoS factors to watch for optimization. This study focuses on only the first two. For the third one, we assumed a simple QoS model with only one dimension.

The proposed approach follows local optimization as the evaluation criteria for service selection. It does not consider global constraints on the solution. Though improvements in global QoS value are appreciated and given preference.

12.13 Conclusions

This paper proposes an approach to improve QoS of a composite web service when some of its partner web services become weakest links in the workflow. The weakest links are identified, and then pruned from the configuration of the composite web service. Alternatives of such web services do not exist in the service registry that the service discovery module explores in routine. Therefore, the search space is expanded to bring some distant/premium service repositories/brokers in the ambit of the service discovery module. Simulation results show that the proposed approach is effective and efficient as well. In the present case, a web service composition configuration is created from scratch for every request. Such an approach is not efficient when requests for the same CWS are pouring at a continuous rate (called a long term composed service). In the second experiment, CWS QoS analysis in static as well as dynamic environments shows that the proposed framework gives better quality of the solution. At the same time, running time (computation cost) of the proposed solution is better. Scalability of the proposed framework is tested for running it for 50 requests to 80,000 requests. Its running time is stable as the number of requests scales up. At present, we are working on a prototype to implement the proposed solution in a real-world application.

References

1. Alamri, A. et al. (2006). Classification of the State-of-the-Art Dynamic Web Services Composition Techniques, International Journal of Web and Grid Services, vol. 2, pp. 148–166, Sept. 2006.
2. Al-Masri, E., Mahmoud, Q. (2007). Crawling Multiple UDDI Business Registries, WWW 2007(poster paper), May 8–12, 2007, Banff, Alberta, Canada, pp. 1255–1256.
3. Baresi, L., Miraz, M. (2006). A Distributed Approach for the Federation of Heterogeneous Registries, (Eds.) A. Dan, W. Lamersdorf Proceedings 4th International Conference Service-Oriented Computing ICSOC 2006:, Chicago, IL, USA, December 4–7, 2006. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 240-251.
4. Cardoso, J. et al. (2004). Quality of Service for Workflows and Web Service Processes, Journal of Web Semantics, vol. 1, pp. 281–308.
5. Chen, M., Yan, Y. (2014). QoS-aware service composition over graphplan through graph reachability, in Proceedings of the 2014 IEEE International Conference on Services Computing, pp. 544–551.
6. Crasso, M., Zunino, A., Campo, M. (2011). A Survey of Approaches to Web Service Discovery in Service-Oriented Architectures, J. Database Management. Vol 22, issue 1, pp. 102–132.
7. Deng, S., Wu, Z., Wu, J. (2012). An Efficient Service Discovery Method and its Application in (Eds.) Zhang, L. Innovations, Standards, and Practices of Web Services: Emerging Research Topics. Information Science Reference, IGI Global.
8. Dustdar, S., Schreiner, W. (2005). A Survey on Web Services Composition, International Journal on Web and Grid Services, vol. 1, pp. 1–30, Aug 2005.

9. Harney, J., Doshi, P. (2008). Selective querying for adapting web service compositions using the value of changed information, IEEE Transactions on Services Computing, 1 (3), pp. 169–185.
10. Jaeger, M., Ladner, H. (2006). A Model for the Aggregation of QoS in WS Compositions Involving Redundant Services, Journal of Digital Information Management, 2006, Digital Information Research Foundation.
11. Jaeger, M., Rojec-Goldmann, G., Muehl, G. (2004). QoS Aggregation for Web Service Composition using Workflow Patterns, Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC 2004), Monterey, California, USA, IEEE CS Press, pp. 149–159.
12. Jiang, W., Hu, S., Lee, D., Gong, S., Liu, Z. (2012). Continuous Query for QoS-Aware Automatic Service Composition. IEEE International Conference Web Services (ICWS), 2012.
13. Kahlon, N.K., Chahal, K. K., Kapoor, S.V., Narang, S.B. (2015). Managing Availability of Web Services in Service Oriented Systems, Proceedings of 2015 Asia-Pacific Software Engineering Conference (APSEC), New Delhi, pp. 316–321.
14. Li, J., Zhang, X., Chen, S., Song, W., Chen, D. (2014). An Efficient and Reliable Approach for QoS aware service composition, Information Sciences, vol. 269, pp. 238–254, June 2014.
15. Liu, X., Bouguettaya, A., Wu, X. and Zhou, Li. (2013). Ev-LCS: A System for the Evolution of Long-Term Composed Services. IEEE Trans. Serv. Comput. 6, 1 (January 2013), 102–115.
16. Ma, H., Bastani, F., Yen, I., Mei H. (2013). QoS-Driven Service Composition with Reconfigurable Services, IEEE Transactions on Services Computing, 6(1):20–34.
17. Sivashanmugam, K., Verma, K., Sheth, A. (2012). Discovery of Web Services in a Federated Registry Environment, Proceedings of the Second International Conference on Computer Science, Engineering and Applications (ICCSEA 2012), May 25–27, 2012, New Delhi, India, Volume 1.
18. Tukey, J. (1977). Exploratory Data Analysis, Addison-Wesley, 1977, pp. 43–44.
19. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J. and Chang, H. (2004). QoS-Aware Middleware for Web Services Composition, IEEE Transactions on Software Enggineering. 30(5): 311–327, 2004.
20. Zhou, X., Shen, J., Li, Y. (2013). Immune based chaotic artificial bee colony multiobjective optimization algorithm, in Proceedings of the 4th International conference on Swarm Intelligence, vol. 7928, pp. 387–395.