IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

MultiCuckoo: Multi-Cloud Service Composition using a Cuckoo-Inspired Algorithm for the Internet of Things Applications

HEBA KURDI[1,2], (Member, IEEE), FADWA EZZAT[1], LINA ALTOAIMY[3], (Member, IEEE), SYED HASSAN AHMED [4],(Senior Member, IEEE), AND KAMAL YOUCEF-TOUMI [2],(Senior Member, IEEE)

[1]Computer Science Department, King Saud University, Riyadh, Saudi Arabia (e-mail: HKurdi@ksu.edu.sa)
[2]Mechanical Engineering Department, Massachusetts Institute of Technology (MIT), Cambridge, MA 02139, USA
[3]Information Technology Department, King Saud University, Riyadh, Saudi Arabia
[4]Computer Science Department,Georgia Southern University, Statesboro, GA 30458, USA

Corresponding author: Heba Kurdi (e-mail: HKurdi@ksu.edu.sa).

This work was supported through King Saud University's Deanship of Scientific Research via research group no. RG-1438-002.

ABSTRACT Internet of things (IoT) applications aim to provide access to widespread interconnected networks of smart devices, services, and information. This can be achieved by integrating IoT and cloud computing (CC). By using cloud computing service composition (SC), multiple services from various providers can be combined to meet users' requirements. However, SC is known for its complexity and is classified as an NP-hard problem; such problems are usually approached using heuristics, such as bio-inspired algorithms. This paper aims at developing a bio-inspired algorithm that mimics the behavior of cuckoo birds (which examine the nests of other birds to find eggs similar to their own) to find a composite service that fulfills a user's request in a multi-cloud environment (MCE). Previous work on cuckoo-inspired algorithms has generally utilized metaheuristics, which try to fit a "good" solution to a general optimization problem. In contrast, we propose a problem-dependent heuristic that considers the SC problem and its particularities in MCE. The proposed algorithm, MultiCuckoo, was thoroughly evaluated based on a well-controlled experimental framework that benchmarks the performance of the new algorithm to other outstanding SC algorithms, including the all clouds combination (ACC) algorithm, base cloud combination (BCC) algorithm, and combinatorial optimization algorithm for multiple cloud service Composition (COM2). The results show that our algorithm is more efficient in terms of decreasing the number of examined services, the composed clouds, and the running time in comparison to the benchmark algorithms.

INDEX TERMS Cloud computing, Cuckoo-inspired Algorithm, Service Composition, Internet of things, IoT

I. INTRODUCTION

The internet of things (IoT) is an evolving technology that connects physical devices and allows them to share and exchange information about the surrounding physical world [1]. The use of a sensor network (SN) plays a major role in the success of IoT applications [2]. However, SNs have limited resources and capabilities, specifically in terms of power consumption, storage capacity, and bandwidth; therefore, effective SNs require some type of efficient communication mechanism to overcome these constraints [3]. In addition, the vast amount of data generated by devices and sensing activities must be effectively collected, analyzed, and stored [4]. This can be achieved by utilizing cloud computing (CC) [5].

Providing ubiquitous services is the ultimate goal of any IoT application [1]. For instance, end users should have access to an unlimited number of services and information that is specific to their time, location, and needs. However, due to the nature of IoT devices and applications, some challenges remain to be addressed. Integration of IoT and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

cloud computing can help overcome these challenges. This solution takes advantage of the cloud, allowing IoT devices to be accessed as services via the cloud [6].

Cloud computing aims at providing users with services regardless of geographical constraints. In addition, it supports easy collection and distribution of services at a low cost over dissimilar environments [7]. However, users' cloud service requests are sometimes complicated and cannot be fulfilled using one service; rather, they may require multiple cloud services integrated from different clouds. To this end, some cloud service providers may form a federation of multiple clouds to fulfill users' requests while providing high quality of service (QoS) according to the service level agreement (SLA) [8]. Such a system of cloud services requires a mechanism for locating the required services within the multi-cloud environment (MCE), as well as finding the best way to collect and provide them together in an efficient manner while remaining within performance parameters. Such integration of multiple services into one service is a complex combinatorial problem known as service composition (SC) [9]. Most previous work on SC has focused on single clouds [9], [10]. However, because–as just noted–users' needs usually cannot be met from within a single cloud, some attempts have been made to study SC in multiple clouds. Because this is an NP-hard optimization problem, one solution is to apply bio-inspired algorithms to the service composition process [11]–[16].

This paper proposes a new bio-inspired algorithm for cloud SC in MCE. This is achieved via the following steps:

- First, we carefully study past and existing SC methods for single- and multi-cloud environments;
- Then we study bio-inspired methods and attempt to understand how they could be applied to the cloud;
- We then point out any shortcomings in existing work to find areas for improvement;
- Next, we design and implement a bio-inspired system for multi-cloud SC; and
- Finally, we test the proposed system and evaluate its performance.

The main contributions of this paper include a cuckoo-inspired problem-dependent heuristic designed specifically for the SC problem in MCE, benchmarked against existing algorithms, as well as a comparative analysis between the main studies related to the SC problem.

This paper is organized as follows: Section II presents background information. Section III reviews related work in SC in single- and multi-cloud environments. The design of the proposed algorithm is introduced in Section IV, while the methodology used to evaluate the algorithm and the results of the evaluation are presented in Section V. Finally, the paper concludes with a summary in Section VI.

II. BACKGROUND

This section briefly presents a general overview of cloud and web services. It also presents the history of our approach and other related topics.

A. WEB SERVICES AND CLOUD COMPUTING

Web services (WS) are software components with features that provide new ways to develop applications that meet the needs of internet users, thus making the web more dynamic. WS are designed to support inter-operable machine-to-machine communications over the internet [17]. QoS attributes are the non-functional requirements of services that are either measurable, such as response time, or non-measurable, such as security [14]. CC is becoming the trending platform for providing web services [15], [18]. CC is a general term for offering a convenient pool of resources on demand, delivered as a service over the internet, with the least possible management or interaction with the service provider. The resources are either the delivered applications or the hardware and systems software in data centers providing those services [19]. The desire for reliable access to affordable and trustworthy on-line resources has encouraged companies to move to CC and to approve of its usage. The resulting increases in CC usage have led to the release of even more cloud services. However, it is difficult to anticipate all the possible services that might be needed by cloud users [20]. In many real-world cases, complex and diverse services are needed and a single simple service cannot satisfy all the functional requirements of the case. To obtain a complex service, it is important to have a batch of atomic simple services working together; hence, there is a strong need to embed an SC system in CC environments [8]. Unfortunately, the optimum single services that are provided by the selected service providers have diverse quality in terms of their system attributes. This means that bringing them together to form the necessary complicated service must also be optimal [21]. The idea of composing business processes by determining and running the most suitable services rather than building new applications to satisfy a business requirement led to the initiation of SC [22].

B. SERVICE COMPOSITION (SC)

If the implementation of a service invokes additional services, then it is necessary to combine the functionalities of existing services. This implemented service is referred to as a composite service as it is developed through SC. Composite services are defined as an aggregation of elementary services; it is not simply a matter of putting together a set of WS [23]. Chaining of WS in composition can be identified based on semantic matching between output and input parameters of discovered services. Network QoS is an important metric in the web service composite (WSC) problem; it is also important to avoid violating the contract between users and providers with regard to the guaranteed QoS criteria–or service-level agreement (SLA)–during the SC process. Data-intensive SC faces many challenges including the users' QoS requirements, the mechanism used for composition and the objectives of the providers [19]. Cloud computing service composition (CCSC) is a problem with many potential solutions, among which one or a limited number of solutions are optimal, making it an optimization problem [24]. SC is an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

NP-hard problem [12], [25] in an n-dimensional hyperspace, as the QoS can be large, with non-linear effects on the SC objective function; hence, the solution space is too large to be searched in polynomial time. For data in the cloud environment, existing algorithms cannot reach a reasonable outcome within a reasonable amount of time. The factors affecting the complexity of this problem include the number of tasks to be performed, the number of candidate web services for each task, and the criteria on which the optimization relies. Most existing studies focus on applying intelligent optimization algorithms to find the optimal solution of the nonlinear integer programming problem, but the slow speed of convergence and the tendency to fall into the local optima are bottlenecks of these intelligent optimization algorithms [26]. The significant problems in SC that need to be solved include the discovery and selection of available services while still considering the necessary functionalities and the concepts used by the services. The most important thing is that the composition process should be fully automated so that the user is not included in any of the machine tasks. The first two problems show that in SC it is essential to choose WS according to the users' preferences and the QoS with the same functionality and not just according to the functionality of the services [11]. At present, despite the automation of composition, WSC operational problems include issues like not selecting the optimal composition, performance reduction with an increase in the number of WS, increases response time to requests, etc.; these problems are not yet fully resolved [13]. The dynamic nature of the cloud environment combined with changes exposes SC to a set of challenges such as describing and measuring QoS attributes of network services. Furthermore, cloud service providers often do not agree on the description of the QoS parameters and how to measure them, making the job of cloud developers even more difficult. Moreover, the presence of dependencies and conflicts between services further complicates the SC problem [27]. Network management is easy in a single-cloud environment, but if any one cloud server fails, the whole system fails. However, in multi-cloud environments, the real-time use of more than one cloud service decreases the possibility of interruptions due to a defect in the environment [28]. Due to the high cost of communication when using WS from different clouds, SC should effectively and efficiently minimize the number of clouds involved in the composition sequence. This is challenging, as the constraints provided by the services are usually different and hence must be taken into consideration when optimizing the cloud selection. In a multiple-cloud-based (MCB) environment, the service composer contains a number of related components, such as the cloud combiner, composition converter, service ontologies, and the MCB environment itself. For a specific request, a service requester provides its initial and goal descriptions using the service ontologies. Following that, the cloud combiner chooses a suitable cloud combination from the MCB. The composition domain and a composition problem are then formed by the composition converter based on the received request and the selected cloud combination. Finally, an optimization algorithm is used to find the solution that satisfies the requester's goal; this solution consists of an SC sequence. The role of the cloud combiner is to try to choose the most appropriate clouds for the service conversion and generate an SC scheme. Therefore, it is important to select a cloud set of high quality, as it affects the number of clouds involved in the final SC sequence and thus the efficiency with regard to time of finding a high-quality composition scheme [10], [18]. Today, WSC is done semantically through ontological descriptions of the WS. Ontologies are the primary building block for adding semantics to WS and are used to express users' preferences. They include the set of concepts, their properties and any relationships between them [13]. WSC can be represented as an inter-connected network (graph) in which each node represents a semantic WS with standard OWL-S and with the connecting edge labeled by the similarity between the two services [29]. Basically, there are three composition structures: sequential, parallel, and conditional. The sequential structure QoS provides the basis for computing the other structures [18]. SC is a multi-objective optimization problem (MOP) as it involves optimizing two or more objectives subject to certain constraints which may conflict with each other, for instance, maximizing availability and minimizing cost or maximizing reliability and minimizing total response time [16]. In multi-objective optimization problems, the aim is to find good compromises among the multiple objectives. This means finding a solution for which all objectives have been optimized to the point that further optimization of any of the objectives with result in deterioration of other objective(s). This phenomenon is referred to as pareto optimality. The goal is to find such a solution and quantify how much better it is than other such solutions, of which there are usually many, given a measurement standard (utility function) [16], [30]. SC involves the selection of the optimal composition solution so that the non-functional requirements like response time can be satisfied. This involves searching through a large number of services to choose the best combination that meets several objectives. To avoid processing the whole search space and thus reduce search time, heuristic methods can be used to find solutions [25]. Bio-inspired metaheuristics is a subclass of heuristic methods inspired by the behaviors of living beings that allow them to find solutions to survival problems. A metaheuristic is designed to find an approximate solution to a wide range of hard optimization problems without needing to adapt to each problem. The Greek prefix "meta" indicates that these algorithms are "higher level" heuristics, in contrast with problem-specific heuristics [31]. Research has demonstrated that bio-inspired principles can be successfully applied to solving optimization problems [19].

C. CUCKOO-INSPIRED ALGORITHMS

The survival of living things in the dynamic natural environment is determined by their ability to adapt to environmental changes, to self-organize in the absence of a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

central coordinator, and to self-optimize within their daily activities. Researchers noticed that principles inspired by the behavior of living organisms (e.g., foraging behavior of ants or bees, beehive construction mechanisms, cuckoo parenting techniques) could be applied to many complex problems from computer science to solve them more efficiently [23], [25], [32]. Cuckoos follow an aggressive parenting strategy; they lay their eggs in the nests of host birds. To do so, they search for nests containing eggs highly similar to their own eggs, enough so that the host bird cannot distinguish between its own eggs and the eggs of the cuckoo. As time passes and the baby cuckoo hatches from its egg, it tries to get rid of the host bird eggs in the nest [25], [33]. Cuckoo search (CS) is a metaheuristic population-based algorithm inspired by the parenting behavior of cuckoo birds, developed in 2009 by Xin-She Yang of Cambridge University and Suash Deb of C. V. Raman College of Engineering [34]. Subsequent studies proved that CS is simple and efficient in solving global optimization problems and that it is much more efficient than particle swarm optimization and genetic algorithms. This is partly because the number of parameters is smaller, allowing better adaptation to a wider class of optimization problems. Also, it has been proven that CS satisfies global convergence requirements and hence usually converges to a globally optimal solution, as it ensures a balance between exploration and exploitation. Although various applications of CS exist, there are few theoretical studies of this algorithm [35]–[39]. All of the previous cuckoo-inspired work involves metaheuristics, i.e., problem-independent algorithms, which begin with a random (or simple) solution and iteratively optimize it. Metaheuristics are extremely computationally expensive, with steep exponential running times. Therefore, the present project instead develops a problem-dependent heuristic (PDH). A PDH is strongly connected to a specific problem domain and takes into account the particularities of that problem. It works similarly to best-effort protocols, attempting to find a good guess (solution) from the first try without subsequent optimization iterations. Therefore, PDH's are more efficient than metaheuristics in terms of time and space, despite being more difficult to implement [31], [40].

III. RELATED WORK

Given recent trends, some solutions have been proposed to solve the SC problem in web services and cloud environments. SC is evolving as a widespread skill allowing mixing of distributed and unrelated services so as to combine and merge cloud services. It focuses on the innovation of a new service that includes current services in order to reduce cost and time and increase efficiency. Cloud SC methods can be divided into three popular categories: framework-based, agent-based, and heuristic-based [41]. Because composition is an NP-hard optimization problem [12], studies have shown that using heuristic-based methods like bio-inspired algorithms yields good results. This section focuses on the heuristic category and discusses several works in the field of bio-inspired SC for both single- and multi-cloud environments.

A. SINGLE CLOUDS SERVICE COMPOSITION

Zhang et al. [16] devised a divide-and-conquer control strategy involving the decomposition of a composite service into parallel execution paths and modeling a dynamic SC for each execution path as a multi-objective optimization problem. Using Visual Basic, they presented a new version of the ACO algorithm that treated the dynamic selection problem on the WS candidate graph as an ant system. The ant pheromone was updated and evaporated according to formulas taking into account the weight of each QoS attribute. Services are represented as nodes and any likely data dependency is represented by a directed acyclic graph; therefore, the service selection problem turns into a path selection problem. Zhang et al. simulated the services as four tuples: response time, cost, availability, and reliability, where the objective is to minimize the first two and maximize the last two. Their approach can find near-optimal solutions on the fly for multi-objective problems in huge search spaces and is scalable to support composition of very complex web services.

Xia et al. [11] built an improved ACO–OAWSCP–whereby the SC graph is generated dynamically. In the graph, each path represents a complete SC and nodes denote a match between services while arches represent an abstract service (services with the same function), where each abstract service has many concrete candidate services. This approach is dynamic in the sense that it can react to flow change of SC and can detect whether it is converging to the local optima, change the operating direction of the flow, and seek global optimization. The dynamic semantics of services could be applied at runtime, and ontologies were created to describe service capabilities, inputs, outputs, and execution semantics. The ontologies were built by using Web Ontology Language (OWL) and OWL-S technologies (which are XML based and hence platform independent and easily transferred and manipulated). The success ratio in seeking global optimization was used as their performance measure. They built a simulation application system using Visual C++.

Chifu et al. in [25] and [42] devised a cuckoo-inspired method for finding optimal or near-optimal solutions encoded in the enhanced planning graph (EPG). Construction of the search space does not start from a predefined work flow but is dynamically built based on the user request. The fitness function was an equation depending on the QoS score, semantic quality score, and weights corresponding to the relevance of QoS. They performed a set of experiments to adjust the parameters to provide optimal compositions in few iterations, without processing the entire search space. To measure the performance of their approach, they used percent of explored search space, simulation time, and number of cases in which the optimal solution was obtained. They carried out an analysis of how the values of four adjustable parameters influence the number of solutions processed, execution time, optimal fitness, and standard deviation from the global optimal solution provided through exhaustive search.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

This involved two steps: the first was an exhaustive search of the composition model to identify the optimal solution score, which was used to verify whether the average fitness obtained is close to the optimal. The second step involved iteratively fine-tuning parameters to identify their optimal values; the fitness of the optimal solution returned by the algorithm is compared to the fitness of the solution obtained by an exhaustive search.

Azari et al. [20] simulated three designs with good fitness and low response time in a single-cloud environment. The first design was of a bee colony algorithm for WS ranking based on a fitness function whose output is sent to the cuckoo algorithm for SC. The second design used K-means clustering of the dataset data before sending it to the bee colony algorithm. The last design was similar to the second but also used K-means clustering of the bee colony output (cuckoo algorithm input). Their fitness function was optimized when (response time + latency) / (availability + success + reliability + throughput) was minimized. When work is completed or new work is presented, the fitness function is re-calculated because the total size of the entrusted works to the service changes. They indirectly combined statistical information collected from the services in order to balance the load on the services close to the optimal solution and hence minimize network traffic. Moreover, they considered important aspects of the cloud environment such as self-adaptation of SC and SC based on customer demand. However, their simulation was for a small number of tasks.

Boussalia et al. [21] proposed an approach to determining the best WSC that uses semantic descriptions of WS and QoS as the criteria for optimization. Four QoS parameters were used: time, cost, availability, and reputation. The values of these four parameters were aggregated into a single real value to estimate the QoS solution. The sum was then computed of all the semantic distances between the inputs and outputs of each pair of WS belonging to the sequence of potential WSC, where the output of the first is the input to the second and so on. The best composition solution was gradually built starting from the user request without enumerating all possible solutions. Two optimization criteria were considered, QoS and semantic similarity (distance), by considering the pareto front. The performance was assessed using two metrics: contribution and entropy. The problem was formulated in terms of equations and their prototype was applied to a text translation case study using a Java-based prototype.

Ghobaei-Arani et al. [18] presented CSA-WSC, which was developed step by step based on the structure of the CS algorithm. Their solution to the cloud WSC problem is a sequence of WS ordered by usage and therefore the composite service is the process of selecting a subset of the provided WS. The fitness function was formulated as the maximum of a double summation of the quality of the qualitative parameters for a subset of WS multiplied by the weight of the qualitative parameters, where quality is less than or equal to the SLA. To measure the performance, they used response time, cost, reliability, and availability and to evaluate their system, they compared it with three effective WSC algorithms: a genetic-based generic algorithm, a genetic Particle Swarm Optimization (PSO) algorithm, and a greedy-based algorithm for WSC. Their algorithm showed a better fitness than the other three, i.e., it reduced both costs and response time. Moreover, they were able to prove that the convergence speed of the cuckoo algorithm is effective for finding near-optimal solutions and provides better fitness among the three different tested scenarios. They reached the conclusion that with regard to response time and resource costs, CSA-WSC is a more suitable algorithm for solving the WSC problem in geographically distributed cloud environments. They considered the QoS of the WS and network QoS using the Cloudsim toolkit. Results of the simulation indicate that they can achieve close to optimal result in terms of QoS criteria.

B. MULTI CLOUDS SERVICE COMPOSITION

Rostami et al. [13] presented an algorithm based on both clustering and the ant colony optimization (ACO) method. At design time, related services are hierarchically clustered (tree) and a semantic network is created to reduce time and complexity. After finding the WS using clustering, the ant colony algorithm is applied to find the best set of WS with the most composition capability. To measure the performance of their algorithm, they used precision, that is, the probability that the returned services are relevant to the user or the probability that the successfully retrieved services are relevant to the query. They evaluated the system by comparing the response time, accuracy, and composition optimality with other systems and found that they achieved better results. OWL-S language was used to semantically express descriptions of web services.

Kurdi et al. [7] proposed a combinatorial optimization algorithm for cloud SC (COM2) to compose services with a small number of examined services and combined clouds. They ensured that the cloud with the maximum number of service files is chosen before other clouds and assumed that clouds are sorted in descending order by number of service files. The clouds were repeatedly checked to see whether their services satisfied the user's request until the request was fulfilled. They measured the performance of their approach by using number of combined clouds |B| in the combiner list, and number of services examined by the composer |N| was used to compute execution time. COM2 challenges the available multi-cloud SC algorithms studied in [10] by achieving superior trade-off between the number of combined clouds and the number of examined services.

Yu et al. [15] also studied the WSC problem in MCE and presented two algorithms. The first–greedy WSC–uses a three-level tree and repeatedly selects the cloud that can provide the most services from among those requested until the selected clouds encompass all the requirements. The second algorithm–ACO-WSC–used artificial ants traveling on a logical digraph to construct cloud combinations. Each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

node represents a cloud base and edges connect each pair of nodes. Ants choose their path based on the pheromone and heuristic information of the edges. At first, each ant randomly selects a node (cloud) from the required service files and adds it to the solution it is constructing according to a probability based on the pheromone, heuristic information, and the gain for selecting that particular cloud. The bigger the difference between files in the clouds, the greater the probability that the ant chooses the edge. Yu et al. compared their results in a set of cases against four other algorithms, and their ACO found the optimal cloud combination with the minimum number of clouds among all cases and required less computation time. Their method was not a general optimization that finds random solutions and then chooses the best among these; rather, it tries to solve a specific problem. However, they used many parameters and their complexity was exponential.

This brief review of the literature shows that most research has been done on single clouds and many of the multi-cloud cases used high-cost metaheuristics. Therefore, we put forward a new approach for multi-cloud SC that combines the advantages of using guided and CS algorithms. Table 1 summarizes the different characteristics of the presented SC systems.

TABLE 1. Comparison of Different Cloud Service Composition Systems

System	Algorithm	Environment	Evaluation Metrics	Tools Used
[16]	Ant colony optimization	Single cloud	- Response time - Cost - Availability - Reliability	Visual Basic
[11]	Ant colony optimization on preference ontology	Single cloud	Success ratio	Visual C++ 6.0
[25], [42]	Cuckoo search	Single cloud	- Percent of explored search space - Simulation time - Number of cases with optimal solutions	-
[20]	Bee colony + cuckoo search	Single cloud	- Response time - Latency - Reliability - Availability - Throughput - Success of operation	-
[21]	Cuckoo search + bat inspired	Single cloud	- Time - Cost - Availability - Reputation	Math formulation + Java
[18]	Cuckoo search	Single cloud	- Response time - Cost - Reliability - Availability	Cloudsim toolkit
[13]	Clustering and ant colony optimization	Multiple cloud	- Response time - Accuracy - Composition optimality - Precision - Recall	-
[7]	Combinatorial optimization algorithm	Multiple cloud	- Execution time - Number of combined clouds	OWL-S XPlan package

IV. SYSTEM DESIGN

This section describes the design of the MultiCuckoo system, which uses the similarity scaler (SS) algorithm for efficient SC in multi-cloud environments.

A. SYSTEM MODEL

The multi-cloud model that we consider in this work is illustrated in Fig. 1. It is comprised of the following main components:

- A set of clouds (MCE) = {C1, C2, ..., Cm}, where each cloud has a set of service files F = {F1, F2, ..., Ff} and each file is composed of a set of services S = {S1, S2, ..., Ss};
- A service requester, which is a user interface that accepts a user's request and then displays the SC sequence formed;
- A similarity scaler to measure how similar the received request is to previously received requests;
- A cloud combiner to select suitable clouds (those with the highest number of matching services that fulfill the request) and then generate the combination list based on the suitable clouds; and
- A composer to receive the combination list of clouds from the combiner and then select the services that best fulfill the request from each cloud. Then, the composer produces the SC sequence.

B. SIMILARITY SCALER

The similarity scaler checks whether a similar request was previously received and if so, retrieves the fulfilled composed sequence for that request. It displays this sequence directly (hence eliminating the need to go through the time-consuming selection and composition process). However, if no similar request can be found, the requester sends the request to the cloud combiner, which in turn chooses the most appropriate set of clouds from the multiple-cloud environment according to the suggestions of the algorithm and sends it to the composer. Following that, the composer checks whether the sequence of services meets the request, generates a composition sequence, and sends it to the requester. In Algorithm 1 (which is invoked by Algorithm 2), we iterate on all the saved requests and their services. If the services of one of the saved requests happen to be similar to the newly requested services, the similarity ratio is higher. The propor-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

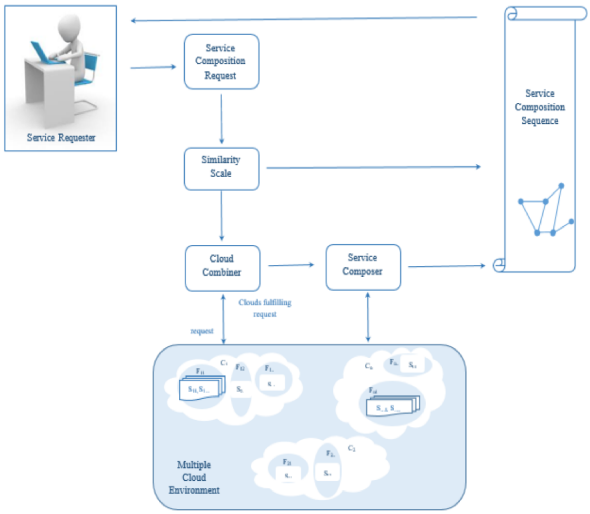


FIGURE 1. System overview.

tion of these services to the total services in this saved request is therefore added to the similarity ratio, as seen in lines 18 and 19 of the algorithm. Upon termination, we have the maximum possible similarity ratio of all the saved requests in the list. We check whether this ratio is larger than the similarity threshold and if it is, the saved composed service belonging to this request with the maximum (and hence best) ratio is directly retrieved. This alleviates the complication and time of running through the complex service composition process. A demonstration of the flow of Algorithm 1 is seen in Fig. 2.

C. CLOUD COMBINER

The problem of searching for a composite service that best matches a client request in MCE can be viewed from the cuckoo perspective. In this scenario, the cuckoo egg represents the service request, the bird eggs represent the cloud services, and the birds' nests represent the multiple clouds. With this analogy in mind, we built our MultiCuckoo algorithm to mimic real cuckoo behavior at a finer level of detail. The details of the MultiCuckoo algorithm are explained in Algorithm 2. It uses a saved list (CSL) containing all previously received requests and their composed services. First (line 1), we check whether the new request is sufficiently similar to any of the requests in the saved list according to a threshold clarified in Algorithm 1. If the new request is similar to a previous request, the previously composed service saved in CSL is instantaneously retrieved; otherwise, the composition process starts. Available clouds are accessed using Levy flight, which passes over the existing clouds and determines whether the current cloud service files can fulfill the user's request. If they cannot, the next cloud is checked until an appropriate cloud is found; otherwise, the algorithm will terminate after reaching the last cloud (lines 13-15). Once an appropriate cloud is located, it and its service files

Algorithm 1 Similarity Measure Algorithm

Input: newRequest(R), listRequests(LS), similarityThreshold
Output: composed services in LS for request similar to R
initialization:
1: ratio = 0

START: *// to iterate over listRequests*
2: maxRatio = 0
3: currentRequest = getNext(listRequests)
4: if (currentRequest is NULL) then *// finished listRequests*
5: goto **LAST**
6: requestSize = currentRequest.length
7: composedSequence (CSL) = currentRequest.getComposed
8: if (ratio > maxRatio) then
9: maxRatio = ratio

LABEL1: *// to iterate over services of currentRequest*
10: s1 = getNextServ(currentRequest)
11: if (s1 is NULL) then
12: goto **START** *// check next request in list*
13: else

LABEL2: *// to iterate over services of newRequest*
14: s2 = getNextServ(newRequest)
15: if (s2 is NULL) then *// finished services in new request*
16: goto **LABEL1**
17: else
18: if (s1 == s2) then *// currentRequest contributes to ratio*
19: ratio += (1 / requestSize)
20: goto **LABEL1**
21: else
22: goto **LABEL2**

LAST:
23: if (maxRatio >= similarityThreshold) then
24: return composedSequence

are added to CSL (line 9). If the user request is still not satisfied, the next cloud that contains new services that can partially fulfill the user's request is selected. Furthermore, to ensure that the current cloud Cm contains services that are not already included in CSL, the algorithm subtracts the content of CSL from the new services in Cm ($Cm \cap R$), as shown in line 9. If new services are unavailable, the selected cloud is disregarded (line 7), and the next cloud is considered. To cope with the dynamic nature of cloud environments, line 17 deletes any expired services in LS. The composed services make up a set of <cloud, service> that is updated upon receiving a new request. Periodically, this list is re-initiated to simulate the dynamic nature of clouds. The algorithm terminates if either the received request is successfully met

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

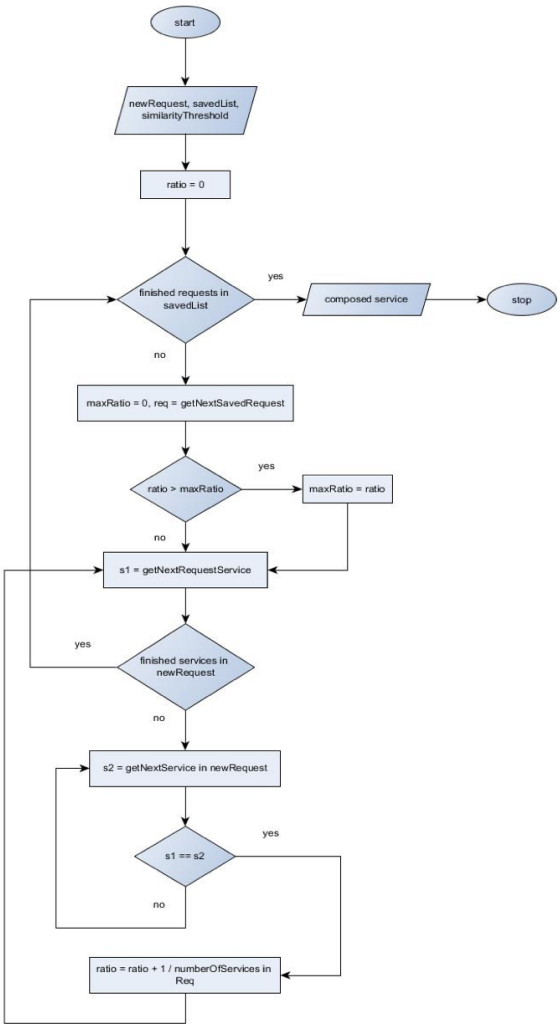


FIGURE 2. Similarity measure flowchart.

or there are no more clouds to be checked. A flowchart of the algorithm is shown in Fig. 3.

V. EVALUATION METHODOLOGY

A. MATERIAL LIST

The materials used to implement and evaluate the Multi-Cuckoo algorithm, including both hardware and software components is as follows:

- Hardware: Windows OS with 2.5 GHz Intel core i7 processor and 8 GB of RAM.
- Software: Java SE 10.0.1 [43] and Eclipse Oxygen 4.7.3 [44].

B. HYPOTHESIS AND PARAMETERS

The factors that affect the running time of SC algorithms in MCE include the average number of services per cloud and the total number of combined clouds in the system. To compare the performance of MultiCuckoo to the benchmarks and to build a robust evaluation framework, all algorithms

Algorithm 2 MultiCuckoo Service Composition Algorithm

Input: user request (R), number of clouds (n), clouds and their service files, saved list that is regularly updated (LS) = <R, service composition sequence CSL>
Output: composed sequence (CSL) for R (if any) and updated LS
Initialization:
1: CSL = Execute similarity measure (R,LS)
2: if (CSL NOT NULL) then
3: goto **STOP**
4: else

BEGIN:
5: Current_Cloud(Cm) = levy_flight(clouds)
6: if ((Cm services ∩ R) - CSL == φ) then
7: goto **NEXT_CLOUD**
8: else
9: CSL = CSL + <(Cm services ∩ R) - CSL, Cm>
10: if (CSL service list == R needed services) then
11: add CSL to LS as <R, CSL>
12: goto **STOP**

NEXT_CLOUD:
13: n = n - 1
14: if (n > 0) then
15: goto **BEGIN**

STOP:
16: periodically delete expired services from LS
18: return CSL

were implemented in the same environment and under similar conditions. Our hypothesis is that using MultiCuckoo for SC in an MCE will decrease the time needed to fulfill users' requests compared to the benchmark algorithms under variable environmental conditions of number of clouds and average number of services per cloud. For each experiment, we measured the following performance indicators:

- **Number of combined clouds**: the total number of clouds involved in the composite service.
- **Number of examined services**: the number of services that an algorithm needs to check while composing the required services.
- **Algorithm execution time**: the total time from the moment when a client enters his request until the system generates the corresponding composite service fulfilling the request.

As benchmarks, we used three well-established SC algorithms:

- **Combinatorial optimization algorithm for multiple cloud service composition (COM2)** [7]: This algorithm was developed to efficiently consider multiple clouds and was aimed at short execution time for the service composition process while reviewing a minimal

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

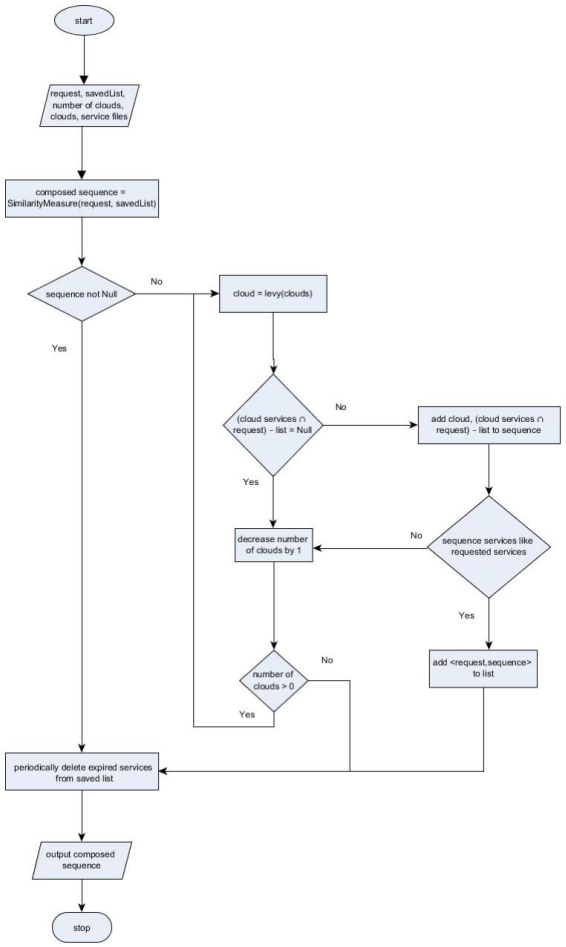


FIGURE 3. MultiCuckoo service composition algorithm flowchart.

number of clouds to reduce communication costs and hence the financial costs.

- **All clouds combination algorithm (ACC)** [10]: The ACC method can find a service composition sequence quickly but does not minimize the number of clouds in the final service composition sequence. This is problematic because utilizing web services distributed in different clouds greatly increase the communication cost and hence the financial costs.
- **Base cloud combination algorithm (BCC)** [10]: The BCC method has a very high time complexity (exponential in the number of clouds) as it needs to enumerate all possible cloud combinations in the worst case.

C. MCE SIMULATOR

To simulate the dynamic nature of an MCE and its components, we tried following a similar approach to [7], [10] which used the OWL-S XPlan package [45]. This package is an open-source service composition planner with a default web service test set that models different factors affecting the SC process in MCEs. Unfortunately, this tool has a very limited and fixed package with only four clouds and a maximum

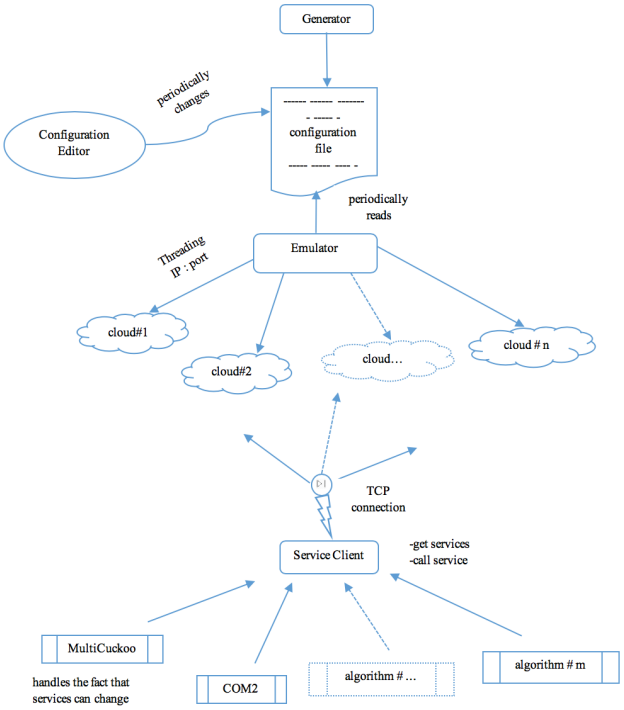


FIGURE 4. MCE simulator.

number of eight services. As a result, we had to implement the cloud environment from scratch as well as all of the benchmark algorithms. The main objective is to evaluate our MultiCuckoo algorithm and test its performance thoroughly in larger environments composed of a larger number of clouds and services. Our MCE simulator is implemented in a Java-based tool that uses client/server programming based on sockets to model the interactions in an MCE at different scales. Fig. 4 shows the main components of the simulator:

- **Generator**: The generator tool is responsible for the production of the configuration file, which stores the number of clouds in the MCE, their properties, and the available services.
- **Configuration editor**: The configuration editor is a thread that periodically changes the configuration file to reflect the dynamic nature of the cloud environment.
- **Emulator**: The emulator reads the configuration file and uses threading to produce the clouds needed on their corresponding IP address and port. The produced clouds are accessed by the service client through a TCP connection that retrieves a list of the cloud's services or calls a specific service in order to view its contents.
- **Service client**: The service client is a library used for service requests by our MultiCuckoo algorithm and all the other benchmark algorithms we implemented.

D. PERFORMANCE EVALUATION

The following steps were performed as part of the evaluation framework:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access

TABLE 2. Experimental Settings

Experiments	Number of Services	Number of Clouds
Set 1	15	10 15 20
Set 2	15 25 35	10

1) Building a Java-based multi-cloud environment simulator, as described in Section V-C.
2) Implementing the proposed algorithm as well as the benchmark algorithms in Java.
3) Simulating an MCE at different scales by:
4) Varying the number of clouds (NoC) in the environment at 10, 15, and 20 clouds. Due to limitations of the capabilities of our hardware, we were not able to experiment with a larger number of clouds.
5) Varying the average number of services per cloud (NoS) at 15, 25, and 35 services.
6) For each scenario, randomly generating user's requests for combined services to ensure representative samples of the possible number of combined services and clouds per request.
7) Running each algorithm with different user requests of different lengths.
8) Running the MultiCuckoo algorithm with different similarity thresholds and finding the most appropriate threshold.
9) Measuring the performance of the MultiCuckoo algorithm using:
    a) total number of examined services,
    b) number of combined clouds, and
    c) running time.
10) Comparing the results and analyzing the algorithms' performance.
11) Summarizing the different settings for the experiments, as given in Table 2.

E. RESULTS AND DISCUSSION

An important QoS attribute that should not be overlooked is the time taken to search for and compose the needed services [7], [10]. In an attempt to decrease this time, received requests and their composed services are saved in a list, as described in Section V-D. Upon receiving a new request, this list is first searched for any similar requests according to a similarity threshold and matching composed services are immediately retrieved. To examine the algorithms, we ran different service requests in which each set of services had a similarity threshold of 80% compared to the other service requests. In one set of test cases, the number of services was fixed at 15 and the number of clouds was varied as 10, 15, and 20 (Set 1). In th another set of test cases, the number of clouds was fixed at 10 while the number of services varied as 15, 25, and 35 (Set 2), as described in Table 2. Each scenario run was charted to enable observation and comparison of the algorithms' behavior. The following sections present the performance measures for all scenarios.

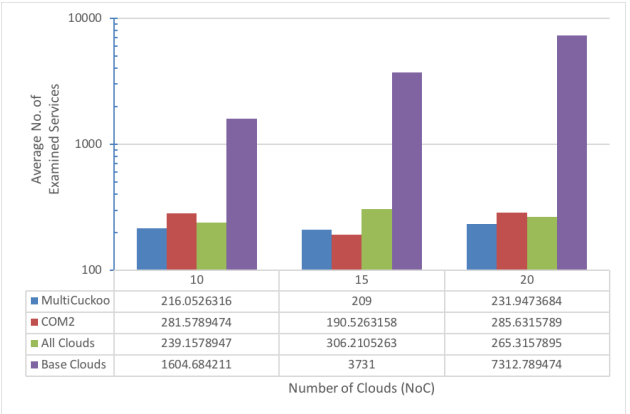


FIGURE 5. Average number of examined services when NoS = 15.

1) Number of Examined Services

The performance of the first scenario (fixed maximum services per cloud, 15; varied number of clouds) is shown in Fig. 5. We found that the MultiCuckoo algorithm examined a steadily increasing number of services as the number of clouds increased, but it always examined the lowest number of services compared to the other algorithms. COM2 and ACC showed similar behavior to one another. Importantly, we found that the number of services examined in the BCC algorithm increased much more than the increase in the number of clouds. BCC gives the maximum (and a very high average number of examined services) compared with the other approaches in this scenario.

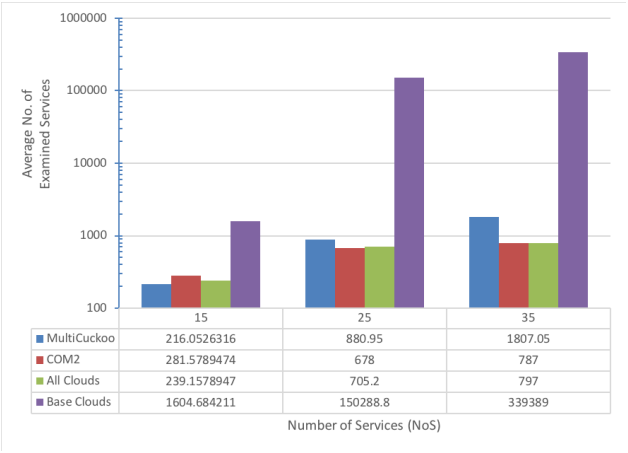
The average number of services examined by each algorithm in the second scenario (fixed number of clouds, 10; varied number of services) is shown in Fig. 6. Interestingly, again, BCC showed the worst performance, with a rapidly growing number of examined services. For the remaining algorithms, MultiCuckoo started off with the lowest number of examined services but concluded with the highest. COM2 and ACC showed similar performance: although they started off higher than MultiCuckoo, they both finished with a lower value.

2) Number of Combined Clouds

Fig. 7 shows the average number of combined clouds for all the algorithms in the first scenario (fixed maximum services per cloud, 15). For the MultiCuckoo algorithm, the number of combined clouds never exceeded three, whereas for the remaining algorithms there were never fewer than three clouds. For all the algorithms, no matter how many clouds were in the environment, the clouds included in the composed service was not greatly affected.

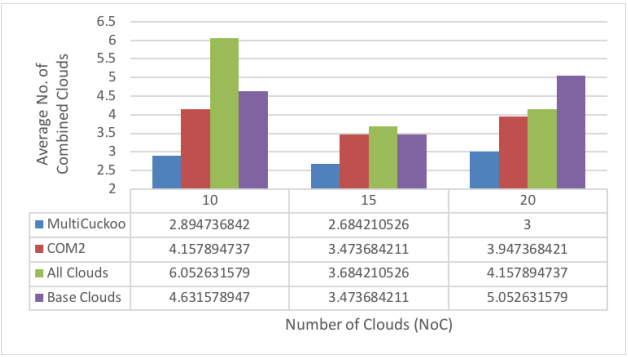
In the second scenario, illustrated in Fig. 8, our MultiCuckoo algorithm again never exceeded three clouds, while

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2872744, IEEE Access



	15	25	35
MultiCuckoo	216.0526316	880.95	1807.05
COM2	281.5789474	678	787
All Clouds	239.1578947	705.2	797
Base Clouds	1604.684211	150288.8	339389

FIGURE 6. Average number of examined services when NoC = 10.



	10	15	20
MultiCuckoo	2.894736842	2.684210526	3
COM2	4.157894737	3.473684211	3.947368421
All Clouds	6.052631579	3.684210526	4.157894737
Base Clouds	4.631578947	3.473684211	5.052631579

FIGURE 7. Average number of combined clouds when NoS = 15.

	15	25	35
MultiCuckoo	2.894736842	2.45	2.35
COM2	4.157894737	4.55	4.4
All Clouds	6.052631579	5	5.4
Base Clouds	4.631578947	4.85	4.8

FIGURE 8. Average number of combined clouds when NoC = 10.

the other algorithms were never lower than three clouds. Increasing the average number of services per cloud did not greatly affect the number of combined clouds in the composed service.

3) Average Running Time

In Fig. 9, it can be seen that the running time for COM2 and BCC increases by around 5 seconds upon increasing the number of clouds by 5. Interestingly, the MultiCuckoo

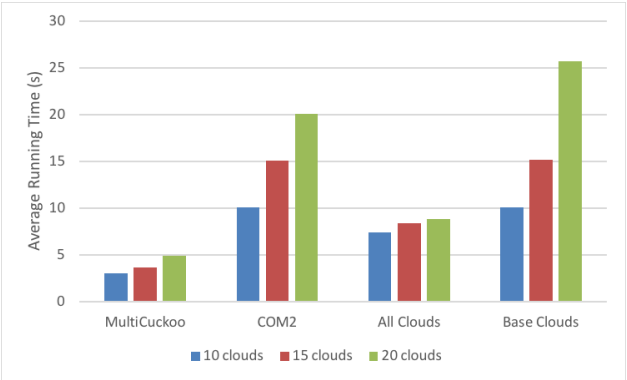


FIGURE 9. Average running time when NoS = 15.

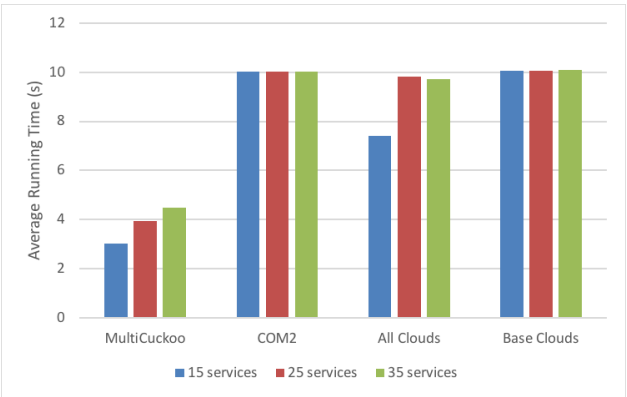


FIGURE 10. Average running time when NoC = 10.

algorithm and ACC algorithm show only a slight increase in running time upon the same increase in cloud count. As expected, the MultiCuckoo algorithm showed a much lower time than the other algorithms, at almost half the running time of the second best algorithm (ACC). BCC and COM2 showed the worst running times among the algorithms.

For COM2 and BCC in Fig. 10, the difference in running time upon increasing the number of services is negligible. However, for the MultiCuckoo algorithm and ACC, increasing the number of services resulted in increased running time, but the increase is small. Again, as expected, our MultiCuckoo algorithm shows the smallest running time among all the algorithms (almost half of the second fastest).

VI. CONCLUSION

In this paper, we proposed an algorithm designed to address challenges facing IoT service composition in multi-cloud environments. As the service composition problem is known for its complexity and is classified among the NP-hard problems, our approach used a bio-inspired algorithm that imitates the behavior of cuckoo birds (which examine the nests of other birds to find eggs similar to their own)to find existing composite services that can be used to efficiently fulfill a user's request. To the best of our knowledge, all previously proposed cuckoo-inspired algorithms are metaheuristic

and have exponential running times that consume system resources and are slow to find a good SC sequence. However, in this paper, we propose a problem-dependent heuristic that considers the service composition problem and its particularities in multi-cloud environments. Consequently, it consumes less time and fewer resources than metaheuristic algorithms. To evaluate the proposed algorithm–MultiCuckoo–a well-controlled experimental framework was used to benchmark the algorithm's performance against that of other outstanding service composition algorithms. The simulations showed that the MultiCuckoo algorithm examined a lower number of services and found a solution utilizing a smaller number of combined clouds than the other algorithms, which indicates that we accomplished our main goal. Furthermore, the results show a much shorter average running time for our algorithm compared to the other algorithms used in our experiments. These findings add to the growing body of literature on using bio-inspired behavior algorithms to solve complex problems. As future work, we intend to test the MultiCuckoo algorithm with other QoS performance measurements. Moreover, we plan to apply our proposed cuckoo-inspired algorithm in other areas to assess its applicability to other fields and compare its performance to existing systems in those areas.

REFERENCES

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," IEEE Communications Surveys Tutorials, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.

[2] S. S. Iyengar and R. R. Brooks, Distributed sensor networks: sensor networking and applications. Boca Raton, FL, US: CRC press, 2016.

[3] L. Altoaimy, A. Alromih, S. Al-Megren, G. Al-Hudhud, H. Kurdi, and K. Youcef-Toumi, "Context-aware gossip-based protocol for internet of things applications," Sensors, vol. 18, no. 7, 2018.

[4] M. Barcelo, A. Correa, J. Llorca, A. M. Tulino, J. L. Vicario, and A. Morell, "Iot-cloud service optimization in next generation smart environments," IEEE Journal on Selected Areas in Communications, vol. 34, no. 12, pp. 4077–4090, Dec 2016.

[5] M. M. E. Mahmoud, J. J. P. C. Rodrigues, S. H. Ahmed, S. C. Shah, J. F. Al-Muhtadi, V. V. Korotaev, and V. H. C. D. Albuquerque, "Enabling technologies on cloud of things for smart healthcare," IEEE Access, vol. 6, pp. 31 950–31 967, 2018.

[6] M. Aazam, I. Khan, A. A. Alsaffar, and E. N. Huh, "Cloud of things: Integrating internet of things and cloud computing and the issues involved," in Proceedings of 2014 11th International Bhurban Conference on Applied Sciences Technology (IBCAST) Islamabad, Pakistan, 14th - 18th January, 2014, Jan 2014, pp. 414–419.

[7] H. Kurdi, A. Al-Anazi, C. Campbell, and A. Al Faries, "A combinatorial optimization algorithm for multiple cloud service composition," Computers and Electrical Engineering, vol. 42, pp. 107–113, 2015.

[8] H. Mezni and M. Sellami, "Multi-cloud service composition using Formal Concept Analysis," J. Syst. Softw., vol. 134, pp. 138–152, 2017.

[9] X.-L. Wang, Z. Jing, and H.-z. Yang, "Service Selection Constraint Model and Optimization Algorithm for Web Service Composition," vol. 10, no. 5, pp. 1024–1030, 2011.

[10] G. Zou, Y. Chen, Y. Xiang, R. Huang, and Y. Xu, "AI Planning and Combinatorial Optimization for Web Service Composition in Cloud Computing," 2010.

[11] Y. Xia, C. Liu, Z. Yang, and J. Xiu, "The Ant Colony Optimization Algorithm for Web Services Composition on Preference Ontology," 2011.

[12] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani, "An approach for QoS-aware service composition based on genetic algorithms," Genet. Evol. Comput. Conf., 2005.

[13] N. H. Rostami, E. Kheirkhah, and M. Jalali, "An Optimized Semantic Web Service Composition Method Based on Clustering and Ant Colony Algorithm," Int. J. Web Semant. Technol., vol. 5, pp. 1–8, 2014.

[14] Y. Huo, Y. Zhuang, J. Gu, S. Ni, Y. Xue, Y. Huo, Y. Zhuang, J. Gu, S. Ni, and Y. Xue, "Discrete gbest-guided artificial bee colony algorithm for cloud service composition," Appl Intell, vol. 42, pp. 661–678, 2015.

[15] Q. Yu, L. Chen, and B. Li, "Ant colony optimization applied to web service compositions in cloud computing," Comput. Electr. Eng., vol. 41, pp. 18–27, 2015.

[16] W. Zhang, C. K. Chang, T. Feng, and H.-y. Jiang, "QoS-Based Dynamic Web Service Composition with Ant Colony Optimization," 2010 IEEE 34th Annu. Comput. Softw. Appl. Conf., pp. 493–502, 2010.

[17] S. R. Boussalia and A. Chaoui, "Optimizing QoS-Based Web Services Composition by Using Quantum Inspired Cuckoo Search Algorithm," 2014.

[18] M. Ghobaei-Arani, A. Rahmanian, M. Aslanpour, and S. Dashti, "CSA-WSC: Cuckoo Search Algorithm for Web Service Composition in Cloud Environments," Soft Comput., 2017.

[19] L. Wang, "Bio-inspired Cost-aware Optimization for Data-intensive Service Provision," Master's thesis, 2014.

[20] M. S. Azari, A. Bouyer, and N. F. Zadeh, "Service composition with knowledge of quality in the cloud environment using the cuckoo optimization and artificial bee colony algorithms," Conf. Proc. 2015 2nd Int. Conf. Knowledge-Based Eng. Innov. KBEI 2015, pp. 539–545, 2016.

[21] S. Rayene Boussalia, A. Chaoui, A. Hurault, M. Ouederni, and P. Queinnec, "Multi-objective Quantum Inspired Cuckoo Search Algorithm and Multi-objective Bat Inspired Algorithm for the Web Service Composition Problem," Int. J. Intell. Syst. Technol. Appl. J. Intell. Syst. Technol. Appl., vol. 15, no. 2, pp. 95–126, 2016.

[22] F. Dahan, K. E. Hindi, and A. Ghoneim, "An Adapted Ant-Inspired Algorithm for Enhancing Web Service Composition," Int. J. Semant. Web Inf. Syst., vol. 13, no. 4, 2017.

[23] I. Salomie, V. R. Chifu, C. B. Pop, I. Salomie, V. R. Chifu, and C. B. Pop, "Hybridization of Cuckoo Search and Firefly Algorithms for Selecting the Optimal Solution in Semantic Web Service Composition," 2014.

[24] A. Jula, E. Sundararajan, and Z. Othman, "Cloud Computing Service Composition: A Systematic Literature Review," Expert Syst. Appl., vol. 41, pp. 3809–3824, 2014.

[25] V. R. Chifu, C. B. Pop, I. Salomie, D. S. Suia, and A. N. Niculici, "Optimizing the Semantic Web Service Composition Process Using Cuckoo Search," 2012.

[26] J. Zhou and X. Yao, "Multi-objective hybrid artificial bee colony algorithm enhanced with Lévy flight and self-adaption for cloud manufacturing service composition," Appl. Intell., vol. 47, no. 3, pp. 721–742, 2017.

[27] J. O. Gutierrez-Garcia and K. M. Sim, "Agent-based Cloud Service Composition," Appl. Intell., vol. 38, no. 3, pp. 436–464, 2013.

[28] S. Asghari and N. J. Navimipour, "Service Composition Mechanisms in the Multi-Cloud Environments: A Survey," 2016.

[29] A. Sawczuk, D. Silva, E. Moshi, H. Ma, and S. Hartmann, "A QoS-Aware Web Service Composition Approach Based on Genetic Programming and Graph Databases," 2017.

[30] B. Huang, C. Li, and F. Tao, "A Chaos Control Optimal Algorithm for QoS-based Service Composition Selection in Cloud Manufacturing System," Enterp. Inf. Syst., vol. 8, no. 4, pp. 445–463, 2014.

[31] I. Boussaïd, J. Lepagnot, and P. Siarry, "A Survey on Optimization Metaheuristics," Inf. Sci. (Ny)., vol. 237, pp. 82–117, 2013.

[32] Q. Yu, L. Chen, and B. Li, "Ant Colony Optimization Applied to Web Service Compositions in Cloud Computing," Comput. Electr. Eng., vol. 41, no. 4, pp. 18–27, 2015.

[33] X. S. Yang and S. Deb, "Cuckoo Search via Levy Flights," 2009 World Congr. Nat. Biol. Inspired Comput. NABIC 2009 - Proc., pp. 210–214, 2009.

[34] X.-s. Yang and M. Karamanoglu, "Swarm Intelligence and Bio-Inspired Computation: an Overview," Elsevier, 2013.

[35] M. Jamil and H. J. Zepernick, "Multimodal Function Optimisation with Cuckoo Search Algorithm," Int. J. Bio-Inspired Comput., vol. 5, no. 2, p. 73, 2013.

[36] X.-S. Yang, "Bat Algorithm and Cuckoo Search: A Tutorial," Artif. Intell., Evol. Comput. Metaheuristics, SCI 427, pp. 421–434, 2013.

[37] I. Fister Jr and X.-s. Yang, "Cuckoo Search: A Brief Literature Review," 2014.

[38] M. Tian, K. Hou, Z. Wang, and Z. Wan, "An improved Cuckoo search algorithm for multi-objective optimization," Wuhan Univ. J. Nat. Sci., vol. 22, no. 4, pp. 289–294, 2017.

[39] X.-s. Yang and S. Deb, "Cuckoo Search: Recent Advances and Applications," 2013.

[40] N. Siddique and H. Adeli, "Nature Inspired Computing: An Overview and Some Future Directions," Cognit. Comput., vol. 7, no. 6, pp. 706–714, 2015.
[41] A. Vakili and N. J. Navimipour, "Comprehensive and systematic review of the service composition mechanisms in the cloud environments," J. Netw. Comput. Appl., vol. 81, pp. 24–36, 2017.
[42] H. Chifu, V.R., Pop, C.B., Salomie, I., Suia, D.S., Niculici, A., Negrean, A. and Jeflea, "Optimising the semantic web service composition process using bio-inspired methods," Int. J. Bio-Inspired Comput. J. Bio-Inspired Comput., vol. 5, no. 4, pp. 226–238, 2013.
[43] "Java SE Development Kit 10," http://www.oracle.com/technetwork/java/, 2017.
[44] "Eclipse Oxygen," https://www.eclipse.org/oxygen/, 2017.
[45] "OWLS-Xplan Service Composition Planner," http://www-ags.dfki.uni-sb.de/ klusch/owls-xplan/, 2017.