Towards Network-aware Service Composition in the Cloud

Adrian Klein
The University of Tokyo
Japan
adrian@nii.ac.jp

Fuyuki Ishikawa
National Inst. of Informatics
Japan
f-ishikawa@nii.ac.jp

Shinichi Honiden
The University of Tokyo
National Inst. of Informatics
Japan
honiden@nii.ac.jp

ABSTRACT

Service-Oriented Computing (SOC) enables the composition of loosely coupled services provided with varying Quality of Service (QoS) levels. Selecting a (near-)optimal set of services for a composition in terms of QoS is crucial when many functionally equivalent services are available. With the advent of Cloud Computing, both the number of such services and their distribution across the network are rising rapidly, increasing the impact of the network on the QoS of such compositions. Despite this, current approaches do not differentiate between the QoS of services themselves and the QoS of the network. Therefore, the computed latency differs substantially from the actual latency, resulting in suboptimal QoS for service compositions in the cloud. Thus, we propose a network-aware approach that handles the QoS of services and the QoS of the network independently. First, we build a network model in order to estimate the network latency between arbitrary services and potential users. Our selection algorithm then leverages this model to find compositions that will result in a low latency given an employed execution policy. In our evaluation, we show that our approach efficiently computes compositions with much lower latency than current approaches.

Categories and Subject Descriptors

H.3.5 [**On-line Information Services**]: Web-based services; H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Algorithms, Performance, Measurement, Management

Keywords

Web Services, Cloud, Network, QoS, Optimization, Service Composition

1. INTRODUCTION

Service-Oriented Computing (SOC) is a paradigm for designing and developing software in the form of interoperable services. Each service is a software component that encapsulates a well-defined business functionality.

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

1.1 Service Composition

SOC enables the composition of these services in a loosely coupled way in order to achieve complex functionality by combining basic services. The value of SOC is that it enables rapid and easy composition at low cost [18].

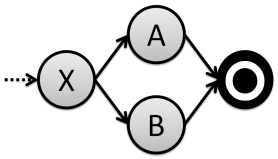


Figure 1: Workflow

Such compositions result in workflows consisting of abstract tasks, like in Fig. 1, that can either be created manually or computed automatically by planning approaches [23]. For each abstract task, concrete services have to be chosen in order to execute the workflow.

1.2 QoS-aware Service Composition

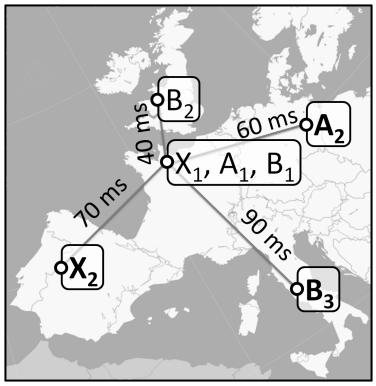
For service compositions, functional and non-functional requirements [17] have to be considered when choosing such concrete services. The latter are specified by Quality of Service (QoS) attributes (such as latency, price, or availability), and are especially important when many functionally equivalent services are available. The QoS of a composition is the aggregated QoS of the individual services according to the workflow patterns [9], assuming that each service specifies its own QoS in a Service Level Agreement (SLA). The user can then specify his QoS preferences and constraints for the composition as a whole, e.g. a user might prefer fast services, but only if they are within his available budget. Choosing concrete services that are optimal with regard to those preferences and constraints then becomes an optimization problem which is NP-hard [19]. Thus, while the problem can be solved optimally with Integer (Linear) Programming (IP) [26], usually heuristic algorithms like genetic algorithms are used to find near-optimal solutions in polynomial time [6].

1.3 Service Composition in the Cloud

With the advent of Cloud Computing and Software as a Service (SaaS), it is expected that more and more web services will be offered all over the world [5]. This has two main impacts on the requirements of service compositions.

1.3.1 Network-Awareness

First, the impact of the network on the QoS of the overall service composition increases with the degree of distribution of the services. A service offered in Frankfurt would in many cases not only be used in Germany but worldwide. Despite this, current approaches do not differentiate between the QoS of services themselves and the QoS of the network. The common consensus is that the provider of a service has to include the network latency in the response time he publishes in his SLA. This is not a trivial requirement, since latency varies a lot depending on the user's location [27]; e.g. users in Frankfurt, New York and Tokyo may have quite different experiences with the same service.



(a) Deployed Services

Serv.	Exec. Time
X_1	100 ms
X_2	**90 ms**
A_1	100 ms
A_2	**95 ms**
B_1	100 ms
B_2	90 ms
B_3	**70 ms**

(b) Service SLAs

Figure 2: Distributed Deployment

For example, consider the abstract workflow depicted in Fig.1, and the corresponding concrete services (X_1, X_2, A_1, etc.), in which X_1 executes task X, etc., in the times conforming to Fig. 2b. We can see the deployed services and the network delays between the different deployment locations in Fig. 2a. In such a scenario, current approaches would select X_2, A_2 and B_3, because their QoS are optimal, and this results in a total execution time of 265 ms. Now, if a user in France wants to execute the workflow, the round trip times would add over 200 ms to that time. In comparison, executing X_1, A_1 and B_1 would just take 300 ms and only incur a minimal delay because of round trip times. On the other hand, if providers added the maximum delay for any user to the execution time in their respective SLAs, this would guarantee a certain maximum response time to all users, but it would discourage users from selecting local providers and instead favor providers with the most homogeneous delays for all users (e.g. providers in the center of Fig.1 in France).

1.3.2 Scalability

Secondly, as the number of services increases, the scalability of current approaches becomes crucial. That is, as the number of services grows, more functionally equivalent services become available for each abstract task, and this causes the complexity of the problem to increase exponentially. One can argue that the number of services offered by different providers with the same functionality might not grow indefinitely. Usually a small number of big providers and a fair number of medium-sized providers would be enough to saturate the market. The crucial point is that in traditional service composition, most providers specify just one SLA for the service being offered. Only in some cases up to a handful of SLAs may be specified; for example, a provider might offer platinum, gold and silver SLAs to cater to different categories of users with different QoS levels [22].

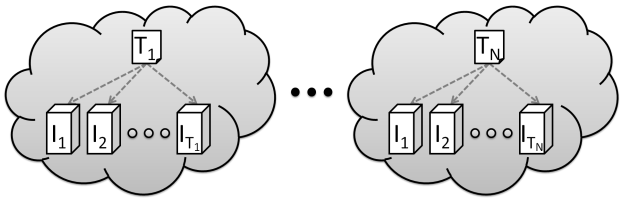


Figure 3: Deployed Services by Different Providers

In contrast to this, in a network-aware approach each provider has to provide many SLAs, as in Fig. 3. Given an abstract task T a provider that offers a service T_i for T might have to supply different SLAs for each of his instances $I_1, \ldots, I_{T_i}$. Each instance might run on a physical or virtual machine with different characteristics (CPU, memory, etc.). Also, these instances might be executed at completely different locations in order to offer the service in different countries. As we want to differentiate between each instance, we get many more choices for each individual task. For example, whereas previous approaches might assume 50 different providers for each task [25], and, thus, consider 50 choices per task, we might easily have to consider 2500 choices, if we assume that each provider deploys 50 instances of his service on average.

1.4 Contributions

Thus, we propose a new approach towards network-aware service composition in the cloud, consisting of the following three contributions:

1. Network Model. We adopt a generic network model that can be fed in a scalable way by stand-of-the-art algorithms from the network research community. We enhance this model by adding scalable facilities to find services which are close to certain network locations or network paths. This allows us to estimate the network latency between arbitrary network locations of services or users and to find services that will result in low latency for certain communication patterns.

2. Network-aware QoS Computation. We specify a realistic QoS model that allows us to compute network QoS, such as latency and transfer rate. Our network-aware QoS computation can handle input-dependent QoS, as well. (For example, a video compression service could specify that the service's execution time depends on the amount of input data supplied.)

3. Network-aware Selection Algorithm. Our selection algorithm is based on a genetic algorithm. We redefine all generic operations including initial generation, mutation, and crossover. By leveraging our network model, we tailor those operations to the problem of service composition in the cloud in order to improve the scalability and the solution quality of our algorithm.

In our evaluations, we show that the latency of service compositions computed by our algorithm is near-optimal and much lower than current approaches. Furthermore, we show that our approach has the scalability needed for ser-

vice composition in the cloud; it beats current approaches not just in absolute numbers but also in runtime complexity.

The structure of this paper is as follows. Section 2 reviews related work. Section 3 defines our approach. Section 4 evaluates the benefits of our approach in a cloud environment. Section 5 concludes the paper.

2. RELATED WORK

In this section, we survey related work from the following five categories.

2.1 QoS-aware Service Composition

The foundation for our research is in [26]. That paper introduces the QoS-aware composition problem (CP) by formalizing and solved it with (Linear) Integer Programming (IP), which is still a common way to obtain optimal solutions for the CP. A genetic algorithm is used in [6, 8]. Moreover, many efficient heuristic algorithms have been introduced [2, 15, 12, 25], the most recent being in [13, 20]. All these approaches share the same definition of the CP, which ignores the QoS of the network connecting the services. Except for IP which requires a linear function to compute the utility of a workflow, most approaches can be easily augmented with our network-aware QoS computation.

As for the recent skyline approach [3] that prunes QoS-wise dominated services, this approach would have to be modified in order to be applicable to our cloud scenario. Even services with the same QoS can give quite different experiences to different users depending on the location of the services and the users; that means QoS-wise dominated services can only be pruned per network location. In such case, the benefit in our scenario would be quite small when there are many different network locations.

2.2 Advanced QoS

The approaches described above simply aggregate static QoS values defined in SLAs. A QoS evaluation depending on the execution time is described in [14]. In a similar way, our algorithm computes the starting time of the execution of each service, so it can be used to compute time-dependent QoS, as well. SLAs with conditionally defined QoS are described in [11]; these can be considered to be a special case of input-dependent QoS, and, thus, they can be handled by our approach as well.

Constraints whereby certain services have to be executed by the same provider are given in [16]. Placing such constraints on critical services could also reduce network delays and transfer times. However, doing so would require significant effort to introduce heuristic constraints and there is no guarantee that this would lead to a near-optimal solution.

2.3 Network QoS

Many studies, such as [4, 10], deal with point-to-point network QoS, but do not consider services and compositions from SOC. One of the few examples that considers this is [24], which looks at service compositions in cloud computing. The difference between this study and our approach is that instead of a normal composition problem, a scheduling problem is solved where services can be deployed on virtual machines at will. However, it is not clear if that approach can handle the computation of input-dependent QoS and network transfer times, because the authors did not provide a QoS algorithm.

2.4 Network Coordinate System

In the related field of network research, there are many algorithms that build network coordinate (NC) systems to estimate the latency between any two points. Two state-of-the-art algorithms, Vivaldi [1] and Phoenix [7], build reliable network coordinate systems in a scalable fashion. Our approach does not depend on a specific network coordinate system; it uses a generic network model based on two-dimensional coordinates that can be fed by NC systems, as in [1], that are based on the Euclidean distance model (which is the most widely used model of NC systems).

2.5 Workflow Scheduling

In the related field of workflow scheduling, a workflow is mapped to heterogeneous resources (CPUs, virtual machines, etc.), and information about the network is sometimes considered, as well. The idea is to achieve a (near-)optimal scheduling minimizing the execution time; this is often done by using greedy heuristic approaches, like HEFT [21]. The reason such greedy algorithms seem to suffice is that only one QoS property (response time) is optimized, and that no QoS constraints have to be adhered to, greatly simplifying the problem. Thus, while the setting is similar to ours, the complexity of the problem is quite different, because we optimize multiple QoS properties under given QoS constraints. In addition, algorithms, like HEFT [21], often work like a customized Dijkstra, which does not scale well in our problem setting, as our evaluation shows.

3. APPROACH

In this section we define our approach. We will present our proposed network model first and then describe our network QoS computation that makes use of this model. After that, we define our proposed network selection algorithm that makes use of the network model, as well.

3.1 Network Model

Our network model consists of two parts: a network coordinate system that forms the basis of our network-aware approach and a locality-sensitive hashing scheme that allows us to find services that are close to certain network locations or network paths.

3.1.1 Network Coordinate System

In our cloud scenario, we face the challenge of dealing with a huge number of services, N. We use existing network coordinate systems in order to compute the latency between any two network locations of services or users. Probing all pairwise link distances would require $O(N^2)$ measurements,

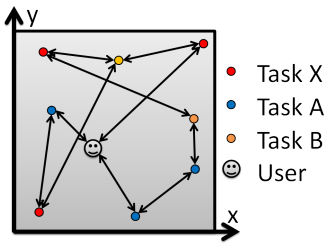


Figure 4: Network Model

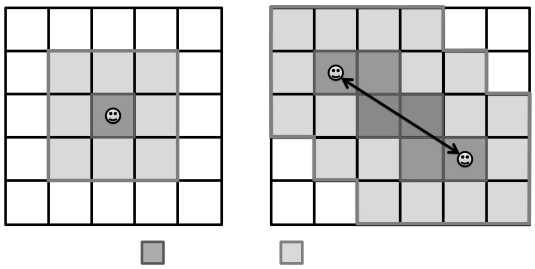


Figure 5: Computed Hulls

giving us exact values, but it would obviously not scale. Also, while services deployed in the cloud might exist for some time (allowing us to cache their latencies), users would frequently show up at new locations, and, thus, they would have to ping all existing services before they could use our approach; this would be prohibitive.

Therefore, we use network coordinate (NC) systems that give us an accurate estimate of the latency between any two network locations; these systems require $O(N)$ measurements in total and just $O(1)$ measurements for adding a new network location. We decided to base our generic NC system on algorithms, like Vivaldi [1], that are based on the Euclidean distance model. While such NC systems might use any number of coordinates, we will consider here a system of two coordinates for the sake of simplicity. Accordingly, each network location (including services and users) is placed in a two-dimensional space, as in Fig. 4, and the latency between two locations simply corresponds to their Euclidean distance. This allows our network-aware QoS computation to accurately estimate the QoS of service compositions in the cloud.

3.1.2 Locality-sensitive Hashing Scheme

In order for our network-aware selection algorithm to work efficiently, it is not enough to be able to compute the latencies between arbitrary network locations. Additionally, we need to be able to find services that are close to certain network locations or network paths, so that we can efficiently search for service compositions with lower latency. We realize this functionality with a simple locality-sensitive hashing scheme (LSH) that relies on our generic NC system.

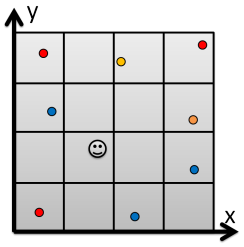


Figure 6: Network Model Grid

We build an overlay grid on top of the two-dimensional coordinate space, as in Fig. 6. Each cell of the grid corresponds to a bucket that can contain services. Given this grid, we hash every network location to the corresponding bucket. Close network locations correspond to identical or close buckets. We maintain a separate grid for each unique task of our workflow, and, thus, support the two following operations with our LSH, illustrated in Fig. 5.

1. Compute the Hull of a Location Given a location, we can find services which perform a certain task close to that location by computing an outer hull of a certain range around the desired location.

Algorithm 1: computeHullOf($loc, task, range$)

```
1  buckets := empty list
2  grid := lookup grid corresponding to task
3  locBucket := hash location loc to its bucket in grid
4  locX := locBucket.x and locY := locBucket.y
5  foreach x ∈ [locX − range; locX + range] do
6  |  foreach y ∈ [locY − range; locY + range] do
7  |  |  if (x, y) within grid then
8  |  |  |  buckets := buckets ⋃ {grid(x, y)}
9  |  |  end
10 |  end
11 end
12 return buckets
```

2. Compute the Hull of a Network Path Given a path, e.g. between two network locations, we can find services which perform a certain task and directly lie on the path or its computed outer hull of a certain range.

Algorithm 2: computeHullBetween($loc_1, loc_2, task, range$)

```
1  buckets := empty list
2  locations := compute trajectory between loc_1 and loc_2
3  foreach loc_x ∈ locations do
4  |  buckets := buckets ⋃
   |  computeHullOf(loc_x, task, range)
5  end
6  return buckets
```

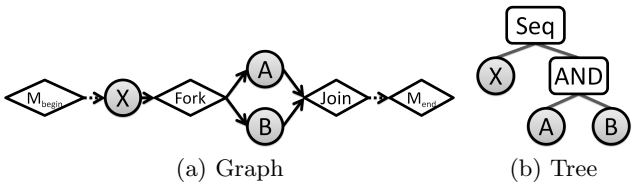
After calling $computeHullOf$ or $computeHullBetween$, we just need to fetch the services contained in the returned buckets. The computation of these two hulls (that contain both the inner and outer hulls depicted in Fig. 5) allows our network-aware selection algorithm to minimize the latency of service compositions, as we will show in detail later.

3.2 Network-aware QoS Computation

Our network-aware QoS computation consists of two phases. The first phase simulates the execution of the workflow in order to evaluate input-dependent QoS and network QoS. The second phase aggregates these QoS over the workflow structure.

1. Simulated Execution We assume that a workflow is either given as a directed graph as in Fig. 7a or is converted into one (e.g. from a tree as in Fig. 7b), before we simulate the execution of the workflow according to Alg. 3.

2. QoS Aggregation In the second phase, we take the obtained QoS for each node and aggregate it in a hierarchical manner (over the tree representation as in Fig. 7b) according



(a) Graph (b) Tree

Figure 7: Different Representations of a Workflow

to the commonly used aggregation rules from [9, 25] that take into account workflow patterns, like parallel(AND) or alternative(OR) executions and loops. Just for the runtime of the workflow, we keep the computation from the first phase, because we cannot compute it with a hierarchical aggregation.

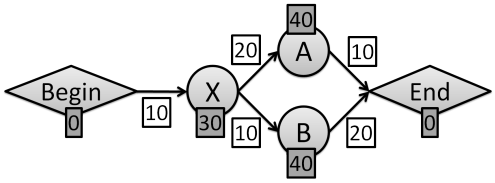


Figure 8: QoS of a Workflow

3. Example If we annotate the services of our previous workflow example with execution durations, and their network links with network delays, as in Fig. 8, our algorithm will produce the values of Fig. 9 for the execution times of the nodes ($start/end$) in five steps[1].

#	Begin	X	A	B	End
0	0 / ?	0 / ?	0 / ?	0 / ?	0 / ?
1	**0 / 0**	10 / 0	0 / ?	0 / ?	0 / ?
2	0 / 0	**10 / 40**	60 / ?	50 / ?	0 / ?
3	0 / 0	10 / 40	**60 / 100**	50 / ?	110 / ?
4	0 / 0	10 / 40	60 / 100	**50 / 90**	110 / ?
5	0 / 0	10 / 40	60 / 100	50 / 90	**110 / 110**

Figure 9: Simulated QoS

This simple example shows that hierarchical QoS aggregation alone would not work, because A and B would be aggregated together first. This makes it impossible to compute the correct network QoS, because the maximum of the delay of the incoming and outgoing nodes of (A,B) each would be aggregated as 20, adding up to 40. Actually, however, both paths that go through the incoming and outgoing nodes of (A,B) have a cumulative delay of 30, which is 10 less than the aggregated value.

3.3 Network-aware Selection Algorithm

Our selection algorithm is based on a genetic algorithm which can solve the service composition problem in polynomial time. First, we give a basic overview of genetic algorithms. Then, we introduce the customizations of our algorithm that are tailored to the problem of service composition in the cloud.

[1]the computational steps are denoted in the # column

Algorithm 3: simulateExecution(graph)

```
1  foreach vertex v ∈ graph do
2  |  v.execStart = 0
3  |  v.requiredIncoming = |v.incoming|
4  end
5  while ∃ v ∈ graph . v.requiredIncoming = 0 do
6  |  pick any v ∈ graph . v.requiredIncoming = 0
7  |  evaluate QoS of v
8  |  v.executionEnd = v.execStart + v.qos.runtime
9  |  foreach w ∈ v.outgoing do
10 |  |  netQoS = getNetworkQoS(v, w)
11 |  |  duration = v.resultSize / netQoS.transferRate
12 |  |  transEnd = v.execEnd + netQoS.delay + trans
13 |  |  w.execStart = max{w.execStart, transEnd}
14 |  |  w.requiredIncoming -= 1
15 |  end
16 end
```

3.3.1 Genetic Algorithm

In a genetic algorithm (GA), possible solutions are encoded with genomes which correspond to the possible choices available in the problem. For a service composition, a genome, as in Fig. 11, contains one variable for each task of the workflow, with the possible values being the respective concrete services that can fulfill the task.

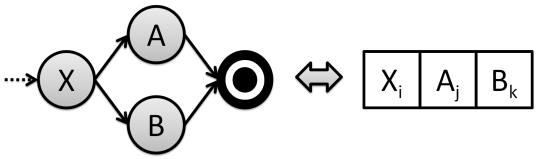


Figure 11: Workflow with corresponding Genome

Given a fitness function that evaluates how good a possible solution (individual) is, the GA iteratively finds a near-optimal solution as follows. First, an initial population is generated. Then, in every iteration, individuals are selected, changed either by mutation or crossover operations, and inserted into the new population for the next iteration. This procedure is repeated until a convergence criteria is met, which checks if the fitness of the population has a reached a satisfactory level, or if the fitness does not improve anymore.

3.3.2 Initial Population

The initial population is usually generated completely randomly. While it is desirable to keep some randomness for the GA to work properly, adding individuals that are expected to be better than average, has beneficial effects both on the speed of convergence and on the final solution quality. Thus, we generate about a quarter of the initial population with our *Localizer* heuristic, which is illustrated in Fig. 10. Using our previously built network model, we build a possible initial solution, task by task. In each step, we only select randomly among those services that can fulfill the current task and are close to the network location of their preceding service. In the last step, we also consider that the final result has to be sent back to the user, and select randomly among those services that are close to the network path between the preceding location and the user's location.

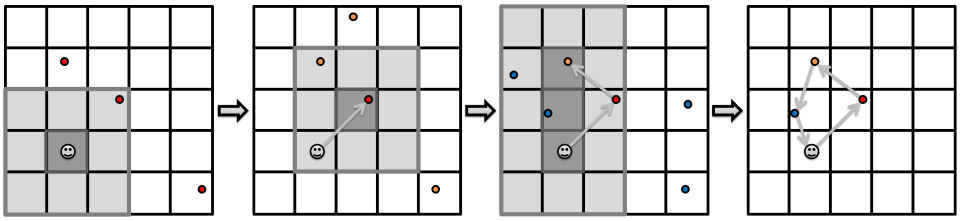


Figure 10: *Localizer* Heuristic for a good Initial Solution

3.3.3 Selection

We select individuals that are to be reproduced via mutation and crossover through a roulette-wheel selection. The probability of an individual with fitness f_i to be chosen out of a population of size N is:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Thus, it is a fitness-proportionate selection ensuring that better individuals have a higher chance of being chosen. There is no need to customize the selection operator to our problem, as it is already reflected in the fitness function.

3.3.4 Elitism

Elitism keeps the top-k individuals seen so far in order to achieve convergence in a reasonable time. We keep the most fit percent of our population in each generation.

3.3.5 Mutation

The purpose of the mutation operator is to change individuals slightly in a random way in order to randomly improve their fitness sometimes and to escape local optima. For this part, we partly use the standard uniform mutation operator which changes each element of the genome with equal probability, and on average changes about one element, as shown in Fig. 12.

Figure 12: Uniform Mutation

In addition, we use our own *Localizer* mutation operator which does a combination of mutation and local search. First, some parts of the genome are picked randomly. For each picked part, we then try to exchange the chosen service with a constant number of services that are randomly chosen close to the network path between the preceding and following service (graph-wise), as shown in Fig. 13. The best replacement is kept.

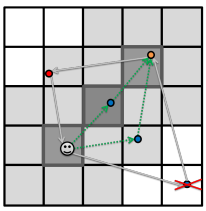


Figure 13: *Localizer* Mutation

This operator speeds up the convergence and improves the solution quality, but the key is to make it not too aggressive, and to only use it for a certain fraction of the mutations. (If used intensively, it could cause the GA to converge too quickly to a local optimum.)

3.3.6 Crossover

The crossover operation combines two individuals (parents) to create improved individuals (offspring) that can draw from the good points of both parents. A standard single-point crossover operator is depicted in Fig. 14: A single point of the genome is chosen, and the new individuals are recombined from both parents around that point.

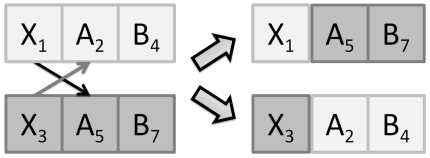


Figure 14: Single-Point Crossover

In an analogous way, two-point crossover, three-point crossover, etc. can be implemented until one has a uniform crossover operator in which every part of the genome of the parents is randomly distributed among the offspring. Our *Localizer* crossover is a customized uniform crossover operator which, for each part of the genome, chooses a value from its parents with a certain probability. We build one final offspring, task by task, by choosing between the parents par_1 and par_2. Let d_{prev_i} be the distance to the location of the previous service (or the user at the start), and d_{next_i} be the distance to the location of the following service (or the user at the end). If the following service is not chosen yet, we take the average of the locations of both possible following services from the two parents. The probability that we choose a parent par_i for the current task then is:

$$p_i = \frac{d_{prev_i} + d_{next_i}}{\sum_{j=1}^{2} d_{prev_j} + d_{next_j}}$$

The effect is that the offspring is a kind of smoothened version of the parents with regard to the network path. This quickly minimizes the latency of our offspring. Note that when the parents are spread in a similar fashion network-wise, our *Localizer* crossover comes very close to standard uniform crossover. We hence only use our crossover, as it was shown to be effective in our experiments.

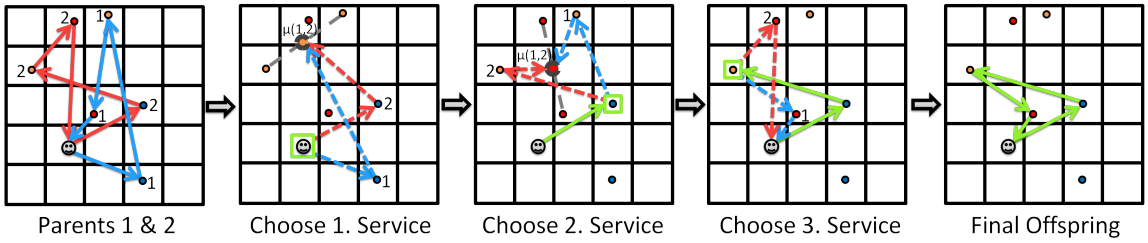


Figure 15: *Localizer* Crossover

4. EVALUATION

In this section we evaluate our approach. First, we describe the setup of our evaluation and the algorithms we want to compare. Then, we evaluate the optimality and scalability of our approach.

4.1 Setup

The evaluation was run on a machine with an Intel Core 2 Quad CPU with 3 GHz. All algorithms were evaluated sequentially and given up to a maximum of 2 GB of memory if needed. We randomly generated a network containing 100,000 unique locations within our two-dimensional coordinate space $[0, 1] \times [0, 1]$. Then, we generated our workflows with randomly inserted tasks and control structures. For each task, we randomly chose a number of those network locations and created services there. Fig. 16 depicts an example of a generated workflow of length 10:

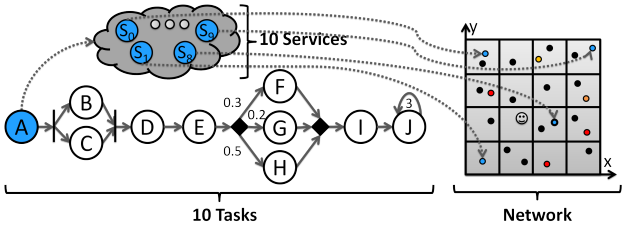


Figure 16: Example Workflow of Size 10

We generated workflows with sizes between 10 and 80 (in steps of 10); these sizes are comparable to those of other recent approaches with 10 [3], up to 50 [25] or 100 [2]. We varied the number of services available per task between 500 and 4000 (in steps of 500), which is considerably more than what most of the previous studies have used (50 [25], 250 [20] or 350 [15]). As mentioned, the reason that we consider such a large number of services per task is that we want to differentiate between instances of the same service that are deployed in different network locations. The QoS attributes of each service were generated according to a uniform distribution.

The network latency l between two services or between a user and a service was computed from the Euclidean distance d of their network coordinates as follows:

$$l(lc_1, lc_2) = \begin{cases} 0 & \text{if } d(lc_1, lc_2) < \epsilon \\ 20 + 400 \cdot d(lc_1, lc_2) & \text{otherwise} \end{cases}$$

Thus, the latency for two locations is either 0 if they are in a local area network (closer than a small ϵ), or between 20 and 420 ms depending on their distance so that a good range of realistic values will be covered.

4.2 Algorithms

We chose several algorithms and compared their optimality and scalability.

Random. A simple random algorithm that picks all services at random from the available choices. It shows the expected value of a randomly chosen solution and provides a baseline that more sophisticated algorithms should be able to outperform easily.

Dijkstra. An optimal algorithm for the shortest-path problem. It can be used to find the service composition with the lowest latency, against which we can conveniently compare our algorithm. However, *Dijkstra* cannot be used with QoS constraints or when we have several QoS.

GA 100. The normal *GA* that uses uniform crossover and uniform mutation. The size of the population is 100. We supply this GA with the standard QoS model, which does not model network latency. Accordingly, each service provides not just its execution time, but adds its expected (maximum) latency on top of that. This is the current standard approach for the service composition problem.

GA* 100. The same settings as *GA*, except that we provide *GA** with our network model that allows it to accurately predict the network latency. G* represents naïve adoption of the current standard approach to service composition in the cloud.

GA* 50. *GA** with a population size of only 50. We evaluated *GA* 50*, as well, but while it was slightly faster than *GA* 100*, we decided to keep it out of the final evaluations results, because of the bad quality of the solutions it found.

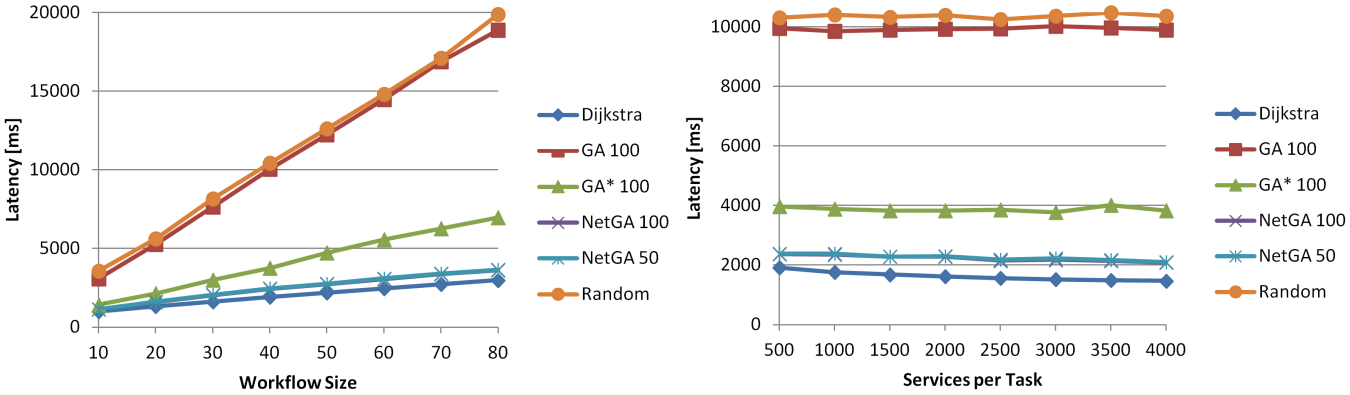
NetGA 100. Our network-aware approach as described in Sect. 3. The size of the population is 100.

NetGA 50. Our network-aware approach with a population size of only 50. Its results differ from *NetGA 100* by only about one or two percentage points, but it requires less runtime, as we will see later. Note that in the following graphs the results of *NetGA 50* might sometimes overlie those of *NetGA 100*, because their results are so close.

The convergence criteria are identical for all GAs: If the fitness of the best individual does not improve by at least one percentage point over the last 50 iterations, the algorithm terminates.

4.3 Optimality

To evaluate the optimality of our approach we plotted the latencies of the compositions found by all algorithms versus an increasing problem size.



(a) Lat. vs. Workflow Length (500 Services per Task)

(b) Lat. vs. Services per Task (Workflow Length 40)

Figure 17: Latency vs Increasing Problem Size

4.3.1 *Workflow Length*

Fig. 17a plots the latency against an increasing workflow size with a fixed number (500) of services available per task. We randomly generated the QoS of the services, and fixed the expected runtime of the generated workflows as 1000 ms without considering network latency. That means for a workflow size of 10, each task in a sequential workflow would have an expected runtime of 100 ms; for 20 sequential tasks, the expected runtime would be 50 ms, etc. Fig. 17a reveals the following findings.

Impact of Network. Even though the expected runtime of the workflow does not change, the optimal latency found by *Dijkstra* increases linearly with the workflow size, as more network communication is needed.

Impact of Network Model. The latency of the compositions found by *GA 100* is pretty much the same as that of the *Random* approach. Thus, the current standard approach is not effective at finding compositions with low latency at all. This backs up our claims that services should specify their runtime in their SLAs and that approaches should properly account for network latency in order to be effective.

Impact of Network-aware Approach. *GA* 100* finds compositions with latencies that get farther from the optimal latency with increasing workflow size: For a workflow size of 10, the compositions are about 1.5 times slower on average, and for a workflow size of 80, they are more than twice as slow. On the other hand, our network-aware approach *NetGA* manages to keep a good approximation ratio of the optimal solution regardless of the workflow size. The results of *NetGA 100* and *NetGA 50* are almost identical.

Algorithm	a	o(t)	Θ(t)	O(t)
Dijkstra	32	$n^{1.18}$	$\mathbf{n^{1.19}}$	$n^{1.22}$
GA 100	2.7	$n^{1.62}$	$n^{1.67}$	$n^{1.77}$
GA* 100	3.4	$n^{1.71}$	$\mathbf{n^{1.72}}$	$n^{1.84}$
NetGA 100	17	$n^{0.95}$	$n^{0.97}$	$n^{1.03}$
NetGA 50	10	$n^{0.87}$	$\mathbf{n^{0.90}}$	$n^{0.92}$

Figure 18: Empiric Algorithmic Complexity ($a\ n^x$) (n := Workflow Length; 500 Services per Task)

4.3.2 *Services per Task*

Fig. 17b shows that the optimal latency decreases slightly as the number of services per task increases, as there are more choices available. Except for *NetGA* which also finds compositions with slightly lower latency, the other GAs do not show a decreasing tendency. Still, the latency does not change much, once there are more than 2000 services available per task.

4.4 Scalability

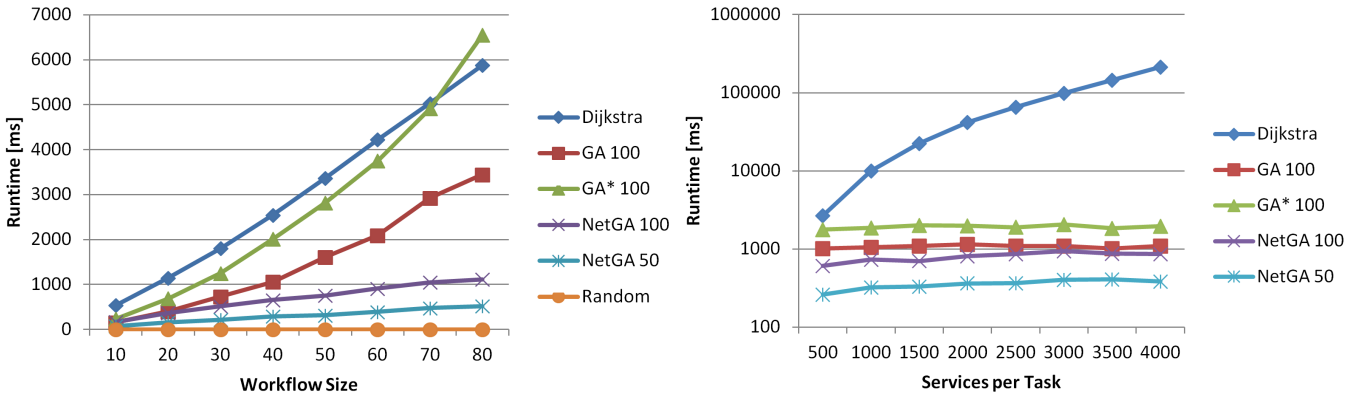
We evaluated the scalability of our approach under the same settings as for optimality.

4.4.1 *Workflow Length*

In Fig. 19a, we can see that *GA* runs considerably faster than *GA**. However, the qualities of the solutions found by *GA* are not much better than those of *Random*, which takes much less than 1 ms to compute its results. Therefore, it seems that *GA* is only faster, because it fails to improve the quality of its solutions significantly, thus, converging more quickly to a bad local optimum. *GA**, on the other hand, finds solutions with a somewhat reasonable quality (as we saw earlier), but takes a considerable runtime to do so; it takes more than double the runtime of *NetGA* for a workflow size of 40, and more than six times for a workflow size of 80.

Algorithmic Complexity. In order to summarize the different runtimes from Fig. 19a, we analyzed our empirical results and computed approximations for the algorithmic complexities. We assumed runtimes of the form of $a \cdot n^x$ where a is a constant factor, n stands for the workflow size, and x is a power. First, we found the best combination of a and x that minimizes the square of the error of the representation $\Theta(t)$. Then, we computed x for the tightest upper and lower bound given the previously computed a.

Fig. 18 explains why at first *GA** is slightly faster than *Dijkstra*, but then overtakes it at a certain workflow size: *GA**'s algorithmic complexity is much higher, only its constant factor is smaller than *Dijkstra*'s. In addition, we can conclude that *GA* will also overtake *Dijkstra* for sufficiently big workflows, whereas *NetGA* will not. We can also see that *NetGA* is not only much faster than *GA**; it also has a much lower algorithmic complexity, which is roughly linear with



(a) Runtime vs. Workflow Length (500 Services per Task)

(b) Runtime vs. Services per Task (Workflow Length 40)

Figure 19: Runtime vs. Increasing Problem Size

regard to the workflow size (as far as we can observe it in our experiments). This is due to the good convergence behaviour of *NetGA* which effectively finds compositions with low latency. Furthermore, if we are willing to sacrifice one or two percentage points of solution quality, we can use *NetGA 50* instead of *NetGA 100* and get even shorter runtime and smaller algorithmic complexity.

4.4.2 Services per Task

The runtime of *Dijkstra* significantly increases, as the number of services per task increases. Thus, *Dijkstra* quickly becomes infeasible for practical purposes; it takes 10 seconds for 1000 services per task, and almost 100 seconds for 3000 services per task. Also note that the runtimes of all GAs increase by only about 10 to 15%, so all of them scale well in this regard in our scenario. For the same workflow size, *GA* 100*, *NetGA 100* and *NetGA 50* maintain runtime ratios of about 4:2:1 regardless of the number of services per task.

5. CONCLUSION

In this paper we described a network-aware approach to service composition in a cloud, consisting of a network model, a network-aware QoS computation, and a network-aware selection algorithm. We showed that our approach achieves near-optimal solutions with low latency, and that it has roughly linear algorithmic complexity with regard to the problem size. It beats current approaches for which the approximation ratio of the optimal latency worsens with increasing problem size, and which have algorithmic complexity between $n^{1.5}$ and n^2. This means that our network-aware approach scales comparatively well in situations where the network has a significant impact on the QoS of service compositions. Our approach would facilitate service compositions in settings where dozens of services each get deployed on dozens or even hundreds of instances in clouds worldwide.

As future work, we would like to investigate how our approach fares in scenarios with multiple QoS where low latencies are beneficial for the user, but not the top priority, or might even conflict with other QoS. In some cases, some fine-tuning might be needed to always beat the standard approaches. Furthermore, we plan to conduct real-world experiments in a distributed setting in order to test how well our network model can cope in practice, e.g. when the network latency changes dynamically due to factors such as congestion. Additionally, we would like to evaluate our approach with other, more complex coordinate systems (especially, three-dimensional ones).

6. ACKNOWLEDGEMENTS

We would like to thank Florian Wagner for the fruitful discussions and the detailed feedback that helped us improve our approach. Adrian Klein is supported by a Research Fellowship for Young Scientists from the Japan Society for the Promotion of Science.

7. REFERENCES

[1] Vivaldi: a Decentralized Network Coordinate System. *Communication*, 34(4):15–26, 2004.

[2] M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 881–890, 2009.

[3] M. Alrifai, D. Skoutas, and T. Risse. Selecting Skyline Services for QoS-based Web Service Composition. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 11–20, 2010.

[4] R. Boutaba. QoS-aware service composition in large scale multi-domain networks. In *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.*, pages 397–410, 2005.

[5] K. Candan, W.-S. Li, T. Phan, and M. Zhou. Frontiers in Information and Software as Services. In *ICDE '09. IEEE 25th International Conference on Data Engineering, 2009*, pages 1761 –1768, 2009.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, 2005.

[7] Y. Chen, X. Wang, C. Shi, E. K. Lua, X. Fu, B. Deng, and X. Li. Phoenix: A Weight-based Network

Coordinate System Using Matrix Factorization. *IEEE Transactions on Network and Service Management*, pages 1–14, 2011.

[8] M. Jaeger and G. Mühl. QoS-Based Selection of Services: The Implementation of a Genetic Algorithm. In *KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC), Bern, Switzerland*, pages 359–370, 2007.

[9] M. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS Aggregation for Web Service Composition using Workflow Patterns. In *EDOC '04: Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*, pages 149–159, 2004.

[10] J. Jin, J. Liang, and K. Nahrstedt. Large-scale QoS-Aware Service-Oriented Networking with a Clustering-Based Approach. In *Proceedings of 16th International Conference on Computer Communications and Networks*, pages 522–528, 2007.

[11] A. Klein, F. Ishikawa, and B. Bauer. A Probabilistic Approach to Service Selection with Conditional Contracts and Usage Patterns. In *ICSOC-ServiceWave '09: Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, pages 253–268, 2009.

[12] A. Klein, F. Ishikawa, and S. Honiden. Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy. In *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 182–196, 2010.

[13] A. Klein, F. Ishikawa, and S. Honiden. Efficient Heuristic Approach with Improved Time Complexity for QoS-aware Service Composition. In *IEEE International Conference on Web Services (ICWS 2011)*, pages 436–443, 2011.

[14] B. Kloepper, F. Ishikawa, and S. Honiden. Service Composition with Pareto-Optimality of Time-Dependent QoS Attributes. In *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 635–640, 2010.

[15] F. Lecue and N. Mehandjiev. Towards Scalability of Quality Driven Semantic Web Service Composition. In *ICWS '09: IEEE International Conference on Web Services*, pages 469–476, 2009.

[16] D. A. Menascé, E. Casalicchio, and V. Dubey. On Optimal Service Selection in Service Oriented Architectures. *Performance Evaluation*, 67(8), 2010.

[17] J. O'Sullivan, D. Edmond, and A. Ter Hofstede. What's in a Service? *Distributed and Parallel Databases*, 12(2–3):117–133, 2002.

[18] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-Oriented Computing: A Research Roadmap. In *Service Oriented Computing (SOC)*, Dagstuhl Seminar Proceedings, 2006.

[19] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, Dept. of Computer Science, 1995.

[20] F. Rosenberg, M. B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar. Metaheuristic Optimization of Large-Scale QoS-aware Service Compositions. *IEEE, International Conference on Services Computing*, pages 97–104, 2010.

[21] H. Topcuoglu, S. Hariri, and M. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[22] H. Wada, J. Suzuki, and K. Oba. Queuing theoretic and evolutionary deployment optimization with probabilistic slas for service oriented clouds. In *World Conference on Services - I*, pages 661–669, 2009.

[23] F. Wagner, F. Ishikawa, and S. Honiden. QoS-Aware Automatic Service Composition by Applying Functional Clustering. *IEEE International Conference on Web Services (ICWS)*, pages 89–96, 2011.

[24] Z. Ye, X. Zhou, and A. Bouguettaya. Genetic Algorithm Based QoS-Aware Service Compositions in Cloud Computing. In *Database Systems for Advanced Applications*, pages 321–334, 2011.

[25] T. Yu, Y. Zhang, and K.-J. Lin. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[26] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003.

[27] Z. Zheng, Y. Zhang, and M. R. Lyu. Distributed QoS Evaluation for Real-World Web Services. pages 83–90, 2010.