A Scalable Distributed Architecture for Network- and QoS-aware Service Composition (IJAWS 2012)

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

Fuyuki Ishikawa
GRACE Center and National Institute of Informatics
Tokyo, Japan
f-ishikawa@nii.ac.jp

Shinichi Honiden
University of Tokyo and National Institute of Informatics
Tokyo, Japan
honiden@nii.ac.jp

Abstract—Service-Oriented Computing (SOC) enables the composition of loosely coupled service agents provided with varying Quality of Service (QoS) levels, effectively forming a multiagent system (MAS). Selecting a (near-)optimal set of services for a composition in terms of QoS is crucial when many functionally equivalent services are available. As the number of distributed services, especially in the cloud, is rising rapidly, the impact of the network on the QoS keeps increasing. Despite this and opposed to most MAS approaches, current service approaches depend on a centralized architecture which cannot adapt to the network. Thus, we propose a scalable distributed architecture composed of a flexible number of distributed control nodes. Our architecture requires no changes to existing services and adapts from a centralized to a completely distributed realization by adding control nodes as needed. Also, we propose an extended QoS aggregation algorithm that allows to accurately estimate network QoS. Finally, we evaluate the benefits and optimality of our architecture in a distributed environment. (IJAWS 2012)

Keywords—Service Composition, Quality of Service, Distributed System, Network

I. INTRODUCTION

Service-Oriented Computing (SOC) enables the composition of service agents in a loosely coupled way by realizing many ideas from the research of multiagent systems (MAS). Services can be thought of as specialized agents only allowing access through their published interfaces. SOC requires the modeling of autonomous and heterogeneous service components which form a MAS. The value of SOC is achieved by enabling rapid and easy composition of services with low costs.

A. QoS-aware Service Composition

For service compositions, functional and non-functional requirements have to be considered. The latter are specified by Quality of Service (QoS) attributes and are especially important when many functionally equivalent services are available. A composition should be optimal in regards to the user’s QoS preferences and constraints. The QoS of a composition is the aggregated QoS of its services according to workflow patterns, given each service’s Service Level Agreement (SLA).

Thus, current approaches only consider the QoS of services themselves and ignore the QoS of the network. One reason is that on a small scale services might be executed in a local network where network QoS is not significant. With a growing distribution of services, this is no longer true. Finally, the common opinion is that the provider of a service has to take the network into account in his SLA. This is not trivial, as, in general, response times vary a lot depending on the user’s location, making it hard for the provider to predict what kind of network QoS his users will experience. Thus, the current practice becomes less accurate, as the number of distributed services keeps rising, deployed in locations around the world. Therefore, we think it is essential to develop approaches tackling service composition in a network-aware manner to reduce the burden for providers to supply universal SLAs, and to improve the QoS for users.

B. Network Delay

The example in Fig. 1 illustrates the necessity of a network-aware approach. Consider the abstract workflow depicted in Fig. 1(a), the corresponding concrete services (X1, X2, A1, etc.) where X1 performs task X, etc. and the execution times conforming to Fig. 1(c). We can see the deployment...
of the services and the network delays between the different deployment locations in Fig. 1(b). In such a scenario, current approaches would select $X_2$, $A_2$ and $B_3$, because their QoS are optimal, resulting in a total execution time of 255 ms. Now, if a user in France wants to execute the workflow, the round trip times would add over 300 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 would add 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 also discourage users from selecting local providers and instead favor providers with the most homogeneous delays towards all users (e.g. providers in the center of Fig. 1(b) in France).

C. Service Architecture

In fact, the standard service-oriented architecture (SOA) enforces this problem. As shown in Fig. 2, the actual idea of the architecture is to make the network locations transparent to the middleware, taking away this relevant decision information from the composition process.

![Figure 2: Standard Architecture](image)

In such a centralized architecture [17] all communication between middleware and service happens through the Enterprise Service Bus (ESB). If we execute the workflow from Fig. 1 this means that there will be no direct communication between X and B. Instead, the middleware will call X, wait for the result of X, and then call B, etc., causing unnecessary network overhead. While there also exist decentralized SOAs avoiding that overhead, as in the context of executing e.g. WS-CDL [8] service choreographies, the standard architectures for service orchestrations are all centralized, including the one assumed by BPEL [15].

A simple way to solve this problem would be to design a SOA in which services can communicate their results directly to each other. Even if we ignore the business logic of a workflow that has to be evaluated somewhere, there still are significant obstacles to such an architecture. First, it would not be trivial to implement such an architecture, as services would have to perform several middleware functions (e.g. wait for/buffer input data, handling missing data/timeouts, etc.). Also, there is the principle of the separation of concerns which tells us that a service provider might not want or should not be bothered with implementing such additional functionality. Finally, probably the biggest obstacle is posed by the fact that current standards are already widely adopted. Introducing new requirements for all service providers would most likely lead either to poor acceptance or to a fragmentation of the market. Thus, we instead propose a scalable distributed service architecture that minimizes network delay and transfer times, while requiring no changes on the provider side, allowing for a gradual adoption.

D. Network Transfer

In addition to network delays, the transfer of data across the network can also account for a significant amount of time, as our example in Fig. 3 illustrates.

![Figure 3: Audio Encoding Example](image)

Given services $M_1$ and $M_2$ that take raw audio data and return encoded audio data, their difference is the execution time ($Ex. Time$) and the compression rate ($Comp. Rate$) in relation to the input data. For instance, sending 100 MB of raw audio data to $M_1$ takes 8 s over a 100 MBit/s link, with 50 MB of resulting encoded data ($Ex. Data$). While most current approaches would probably prefer $M_1$, because of its lower execution time (and higher transfer rate), in practice calling $M_1$ is only faster, if we send less than 200 MB of audio data. For more data $M_2$ is faster because of its superior compression. This example also shows that the QoS of a service cannot always be specified as static values in a SLA, as it is common. Instead a service provider might need to specify input-dependent QoS; especially for data-driven scenarios this can be quite significant.

E. Contributions

Thus, we present the following contributions to realize a distributed architecture for network- and QoS-aware service composition:

1) A scalable **distributed service architecture** composed of a flexible number of distributed control nodes; it generalizes the standard architecture and adapts from a completely centralized to a completely distributed realization by adding control nodes as needed.

2) A network-aware **QoS aggregation algorithm** that allows to accurately estimate the QoS of service compositions executed in a distributed fashion through our architecture, extending [9].

Note that our architecture can be gradually adopted, as it requires no changes to existing services. It generalizes the implicitly introduced completely distributed architecture.
from our previous network-aware approach [9]. The network QoS we consider are latency and transfer rates. In our evaluation we show that our architecture is near-optimal even with a limited number of control nodes.

The structure of this paper is as follows. Section II reviews related work. Section III defines our approach consisting of our architecture and QoS aggregation algorithm. Section IV evaluates the benefits of our approach. Finally, Section V concludes the paper.

II. RELATED WORK

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

A. QoS-aware Service Composition

The foundation for our research is given in [23] where the QoS-aware composition problem (CP) is introduced. Common notions, which we also use, are given, and the problem is formalized and solved with (Linear) Integer Programming (IP), which is still a common way to obtain optimal solutions for the CP. A genetic algorithm (GA) is used in [3], [5]. Besides, many efficient heuristic algorithms have been introduced in [1], [13], [22], and most recently in [2]. 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 two-phased QoS algorithm.

B. Advanced QoS

The previously mentioned approaches all simply aggregate static QoS values defined in SLAs. Time-dependent QoS evaluated depending on the execution time are given in [12]. As we will see, our algorithm computes when the execution of each service starts, so we can also compute time-dependent QoS. SLAs with conditionally defined QoS are given in [10], which can be considered a special case of input-dependent QoS, and, thus, can be handled by our approach, as well.

In [14] constraints on the choice of providers are given, requiring certain services to be executed on the same provider. Introducing such constraints for critical services could also reduce network delay and transfer times to some extent. This would require a significant effort to introduce such heuristic constraints though, while still not necessarily leading to a (near-)optimal solution.

C. Network QoS

Many approaches, such as [20], [7], deal with point-to-point network QoS, but they do not consider services and compositions from SOC. One of the few examples that combines this with SOC is [21] which looks at service compositions in cloud computing. The difference is that instead of the normal composition problem a scheduling problem is solved where services can be deployed on virtual machines at will. Also no QoS algorithm is given, so it is unclear, if that approach can compute input-dependent QoS and network transfer times.

D. 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 goal is to achieve a (near-)optimal scheduling minimizing the execution time, which is often achieved by greedy heuristic approaches, like HEFT [19]. 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, as we optimize multiple QoS properties under given QoS constraints.

III. APPROACH

In this section we define our approach. First, we present our proposed architecture. Based on that, we explain and motivate our workflow model. Then, we define our execution policy. Finally, we describe our algorithmic framework to compute the QoS of a workflow.

A. Distributed Service Architecture

We want to minimize communication costs with a distributed middleware that could be deployed as in Fig. 4.

While not requiring services to communicate directly with each other, we can still save network costs by delegating to call services to a part of our middleware $M_{local}$ which is closer to them. The more places exist where we can deploy such a part of our middleware the better, but even just a few places would allow us to reduce the communication cost significantly.

While there are many ways to distribute the middleware, we propose the customized master-slave pattern depicted in Fig. 5 because of its simplicity and robustness. In this architecture, the master control node performs the bulk of
the middleware tasks such as discovery, selection, fault-handling, etc. The only thing that is delegated to the slave control nodes is executing nearby services, and the collection and transmission of their results. Information necessary for fault-handling, monitoring and other tasks is forwarded to the master control node which makes any necessary decisions.

A slave control node basically only has to know which services it needs to execute. Additionally, it waits for necessary data to arrive before the execution, and afterwards sends the obtained results to all the slave and/or the master control nodes as needed. We will give an execution policy which achieves this in Sec. III-C. Of course, the master control node itself can also execute services, e.g. if they are close, or if no closer slave control node can be deployed. Thus, in the case of no slave control nodes, this architecture is equivalent to the standard service architecture. Note that we would assume the number of these network locations to be limited at least in the near future.

B. Workflow Model

We now give the workflow model that provides the basis of our approach. We first introduce our common node concept, before describing our model.

1) Node Concept: While we mostly adhere to common notations, there is one main difference: We consider an executable workflow to consist of nodes, whether be they logical nodes or service nodes. Service nodes represent traditional tasks, and logical nodes represent business logic, such as loops, conditions, etc. As we distribute our middleware, we can choose the network locations of both node types according to available options. Both node types have incoming and outgoing links which inhibit a certain QoS, and, thus, we consider them both equally in our model and in our computations.

2) Model Representation: We support the common workflow patterns of sequences, parallel invocations (alternative, parallel, etc.) and loops, as in [6]. Logical nodes must be explicit in the model, as they have to be distributed, as well. Model-wise the corresponding structure of our workflows is defined as:

\[
W = \begin{cases} 
S & \text{Service Node} \\
L & \text{Logical Node} \\
P(w_1, \ldots, w_n) & \text{Workflow Pattern}
\end{cases}
\]

with \( P \in \{\text{Seq, AND, XOR, OR, Loop}\} \)

For instance, the model of our previous example workflow is depicted in Fig. 6(a). We added explicit fork and join nodes that perform the necessary processing to call A and B in parallel and to join the results afterwards. Also we introduced explicit start and end nodes which both correspond to the master of the middleware in order to compute the correct total network QoS.

**Figure 5: Distributed Architecture**

**Figure 6: Model Representations of an Example Workflow**

Such workflows are commonly given in a hierarchical manner as a tree representation like in Fig. 6(b), e.g. if specified in BPEL [15]. We first have to convert this tree representation to a graph, before we can compute the QoS of the workflow, and add explicit logical nodes. For that purpose we need two helper functions to compute the predecessors and successors of a node in a workflow. The function \( \text{first}(w_f) \), given in the following, computes all atomic nodes of a (sub) workflow \( w_f \) that are executed first (within that workflow). In an analog way, \( \text{last}(w_f) \) computes the nodes that will be executed last.

```
1: procedure \text{FIRST}(w_f)
2: if \( w_f = S\|L \) then
3: \quad return \{w_f\}
4: else if \( w_f = \text{Seq}\|\text{Loop}(w_1, \ldots, w_n) \) then
5: \quad return \text{first}(w_1)
6: else if \( w_f = \text{AND}\|\text{XOR}\|\text{OR}(w_1, \ldots, w_n) \) then
7: \quad return \text{first}(w_1)
8: end if
9: end procedure
```

Using these functions, we can convert a hierarchical workflow into a directed graph with the following \text{mapToGraph} algorithm. The algorithm structurally traverses the hierarchical structure of the workflow in a depth-first manner until it finds an atomic service node which can be connected with its preceding and succeeding service nodes. Once we have converted a workflow into a directed graph we can compute the QoS of the workflow.
necessary rescheduling.

even on a service failure, then the slave reports
structure of the workflow or a part of it, it just executes
are just passed on as needed (e.g. they might never pass
service or evaluate the business logic, and send the result
contains exactly one service node, plus information about
slaves, each slave gets deployed and work packages are
After the master has determined the optimal services and
Fig. 7. The main work is done by the middleware master.
the amount of knowledge and processing required of our
C. Execution Policy

Before computing the QoS of a workflow, we have to define
the execution policy. As mentioned before, we assume that
our middleware is distributed. Our main goal is to minimize
the amount of knowledge and processing required of our
slaves. Thus, we propose the execution policy shown in
Fig. 7. The main work is done by the middleware master.
After the master has determined the optimal services and
slaves, each slave gets deployed and work packages are
distributed to the slaves as in Fig. 7. Each work package
contains exactly one service node, plus information about
preceding and succeeding service nodes.

Once a slave has received all results from the preceding
service nodes, it can execute its service node(s), e.g. call the
service or evaluate the business logic, and send the result
to all succeeding service nodes. The sending and receiving
is handled by the corresponding slave or master. The final
result is returned to the master, but intermediate results
are just passed on as needed (e.g. they might never pass
through the master). Thus, a slave does not know about the
structure of the workflow or a part of it, it just executes
work packages. If more complex decisions have to be made,
for example on a service failure, then the slave reports
all available information to the master which performs the
necessary rescheduling.

D. QoS Computation

In order to compute the execution duration of a workflow,
we simulate its execution with the simulateExecution algo-
rithm introduced in the following. Note that the commonly
used aggregation is not sufficient to compute this, as we will
illustrate later. For each service node of the workflow, we
keep track of how many preceding service nodes still need
to be executed (line 4). Then, we execute ready nodes (line
6) until no nodes are left. We also keep track of the time
when the execution of a node has started and finished. After
a service node is executed, we evaluate its QoS (line 8), e.g.
according to its SLA, and then virtually pass its result to all
succeeding nodes (line 10).

Figure 7: Distributed Workflow Execution

1: procedure MAPTOGRAPH($f, w_f, ls, g$)
2:    if $w_f$ = $S||L$ then
3:        $\{v_f \in f . add$ edge $(f \rightarrow w_f)$ to $g\}$
4:    else if $w_f$ = $Seq||Loop(w)$ then
5:        $\{v_l \in ls . add$ edge $(w_f \rightarrow l)$ to $g\}$
6:    else if $w_f$ = $Seq||Loop(w)$ then
7:        $\{\forall v \in f . add$ Graph $(f, w_f, ls, g)$
8:    $h = w_1, t = w_2, ..., w_n$
9:    $\text{mapToGraph}(f, h, first(Seq/Loop(t)), g)$
10: $\text{mapToGraph}(last(h), Seq/Loop(t), ls, g)$
11: else if $w_f$ = $AND||XOR||OR(w_1, ..., w_n)$ then
12:    $\{\forall v \in \{1..n\} . \text{mapToGraph}(f, w_1, ls, g)\}$
13: end if
14: end procedure

1: procedure SIMULATEEXECUTION($g$)
2: for each vertex $v \in g$ do
3:    $v$.execStart = 0
4:    $v$.reqIn = $|v.incoming|
5: end for
6: while $\exists$ unvisited $v \in g . v$.reqIn = 0 do
7:    visit any unvisited $v \in g . v$.reqIn = 0
8:    evaluateQoS($v$)
9:    $v$.execEnd = $v$.execStart + $v$.qos.runtime
10: for each $w \in v$.outgoing do
11:    $cv = v$.controlNode
12:    $cw = w$.controlNode
13:    $cNet = getNetworkQoS(cv, cw)$
14:    $vNet = getNetworkQoS(v, cv)$
15:    $wNet = getNetworkQoS(w, cw)$
16:    $\text{trans} = \sum_{net \in \{cNet, vNet, wNet\}} \frac{v.resultSize}{net.transRate}$
17:    $\text{delay} = \sum_{net \in \{cNet, vNet, wNet\}} net.delay$
18: end for
19: end while
20: $v$.execStart = $\max\{w$.execStart, end$\}
21: $v$.reqIn = 1
22: end procedure

When passing the result we consider the delay and the dura-
tion of the data transfer between the two service nodes (line
18). For instance if a service node $v$ needs to communicate
with a service node $w$, then first $v$ communicates with its
control node $cv$, $cv$ communicates with $w$'s control node
cw, and finally $cw$ communicates with $w$. Accordingly,
we compute both the network transfer time (line 16) and the
network delay (line 17). Note that in order to estimate the
data transfer we either need some SLA specifying that, or
some historical data.

If we annotate our previous workflow example with execu-
tion durations of the services, and network delays, as
in Fig. 8 our algorithm will produce the values of Fig. 9 for the execution times of the nodes (start/end). This simple example shows that hierarchical QoS aggregation would not work, because first A and B would be aggregated together. This would make it impossible to compute the correct network QoS, as the maximum of the delay of incoming and outgoing nodes of (A,B) each would be aggregated as 20. But actually there exists no path that can go through both the incoming and the outgoing nodes of (A,B) with delay 20, so that the aggregated value would be too high.

Thus, we compute the QoS with a two phased algorithm as in [9]. In the first phase, we simulate the execution of the workflow with simulateExecution based on the graph we obtain by applying mapToGraph. In the second phase, we take the obtained QoS for each node and aggregate it in a hierarchical manner according to the commonly used aggregation rules that take our workflow patterns into account, as in [6], [22]. Just for the runtime of the workflow, we keep the computation from the first phase, as we cannot compute it with a hierarchical aggregation, like argued previously.

IV. Evaluation

In this section we evaluate our approach. First, we describe the setup of our evaluation. Then, we evaluate the benefits of our architecture. Finally, we show that our architecture also scales in regard to the problem size.

A. Setup

The evaluation was run on a machine with 32 AMD Opteron cores with 2.4 GHz. All algorithm instances were evaluated in separated threads and granted a single exclusive core, while memory was shared and less than 1 GB per instance was needed. Note that the following evaluations settings are based on our previous evaluations in [9].

We generated 100,000 unique network locations. The workflows were generated with random tasks and control structures. For each task, we randomly chose a number of network locations and created services there. Figure 10 depicts an example of a generated workflow of length 10. We chose the following algorithms which all use our new QoS aggregation algorithm.

- **Dijkstra**, an optimal algorithm for the shortest-path problem.
- **GA**, a standard genetic algorithm with population of size 100.
- **NetGA**, our previous network-aware approach introduced in [9]. The size of the population is 100, as well.

In addition, we evaluated optimal variations of those algorithms, marked with "[o]", which could deploy an unlimited number of control nodes.

B. Latency

By adjusting the number of control nodes, our architecture can adapt from being completely centralized (no control nodes) to being completely distributed (unlimited number of control nodes). Note that, strictly speaking, by control nodes we mean additional (slave) control nodes, as we always have one (master) control node run by the user requesting the workflow’s execution. Figure 11 plots the latency of found service compositions against the number of control nodes, with a fixed workflow size of 40. The control nodes were chosen randomly. We can observe the following two things. First, a completely centralized architecture results in a quite bad latency for the service compositions evaluated. Second, once a sufficient number of control nodes (≥ 32) is deployed, the latencies of the algorithms come reasonably close to their optimal variations. Furthermore, using 1024 control nodes, already results in near optimal results for all algorithms in our experiments. Note that, as shown in [9], our NetGA algorithm is better in approximating Dijkstra than the standard genetic algorithm, GA; also Dijkstra is used only for comparison purposes as it cannot be applied to many common service scenarios and as its performance does not scale well in realistic settings.

C. Scalability

Our final evaluation, plotted in Fig. 12, shows how well our architecture scales in term of the problem size. For a fixed number of 1024 control nodes, we can see that the optimality of our NetGA algorithm does not decrease significantly for the workflow sizes (≤ 80) we evaluated. Thus, even for more
complex settings we do not have to increase the number of control nodes. Note that the optimality of $\text{GA}^*$ seems to decrease slightly, but, as mentioned in [9], $\text{GA}^*$ is not very efficient at optimizing the latency in the first place.

V. CONCLUSION

In this paper we motivated the network- and QoS-aware service composition problem which is highly relevant in today’s distributed environments. Then, we introduced a scalable distributed service architecture which significantly reduces network delay and transfer times by eliminating unnecessary communication required in case of a central middleware, as it is common today. We showed that our architecture is a generalization of the standard architecture, being able to adopt from a completely centralized to a completely distributed realization. As it requires no changes on the side of service providers, it guarantees compatibility to existing services and allows for gradual adoption. On top of that, we introduced an extended QoS aggregation algorithm that estimates real-world QoS performance by computing the network QoS for any realization of our architecture. Our algorithm can easily be used to augment current approaches. Finally, we evaluated the benefits of our architecture, showing that it is near-optimal even with a limited number of control nodes.

As future work we want to explore algorithms for choosing good control nodes. In this work we just selected the control nodes randomly, as it did not make a significant difference how we picked those nodes except when just choosing one or two control nodes. We think that in realistic settings there are many factors, such as availability or queuing times, that could effect how useful certain control nodes are. Also minimizing the number of control nodes even further while still obtaining near-optimal results could be critical in certain settings.

VI. ACKNOWLEDGMENTS

We would like to thank Michael Nett and Michael E. Houle at the National Institute of Informatics (Japan) for kindly offering us one of their machines for running our extensive evaluations. Adrian Klein is supported by a Research Fellowship for Young Scientists from the Japan Society for the Promotion of Science.

REFERENCES

[1] 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, 2009, pp. 881–890.

[2] 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, 2010, pp. 11–20.

[3] 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, pp. 1069–1075.

[4] M. Huhns and M. P. Singh, “Research Directions for Service-Oriented Multiagent Systems,” IEEE Internet Computing, vol. 9, no. 6, pp. 65–70, 2005.

[5] M. Jaeger and G. Mühl, “QoS-Based Selection of Services: The Implementation of a Genetic Algorithm,” in KIVS 2007 Workshop: Service-Oriented Architectures and Service-Oriented Computing (SOA/SOC), Bern, Switzerland, 2007, pp. 359–370.

[6] 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, 2004, pp. 149–159.

[7] J. Jin, J. Liang, and K. Nahorst, “Large-scale QoS-Aware Service-Oriented Networking with a Clustering-Based Approach,” in Proceedings of 16th International Conference on Computer Communications and Networks, 2007, pp. 522–528.
[8] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, “Web Services Choreography Description Language (WS-CDL), Version 1.0,” http://www.w3.org/TR/ws-cdl-10/, 2005.

[9] A. Klein, F. Ishikawa, and S. Honiden, “Towards Network-aware Service Composition in the Cloud,” in Proceedings of the 21st International Conference on World Wide Web, ser. WWW ’12. New York, NY, USA: ACM, April 2012, pp. 959–968.

[10] A. Klein, F. Ishikawa, and B. Bauer, “A Probabilistic Approach to Service Selection with Conditional Contracts and Usage Patterns,” in Service-Oriented Computing, ser. Lecture Notes in Computer Science, L. Buresi, C.-H. Chi, and J. Suzuki, Eds. Springer Berlin Heidelberg, November 2009, vol. 5900, pp. 253–268.

[11] A. Klein, F. Ishikawa, and S. Honiden, “Efficient Heuristic Approach with Improved Time Complexity for QoS-Aware Service Composition,” in IEEE 9th International Conference on Web Services (ICWS), July 2011, pp. 436–443.

[12] B. Klöpper, F. Ishikawa, and S. Honiden, “Service Composition with Pareto-Optimality of Time-Dependent QoS Attributes,” in Service-Oriented Computing, ser. Lecture Notes in Computer Science, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds., vol. 6470. Springer Berlin Heidelberg, 2010, pp. 635–640.

[13] F. Lécué and N. Mehandjiev, “Towards Scalability of Quality Driven Semantic Web Service Composition,” in ICWS 2009. IEEE International Conference on Web Services, July 2009, pp. 469–476.

[14] D. A. Menascé, E. Casalicchio, and V. Dubey, “On Optimal Service Selection in Service Oriented Architectures,” Performance Evaluation, vol. 67, no. 8, pp. 659–675, 2010, special Issue on Software and Performance.

[15] OASIS, “Web Service - Business Process Execution Language (WS-BPEL), Version 2.0,” http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html, 2006.

[16] J. O’Sullivan, D. Edmond, and A. Ter Hofstede, “What’s in a Service?” Distributed and Parallel Databases, vol. 12, no. 2–3, pp. 117–133, 2002.

[17] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, “Service-Oriented Computing: A Research Roadmap,” in Service Oriented Computing (SOC), ser. Dagstuhl Seminar Proceedings, 2006.

[18] 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, pp. 97–104, 2010.

[19] H. Topcuoglu, S. Hariri, and M. Wu, “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing,” IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260–274, 2002.

[20] J. Xiao and R. Boutaba, “QoS-Aware Service Composition in Large Scale Multi-Domain Networks,” in IM 2005, 9th IFIP/IEEE International Symposium on Integrated Network Management., May 2005, pp. 397–410.

[21] Z. Ye, X. Zhou, and A. Bouguettaya, “Genetic Algorithm Based QoS-Aware Service Compositions in Cloud Computing,” in Database Systems for Advanced Applications, 2011, pp. 321–334.

[22] 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, vol. 1, no. 1, p. 6, 2007.

[23] 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, pp. 411–421.

[24] Z. Zheng, Y. Zhang, and M. R. Lyu, “Distributed QoS Evaluation for Real-World Web Services,” in Proceedings of the 2010 IEEE International Conference on Web Services, ser. ICWS ’10. IEEE Computer Society, 2010, pp. 83–90.