2012 ACM/IEEE 13th International Conference on Grid Computing

An Adaptive Heuristic Approach to Service Selection Problems in Dynamic Distributed Systems

Peter Paul Beran*, Elisabeth Vinek†, Erich Schikuta‡ and Maria Leitner‡
*Workflow Systems and Technology, University of Vienna, Austria, Email: peter.beran@univie.ac.at
†CERN, Genève, Switzerland, Email: elisabeth.vinek@cern.ch
‡Workflow Systems and Technology, University of Vienna, Austria, Email: {erich.schikuta,maria.leitner}@univie.ac.at

***Abstract*—Quality-of-Service (QoS) aware service selection problems are a crucial issue in both Grids and distributed, service-oriented systems. When several implementations per service exist, one has to be selected for each workflow step. Several heuristics have been proposed, including blackboard and genetic algorithms. Their applicability and performance has already been assessed for static systems. In order to cover real-world scenarios, the approaches are required to deal with dynamics of distributed systems. In this paper, we propose a representation of these dynamic aspects and enhance our algorithms to efficiently capture them. The algorithms are evaluated in terms of scalability and runtime performance, taking into account their adaptability to system changes. By combining both algorithms, we envision a global approach to QoS-aware service selection applicable to static and dynamic systems. We prove our hypothesis by deploying the algorithms in a Cloud environment (Google App Engine) that allows to simulate and evaluate different system configurations.**

***Keywords*-Quality of Service, Service Selection, Genetic Algorithm, Blackboard.**

I. Introduction

In distributed, service-oriented systems, it is a crucial task to efficiently select and compose services required to respond to a given request. In fact, in many modern, mainly web-based systems, several instances of an abstract service coexist and compete. While their exposed functionality is identical, these instances differ in non-functional attributes such as performance measures, availability and reliability. When constructing a concrete workflow that is executed as response to a request, each service of the abstract workflow has to be instantiated with what is in the following called a *deployment*. This has to be done in a way that is optimizing (seeking for a minimum or a maximum) a custom utility function – the exact objective functions depend on the specific problem. Mathematically, this can be mapped to a multi-dimension multi-choice knapsack problem – when only considering attributes of the deployments – or to a shortest-path problem, when also considering attributes of the links between two deployments. Several heuristics have been proposed to solve these QoS-aware service selection problems known as NP-hard.

In our previous work [1], [2], we studied the applicability of blackboard and genetic algorithms to solve QoS-aware service selection problems and compared them in terms of runtime, scalability and quality of the solutions. We showed that the blackboard approach outperforms the genetic algorithm for small problem spaces, while the situation is reversed for bigger problem spaces, although generally the blackboard reaches solutions of slightly better overall quality. We also proposed parallel versions of both algorithms to improve the overall runtime performance for big problem spaces.

For the simulations that were carried out, we assumed a static environment, consisting of a number of abstract services with respective deployments implementing them, and performance attributes for all these deployments as well as for the network links between them. However, real distributed systems are not usually static in nature. They are rather dynamic in various aspects. Deployments can be added or removed from the system, and non-functional attributes can change quite often. A typical example for such a highly dynamic attribute is the server load. Approaches for solving such problems should thus be able to adapt to these dynamics.

In this paper, we propose how dynamic aspects of a system can be captured, described and analyzed. Based on this and on more detailed findings regarding performance and scalability of the blackboard and genetic algorithm, we propose a hybrid approach that takes advantage of the strength of both approaches. The vision is to provide a global and adaptive approach to QoS-aware service selection problems, not tailored to only a specific application domain, but capable of capturing system behavior and reacting to it. To carry out the evaluations, the Cloud-based optimization framework on the Google App Engine [3], introduced in [2], has been extended. To motivate the hybrid approach, we take advantage of having two very different application environments. One is the Datagrid based distributed metadata system from the ATLAS Experiment [4], a particle detector experiment constructed at the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN). The second one is a highly dynamic University information system, consisting of a series of distributed, heterogeneous applications [5].

The remainder of this paper is organized as follows.

1550-5510/12 $26.00 © 2012 IEEE
DOI 10.1109/Grid.2012.26

Section II discusses related work in the area of QoS-aware service selection problems, emphasizing approaches using blackboard and genetic algorithms. In Section III, the motivation for our work is emphasized, and two concrete application scenarios are described and compared. Section IV introduces a way of representing the system status and capturing dynamics. A blackboard and genetic algorithm are introduced in Section V and evaluation results for different system setups are discussed. Based on these findings, a novel hybrid and adaptive approach for service selection problems is proposed in Section VI. Finally, Section VII concludes the work and presents an outlook on future research directions.

II. Related Work

Several heuristic approaches have been presented and evaluated for solving QoS-aware service selection problems. Recently, several authors have applied different genetic algorithms for solving those NP-hard problems [6], [7], [8]. Both single- and multiple-objective algorithms have been discussed. Few comparative studies exist, evaluating a genetic algorithm against other approaches. An example is the work by Jaeger and Mühl [7]. The authors compare a genetic algorithm in terms of QoS ratio, performance and scalability to three other algorithms, one exhaustive search and two heuristics (branch-and-bound and local hill-climbing). The results show that the genetic algorithm outperforms the other two in terms of performance and scalability, but no further conclusions are made. In [9], several heuristic approaches for the selection of web services are evaluated.

Only few publications address the application of blackboards to the QoS-aware service selection problem. In Wanek et al. [10] and Schikuta et al. [11] the authors use blackboards for optimizing Grid workflows regarding dynamically changing resources and conditions. Lepouras et al. [12] present an active ontology-based blackboard architecture for the discovery of Web services. In Nolle et al. [13] a distributed algorithmic and rule-based blackboard system (DARBS) is presented, where knowledge sources are implemented as parallel processes. The blackboard itself comprises a centralized database storing the acquired knowledge and making use of a multitude of agents.

To the best of our knowledge, no comparative study or hybrid approach combining genetic and blackboard algorithms has been proposed until now. However, other hybrid approaches have been proposed for the QoS-aware service selection problem, mainly hybrid evolutionary algorithms. Tang and Ai [14] propose a hybrid genetic algorithm based on a local optimizer that improves the fitness value reached and is capable of handling a large number of constraints. The focus in this work is mainly on improving the fitness value – the computation time of the presented hybrid algorithm is slightly higher than of "standard" genetic algorithms. This approach is different from ours because it merges two approaches into a single algorithm, whereas we propose a framework capable of "choosing" the best algorithm for a given problem. Canfora et al. [6] compare a genetic algorithm to integer programming to solve QoS-aware service selection problems. The results show that integer programming outperforms the genetic algorithm for a small number of concrete services. For workflows comprising more services, the genetic algorithm keeps its performance almost constant, whereas the time for the integer programming grows exponentially. The authors conclude that both algorithms have their applicability in different scenarios, but they do not propose an approach combining both algorithms.

III. Motivational Examples

Data- and resource intensive experiments, such as High-Energy Physics (HEP) experiments, require large scale computing capabilities for various applications. In modern web-based service-oriented systems, appropriate scaling can be achieved by distributing services and data [15], [16], thus distributing the requests and load. Batch processing and analysis jobs in HEP experiments mainly run and rely on Grid infrastructures. An example for such an application, building the motivation for our work, is the ATLAS TAG system. ATLAS is a detector of the Large Hadron Collider (LHC) hosted at CERN, the European Organization for Nuclear Research. The overall goal is to discover new physics by analyzing a massive amount of data from proton-proton collisions [4]. The TAG system is composed of the actual metadata, residing in relational databases, and services that can be used to access the data, make queries and extract data in a format suited for further physics analysis. The data as well as the services are distributed worldwide. In such a case a well-defined mechanism to direct requests to appropriate data sources and (web) services is needed.

While the above described application is the main motivational scenario, we are seeking to adopt our approach to other service selection challenges. In fact, by proposing a hybrid approach, the goal is to cover application scenarios presenting different characteristics. To this end, we are applying our study to a second use case, taken from a University information system. UCETIS, which stands for University Cross European Transfer of Information System [5], has been initiated at the University of Vienna. Many European universities have a heterogeneously grown IT-infrastructure. Requirements arise both from legal aspects and from the organizational structure that typically differs a lot from institution to institution. It is thus a challenge to seamlessly access this distributed data, and join over several sources in order to aggregate relevant information.

In order to allow for optimization, the building blocks of the systems have to be known, described and monitored. The modeling and monitoring of the TAG system is described in detail in [17]. In the UCETIS use case, it is more difficult to gather detailed information about the involved components.

In general, the two application domains differ in several aspects. The TAG system is centrally controlled, i.e. there is a central management unit aware of all the system components and the changes affecting them. These changes (such as added or removed deployments) do not happen often and are in general known well in advance, allowing for a coordinated commissioning – or decommissioning – of the system and its controls. Additionally, the components are tightly coupled, i.e. they are explicitly implemented to be compatible. In the UCETIS use case on the other hand, there is no central control instance disposing of the full system information, the environment is more dynamic and due to the absence of a central control changes are less predictable. Moreover, each deployment in UCETS has little or no knowledge about the other deployments, i.e. the components are loosely coupled. Consequently, in the TAG use case we dispose of complete information, taken from system monitoring and logging, whereas in the UCETIS use case we are confronted with incomplete information regarding the system statistics.

IV. System Model

A. System Components

The main building blocks of the considered system are *Services*, *Deployments*, *Links* and *Attributes*, as shown in form of a database schema in Figure 1.

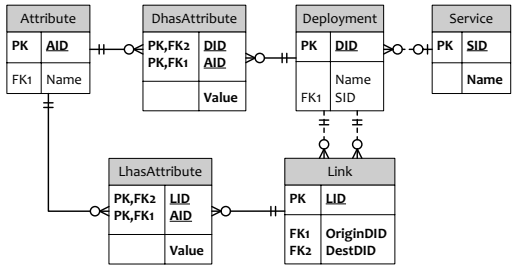

Figure 1. Database Schema of System Components

A **Service** is an abstract component with a certain functionality. It is often referred to as *abstract service*. Formally, a service $S_i, i \in \{1, ..., n\}$ is defined by a unique identifier: $S_i = id(S_i)$. In our model, a service does not have attributes, because we are only considering non-functional attributes. $S = S_1 \cup ... \cup S_n$ is the union of all services and $n = |S|$ is the number of services in the considered system.

A **Deployment** is an instantiation of a service, often referred to as *concrete service*. It implements the functionality of a service and is characterized by its non-functional attributes. Formally, a deployment $D_i, i \in \{1, ..., m\}$ is defined as:

$$D_i = \{id(S_j), \{id(D_i), [q_{i1}, ..., q_{ik}]\}\} \quad (1)$$

where q_{i_k} are QoS attributes associated with the deployments. As defined in Equation 1, the service that a deployment is associated with is part of its definition. $D = D_1 \cup ... \cup D_m$ is the union of all deployments, where $m = |D|$ is the total number of deployments in the considered system.

A **Link** is a connection between two deployments. Formally, a link $L_i, i \in \{1, ..., n\}$ is defined as:

$$L_i = \{id(D_j), id(D_k), [q_{i1}, ..., q_{ik}]\} \quad (2)$$

where $id(D_i)$ and $id(D_j)$ refer to two deployments linked by Link L_i. $L = L_1 \cup ... \cup L_n$ is the union of all links, and $|L| = m \times (m-1)/2$ is the number of links in our system, assuming that no self-links are considered.

An **Attribute** is defined as: $q_i = \{id(q_i), Dom_i, a_i\}$ where $id(q_i)$ is a function associating a unique identifier to the attribute, Dom_i is the *Definition Domain* and a_i is the value taken by the attribute.

B. Dynamic System Aspects

A service-oriented system as described in the previous subsection is not static in nature. In particular, it can be dynamic regarding system configuration and component status (attributes) aspects. Considering the *System Configuration*, deployments of specific services can enter or quit the scenes (by registering respectively unregistering at a central registry). Additionally, scheduled or unscheduled downtimes of underlying resources can cause deployments to be down and thus not reachable. A certain system configuration consisting of specific (active) deployments for each service is thus only valid for a certain period in time. Regarding the *Component Status* in real-world distributed systems, attributes of deployments and links are most likely subject to frequent changes. Furthermore, also attributes that build on the analysis of historical data – such as the availability – can evolve over time. A specific attribute value of a component is thus also only valid for a certain period in time.

For a given point t in time, we define the *system status* or *system blueprint* as the union of the available deployments and links, with their respective attribute values: $S(t) = D \cup L$.

As stated above, the attributes of deployments and links are subject to changes, even at a much higher frequency than the deployments themselves. As it is unmanageable to record every small change in an attribute value, we consider levels for the values of our aggregated performance measures. A *level* V_a of the attribute value a is defined as a range of values of attribute a:

$$V_{a_i} = [v_{a_{i0}}, v_{a_{i1}}[= \{x \in \mathbb{R} | v_{a_{i0}} \leq x < v_{a_{i1}}\} \quad (3)$$

The following operations are defined and feasible on $S(t)$ (details are described in Section VI):

- Addition of a deployment: $S(t+1) + D_i = S(t) \cup D_i$
- Subtraction of a deployment: $S(t+1) + D_i = S(t) \setminus D_i$

- Attribute level change:
  if $a(t) \in V_{a_i}$ and $a(t+1) < v_{a_{i0}} \rightarrow a(t+1) \in v_{a_{i1-1}}$
  if $a(t) \in V_{a_i}$ and $a(t+1) \geq v_{a_{i1}} \rightarrow a(t+1) \in v_{a_{i1+1}}$

Links are implicitly added and subtracted, as they represent a connection between two deployments. If services, resources or sites are added or removed, this can be mapped to adding or removing deployments attached to the respective component.

V. Algorithmic Approaches

A. Algorithms

Two heuristics – a genetic and a blackboard algorithm – are proposed for approaching the above mentioned challenges. Additionally, a random walk (exhaustive search) is used as baseline for assessing the quality of the solutions achieved with the heuristics. In general, the input to the algorithms is an abstract workflow, i.e. an ordered list of services (s_1, s_2, …, s_n). They all use a common knowledge base regarding existing deployments, links and attributes. The output of the optimization process is an ordered list of deployments (d_1, d_2, …, d_n) corresponding to a concrete workflow feasible for the abstract input workflow.

1) Random Walk: The random walk is an exhaustive search algorithm evaluating all possible paths in a graph. Applied to the concrete scenario, the random walk evaluates all possible deployment combinations corresponding to the input workflow and returns the solution with the highest quality, e.g. with the maximum performance indexes. With an increasing problem space, the runtime performance of the random walk grows exponentially. In this regard it cannot compete with heuristic approaches, but as it is an exact search it has been used to determine the best solution as benchmark. The random walk is listed in Algorithm 1.

```
D(1) = findInitial(base=s_1);
foreach d_1 in D(1) do
    assignQuality(d_1);
end
for pos = 1 to n do
    foreach d_pos in D(pos) do
        D(pos + 1) += findSuccessors(origin=d_pos, base=s_{pos+1});
        foreach d_{pos+1} in D(pos + 1) do
            assignQuality(d_{pos+1});
        end
    end
end
solution = maxQuality(d_n in D(n));
```

Algorithm 1: Random Walk Algorithm

2) Genetic Algorithm: A **chromosome** is represented as an ordered list of deployment names or references. Each chromosome corresponds to a possible solution. The **population** is a set of chromosomes at a given **generation**, i.e. iteration of the algorithm. The maximum number of generations is an input parameter to the algorithm. The **fitness** is computed based on available and relevant QoS attributes. The exact function for computing the fitness of individuals is dependent on the considered problem.

The genetic algorithm is listed in Algorithm 2 and its steps are briefly explained in the following:

- **Initialization** of a random population with individuals that are chosen randomly out of all possible ones. An individual corresponds to a concrete workflow. The size of the population is an input parameter to the algorithm.
- **Evaluation** of the fitness – or quality – of each individual in the population, computed according to the custom fitness function.
- **Selection** of individuals for the next generation. In our implementation, a classical roulette wheel selection [18] has been chosen for the presented algorithm. In this strategy, the survival probability of a chromosome is proportional to its fitness, i.e. each chromosome is copied n times, where n is the fitness value casted to an integer. Then, a random selection is performed, giving the fitter individuals a higher chance to be selected for the next generation.
- **Crossover** corresponds to an operation in which two chromosomes called the parents are combined and breed two children. Concretely, the input abstract workflow is split at a certain position, the first part is taken from one parent, and the tail from the other one. Crossover only happens at a given probability that is set as an input parameter to the algorithm.
- **Mutation** refers to an operation that produces spontaneous random changes in chromosomes, i.e. a single gene is replaced by another one, with a certain probability that also has to be set as an input parameter.

```
Pop(0) = initializePopulationRandomly(S);
for t = 0 to generations do
    foreach chromosome in Pop(t) do
        evaluate(Pop(t));
    end
    Pop'(t) = rouletteWheelSelection(Pop(t));
    crossover(Pop'(t), crossover_rate);
    mutate(Pop'(t), mutation_rate);
    Pop(t + 1) = Pop'(t);
end
```

Algorithm 2: Genetic Algorithm

In additional to the sequential genetic algorithm shown in Algorithm 2, a parallel version has been implemented and tested, as a way of exploring tuning possibilities. Several parallelization techniques exist for genetic algorithms, as described and classified in [19] and [20]. The tested parallel genetic algorithm implements a *distributed fitness evaluation*, i.e. the individuals from a population are divided into n groups as many processes as groups are spawned to compute the fitness in parallel. This parallelization is also referred to as *global parallelization* or *master-slave model* [19].

3) Blackboard: A blackboard – initially developed in the area of artificial intelligence [21] – implements an A^*-algorithm to heuristically solve NP-hard problems. It is especially suited for complex problems with incomplete

knowledge and uncertainties regarding the attributes and the behavior of the involved components.

In general, a blackboard consists of the three components. A **global blackboard** represents a shared information space containing input data and partial solutions. In our implementation, the information consists of a list of deployments, links and their attributes (performance indexes) stored in a data store. A **knowledge base** is composed of several independent regions, each of which is owned by a single expert disposing of specific knowledge. The global blackboard acts as a "mediator", allowing the different regions to communicate. In our implementation, the regions are the deployments that have to be selected for each abstract service in the input workflow. The **control component** defines the course of activities (phases) required to perform the optimization problem.

The blackboard mechanism is listed in Algorithm 3. A cost-based decision tree is generated based on cost estimations for the visited paths. As shown in Algorithm 3, the expansion of promising deployments for a step in the workflow is handled by an `OpenList` and `BlockedList`. The `OpenList` contains all list of possible steps (deployments to choose). Each of these steps is rated by applying a cost function that sums up the costs of past decisions and the cost of the next step. Considering this total cost value the cheapest step (deployment) is chosen for the next step in the optimization approach. The `BlockedList` contains steps (deployments) that do not fulfill the given requirements and therefore must be excluded from the set of possible solutions.

```
OpenList = expand(s_1); BlockedList = [];
while OpenList ≠ [] do
    Act = best(OpenList); OpenList \= Act;
    if Act == Goal then
        return Act;
    end
    foreach d_x in expand(Act) do
        d_x.costs = Act.costs + h(d_x.costs);
        if d_x ∉ OpenList ∧ d_x ∉ BlockedList then
            OpenList += d_x;
        else
            if d_x.costs ¡ OpenList[d_x.id].costs then
                OpenList[d_x.id] = d_x;
            end
        end
    end
end
```

Algorithm 3: Blackboard Algorithm

As for the genetic algorithms, a parallel version of the blackboard has been implemented and tested. As first proof-of concept, this has been realized using intra-parallelism, i.e. parallel execution of the expansion phases.

B. Simulation Environment

We initially implemented the algorithms in Python and run them sequentially on local machines. For each execution the required time and the optimization quality of the obtained solution have been measured. To tune the algorithms and reduce the overall execution time, parallelization possibilities have been explored. To allow for parallelization in Python several possibilities exist, such as using threads or leveraging a specific python library (e.g. parallel python or MPI for python). However, moving the implementation to a Cloud infrastructure allows us to take advantage of powerful existing infrastructures and development environments, using the Cloud PaaS (Platform-as-a-Service) level. As a major requirement this platform must natively support Python and allow for parallel execution. After an investigation of existing platforms, we chose the Google App Engine (GAE) as basis to implement the algorithms on. The GAE fulfils our requirements, is well documented and provides a sufficient amount of free quota for CPU time, API calls and storage capicity. Moreover, in a later extension of our experiments – requiring a higher number of resources – we will be able to use them on a pay-per-use base.

As depicted in Figure 2 we make use of the following components and APIs of the GAE.

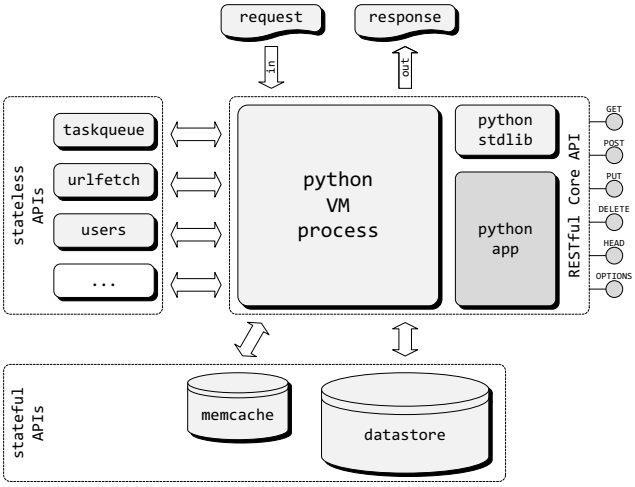

Figure 2. Google App Engine Architecture

In the **RESTful Core API**, each algorithm implementation has to inherit from either `AlgorithmSequential` (in case of a sequential implementation) or `AlgorithmParallel` (in case of a parallel implementation). As depicted in Figure 3 both classes are subclasses of the `AlgorithmHandler` which required the implementation of two abstract methods (`doGet`, `doPost`). Moreover, some utility methods exist that encapsulate some often used functionality and allow for a more convenient code parallelization (see stateless API). The `enqueue*` methods allow to either spawn a new task, or perform an asynchronous urlfetch command. The `dequeue*` methods instead allow to collect the results of either the called task or the asynchronously fetched site.

For our algorithm implementation the system state of a simulated system environment is stored within the **Stateful API** datastore of the GAE, in terms of components (services, deployments and links) and their attributes (e.g. a randomly generated performance index). One the other hand the memcache API allows to store objects for a certain amount of time and therefore can be seen as a caching machanism. In our approach we heavily make use of the memcache for our parallel algorithm implementations to allow them for an easy data exchange between different tasks (see taskqueue API) or between the master task and its worker tasks (see urlfetch API).

Among various existing **Stateless APIs** for Python we only make use of three fundamental ones, the users API, the taskqueue API and the urlfetch API. The users API allows us to distinguish between casual users and administrators within our simulation testbed site. The taskqueue API and urlfetch API are used to enable for parallelism, because due to some security and scalability reasons the Python interpreter runs within a "sandbox" in the GAE and therefore does not allow for processes, threads, or socket connections. The taskqueue API allows to create new tasks within an ongoing web application request. One limitation of this approach is that the master program looses the control flow, because the tasks are not able to return values. Therefore, the master program has to track the number and kind of created tasks and the tasks have to conclude by (re)starting the caller. However, this is not proper to the GAE, other Cloud providers also make use of task-queue-like systems, such as Amazon SQS or Azure Queue. Using the urlfetch API instead, the master program does not loose the flow of control and can wait for results of its workers. However, due to a strict time limit of ten seconds for each urlfetch call only very small parallel programs can be realized.

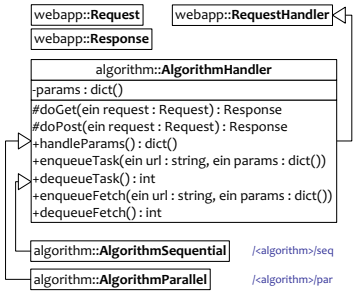

Figure 3. Algorithm Interface Classes

C. Evaluation and Discussion

A series of benchmarks tests have been carried out based on the setup described above. The sequential and parallel versions of the blackboard, genetic and random walk algorithm were run on different datasets, i.e. on an increasing number of deployments, resulting in an increasing number of choices. For each run, benchmarks have been stored in the datastore. A benchmark entry consists of the algorithm and mode (sequential/parallel) producing the benchmark, a timestamp, the system configuration (deployments per service), the total time, the reached quality and the achieved solution. The corresponding data model is depicted in Figure 7. The results are shown in Figures 4, 5 and 6 and the main findings are summarized in the following.

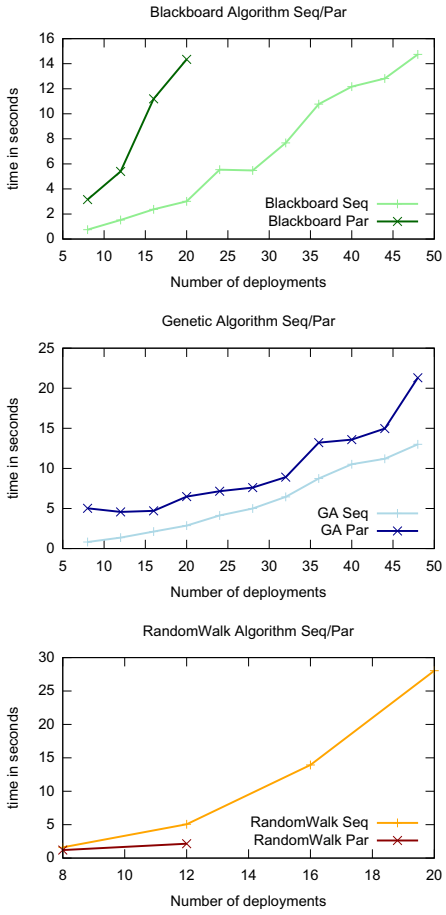

Figure 4. Runtime Scalability per Algorithm

The reason why only small setups could be benchmarked using the random walk is a time limitation per requests in the GAE. Each request is running into a `DeadlineExceeded` error if it does not respond within 30 seconds, which are reached when considering 20 deployments.

In terms of runtime performance and scalability, in general the genetic algorithm outperforms the blackboard, except for a small number of deployments. This is in principle compliant with the results from our previous work [1]. However, it has to be noted that in [1] the threshold was at about 100 deployments, whereas in the present case it is much lower. In the current setup, benchmark runs are being

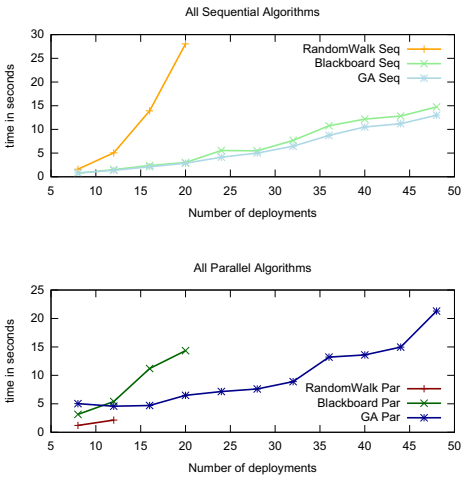

Figure 5. Runtime Scalability Comparison

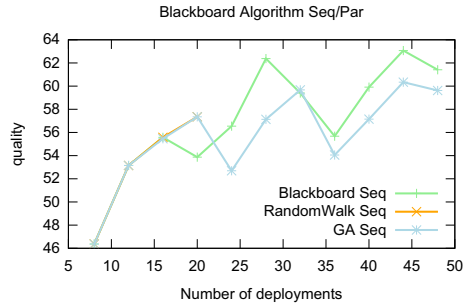

Figure 6. Quality (Fitness) Comparison

made with systems up to only about 50 deployments, due to the urlfetch (10 seconds) and taskqueue (30 seconds) time limitations of the Google App Engine. To completely verify previous results, tests with bigger setups would have to be made.

In general, the blackboard outperforms the genetic algorithm regarding the quality (fitness) of the solution found. As shown in Figure 6, differences in the quality reached start appearing for system setups with more than 20 deployments in total. Changing the input parameters of the genetic algorithm (e.g. choosing a bigger population size and a higher number of generations) would improve the fitness reached, but would affect the runtime performance. The parameters have been chosen in a way so that the fitness achieved with the genetic algorithm is not less then 90% of the optimum.

As shown in Figure 4, the parallelization of the algorithms worsens their runtime instead of improving it, which we consider as an implementation artifact. Neither the *urlfetch* nor the *taskqueue* version are efficient for such small problems, i.e. the overhead added by those mechanisms does not stand in relation to any computational gain. It is to be mentioned that the chosen environment (the Google App Engine) has not been designed for high-performance applications and computations; but for interactive, short request handling. We were thus not expecting real high-performance parallelization, but consider the implementations as a proof-of-concept. An exception is the random walk, where the parallel version outperforms the sequential one, but is feasible only in very small setups. Again this is due to the hard time restrictions for requests from a taskqueue or urlfetch call.

VI. Hybrid Algorithm for Service Selection

As can be seen from the evaluation results presented in Section V, there is a trade-off between execution time and attained quality of the optimization results, e.g. the concrete workflow of deployments. We thus envision a hybrid approach consisting of a combination of the blackboard and genetic algorithm. We extend the optimization framework with a caching mechanism that allows to reuse workflows of previously computed optimizations whenever a system environment has not or only slightly changed. In the future, we plan to also evaluate and integrate other algorithms into the approach.

A. Motivation

The basic idea behind the hybrid approach is to take advantage of the strengths of both participating algorithms in terms of runtime and achieved quality of the result, in their respective value domain. To decide then to apply which algorithm, a mechanism has to be established that keeps track of already computed solutions and the system state when the solution was created. A solution (workflow of concrete deployments) might fit well in the future and can therefore be cached, especially when the system environment – in terms of services, deployments and their interconnection – stays the same or changes only slightly. In a stable system environment optimization results are reusable and therefore we define this as a “cache hit”. On the opposite, in unstable environments, when system components change frequently, a previously computed optimization has to be repeated, we call this a “cache miss”. In the two mentioned extreme cases (cache hit and miss) the decision from the algorithmic point of view is very clear. However, in some cases in between – where only some deployments of the solution have changed – we will be able to reuse a fraction of a previously achieved optimization solution and use it as a starting point for e.g. the seed population of a genetic algorithm. Therefore a key issue is to determine the degree of similarity between a past and actual system state. How a system state is defined is outlined in Section IV. The following subsections detail how such system states can be stored and compared. Based on the obtained *similarity index* we are able to make reasonable decision regarding which optimization algorithm to apply.

B. System State Preservation

At first it has to be clarified how a system states can be stored and compared. As pointed out in Section IV, a

system is described by its components (services, deployments, links) and their attributes, valid at a given point or period in time. For preserving a computed solution and the corresponding system state, we store a map of components $(d_1...d_n; l_1...l_{n-1})$, in case of a sequential workflow, where two deployments are connected via a link. Each component itself has to store a map of the attributes that have been considered in the optimization approach $(q_1, ..., q_k)$.

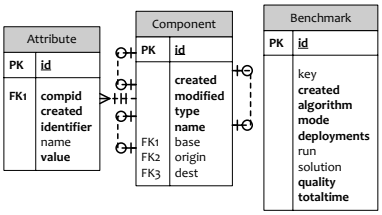

Figure 7. Google App Engine Datastore Model

In Figure 7 the denormalized version of the database schema of system components (Figure. 1) is presented. It allows to efficiently query the GAE datastore. It consists of a *Component* store and an *Attribute* store. A component has a unique id, a created and modified timestamp, a type (service, deployment, link), a name, a base (if deployment points to the corresponding service) and a origin and destination attribute (for qualifying a link). An attribute has a unique id, a created timestamp, an identifier, a name, and a value and is related to a specific component (via the compid). This notion enriches the model of a system environment with time-relevant information that is mandatory for the similarity reasoning process, and provides information about how much a system state changes. The *Benchmark* table stores all information about optimization runs and is used to perform the evaluation.

Especially the attributes *created* and *modified* are important, as they document the life-cycle of a component and/or its attributes. Thus, events that occur during the lifetime of a system environment can be captured and expressed as changes in these timestamp attributes. Such events are comparable to traditional CRUD-operations and may include the creation, update or delete of components and/or their attributes.

In order to preserve a system state for a later comparison with a new system state, the following values are required:

1) The *timestamp* indicating when a solution has been computed.
2) The found *solution* that contains the query (chain of services), the chosen deployments, their interconnecting links and all corresponding attributes that were considered in the optimization (e.g. the performance index).

C. Similarity Index Computation

Based on the preserved system state covering the optimal service selection strategy (solution) and its components we have to compute a similarity index that reflects the degree a system state has changed. The obtained *similarity index* provides a foundation for decisions regarding the choice of the most appropriate optimization algorithm to find a "good" or even the optimal solution. A similarity index of `1.0` means that the environment has not changed and so the existing solution can be reused. A similarity index of `0.0` means that the environment and especially the components that build up the previous solution have changed and thus the existing solution cannot be reused.

Since the comparison of system states is not a trivial task, a method has to be developed to rate the similarity of two system states. To solve this issue, a blackboard-based approach can again be chosen. In a very simplistic approach one of the following two expert strategies for such a similarity classification can be used:

1) **Component-level Classification**: In this approach the components (deployments, links) of a given solution are compared with the currently (at the time of the optimization) existing components. The comparison is based on the created and modified timestamps of the components. For example, if a new deployment for a service (that is part of the solution) is available, the created timestamp will be newer than the timestamp of the deployment in the last solution. Therefore, this deployment might not be the optimal one any more. The same applies to links and attributes. The changes of all components and attributes are rated and aggregated to a similarity index. In a simple setup the changes are rated as follows:
   - components are treated equally: the impact (imp) of a single component is as listed in Equation 4, where n is the number of deployments and $n-1$ is the number of links between deployments.
   - changes due to occurred system events are rated according to the relation of affected to total components, as suggested in Equation 4).
   - the overall similarity index is si_{total} as listed in Equation 4.

$$\begin{aligned} imp_C &= \frac{1}{n+(n-1)} \\ si_C &= imp_C * (1 - \frac{\sum D_{newer}}{\sum D_{total}}) \\ si_{total} &= \sum_{i=1}^{2n-1} si_C \end{aligned} \quad (4)$$

2) **Chain-level Classification**: This classification allows to rate a solution according to the deployment patterns

(chains). If a solution is built of four deployments in a chain, we can identify three pairs of deployments, two triples, and one (total) chain of four deployments. When a system change occurs, the longest chain that is not affected by the change can be identified and reused as part of the solution. For example, if a change affects only the last (fourth) deployment of the previous solution, the sequence consisting of the first, second and third deployment can be reused. The similarity index is computed as listed in Equation 5.

$$si_{total} = \frac{chain_{unchanged\_length}}{chain_{total\_length}} \quad (5)$$

With the help of these two techniques it is possible to compare two system states and quantify their similarity.

D. Hybrid Algorithm for Service Selection

The hybrid approach uses the similarity index to decide which optimization strategy is actually best and promises a good and fast solution, depending on the system state and the recent changes. In the best case a solution can be reused, in the worst case the system state has changed radically and the optimization has to be repeated. As depicted in Figure 8, the hybrid approach defines some border values based on the similarity index that lead to different optimization strategies.

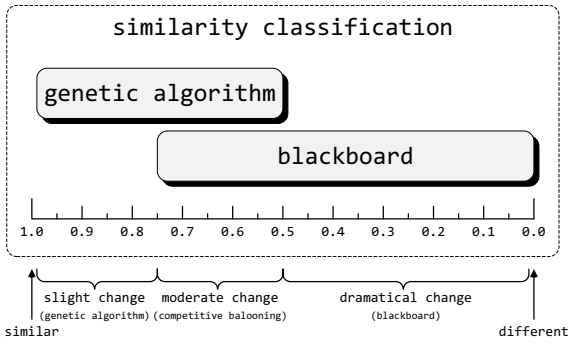

Figure 8. Similarity Classification for the Hybrid Algorithm

- **Similar (1.00)**: The system states match completely and thus the previous solution can be reused. This means that for the observed time frame the system state remains static, without affective changes for the components that build up the solution.
- **Slight Change (0.99-0.75)**: This system state is practically indifferent to the preserved system state. Thus an already computed optimization is good enough and only has to be refined according to the changed components of the solution. Therefore we can apply the genetic algorithm, that has the best runtime performance and can benefit from the previous solution that is put into the initial population.
- **Moderate Change (0.75-0.50)**: For this similarity region it is more difficult to decide which algorithm leads to the best result. Both the genetic algorithm and the blackboard can be used. They can be started in parallel, so that the first algorithm that finishes its execution provides the new solution. This technique is called competitive ballooning and requires enough computing resources. In the genetic algorithm case the initial population is seeded with the given solution. The blackboard makes use of already computed cost values of the unchanged parts of the solution.
- **Radical Change (0.50-0.01)**: This is a drastic change in the system state, leading to a need to restart the whole optimization process, because only minor parts of the solution (in terms of deployments) have not changed since the last computation. Here a blackboard should be applied because it usually delivers a solution with a higher quality rating than the genetic algorithm (according to our simulation runs in Figure 6).
- **Different (0.00)** As in the radical change case the optimization has to be repeated.

VII. Conclusion

In this paper, we proposed a description of the basic, high-level building blocks of a dynamic distributed service-oriented system, including a categorization of the changes that can affect such a system. This foundation allows to apply different algorithmic approaches on a common simulated system setup. To evaluate these algorithms for service selection, a Cloud-based framework on the Google App Engine has been implemented and three algorithms have been deployed, a random walk serving as baseline, a genetic algorithm and a blackboard. Built-in classes allow for extensive benchmarking. Performance indexes on deployments and links have been assumed as aggregated QoS values. The overall goal is to select deployments for a given input workflow in a way to maximize the overall performance index. It has been confirmed that the blackboard slightly outperforms the genetic algorithm for a small number of deployments, whereas the genetic algorithm performs better for large systems, while however reaching a slightly lower overall quality. There is thus a trade-off between runtime performance and achieved results. These characteristics led us to propose a hybrid approach combining the strengths of both algorithms in their respective range. The main idea of this hybrid approach is to implement a caching mechanism, where the system setup is stored and associated with a good result. When the optimization starts again, with the same requested workflow, the first step is to detect and classify the changes that affected the system since the last result has been computed. According to the impact of these changes, it is decided if the old result can be reused, if a genetic algorithm is launched using the previous result in the initial population, or if the blackboard algorithm is run from scratch.

The next step is to complete the implementation of the hybrid approach and evaluate it. This will require a more complex evaluation environment, as real system usage as well as system changes have to be simulated. As a next step, it is further interesting to include more algorithms in our framework, evaluate them, and broaden the hybrid approach to a decision framework capable of launching different algorithms, based on the actual system status.

REFERENCES

[1] E. Vinek, P. P. Beran, and E. Schikuta, "Comparative Study of Genetic and Blackboard Algorithms for Solving QoS-Aware Service Selection Problems," in *HPCS 2011: The 2011 International Conference on High Performance Computing & Simulation*, July 2011, submitted for publication.

[2] P. P. Beran, E. Vinek, and E. Schikuta, "A Distributed Database and Deployment Optimization Framework in the Cloud," in *ICPP 2011: International Conference on Parallel Processing*. IEEE Computer Society, September 2011, submitted for publication.

[3] Google, "Google App Engine," http://code.google.com/appengine/. [Online]. Available: http://code.google.com/appengine/

[4] The ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," in *JINST 3 S08003*, 2008, p. S08003.

[5] E. Schikuta, H. Eich, T. Pitner, P. P. Beran, M. Hoeckner, and B. Vorderegger, "UCETIS - University Cross European Transfer of Information System," http://www.pri.univie.ac.at/workgroups/ucetis/. [Online]. Available: http://www.pri.univie.ac.at/workgroups/ucetis/

[6] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An Approach for QoS-aware Service Composition based on Genetic Algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1069–1075. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068189

[7] M. C. Jaeger and G. Mühl, "QoS-based Selection of Services: The Implementation of a Genetic Algorithm," in *In KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC)*, 2007, pp. 359–370.

[8] S. Yang, "Population-Based Incremental Learning with Memory Scheme for Changing Environments," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 711–718. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068128

[9] M. C. Jaeger and G. Rojec-Goldmann, "SENECA - Simulation of Algorithms for the Selection of Web Services for Compositions," in *TES'05: 6th VLDB Workshop on Technologies for E-Services*, vol. 3811. Springer Press, September 2005, pp. 84–97.

[10] H. Wanek and E. Schikuta, "Using Blackboards to Optimize Grid Workflows with Respect to Quality Constraints," in *Grid and Cooperative Computing Workshops, 2006. GCCW '06. Fifth International Conference on*, October 2006, pp. 290–295.

[11] E. Schikuta, H. Wanek, and I. Ul Haq, "Grid workflow optimization regarding dynamically changing resources and conditions," *Concurr. Comput. : Pract. Exper.*, vol. 20, pp. 1837–1849, October 2008.

[12] G. Lepouras, C. Vassilakis, A. Sotiropoulou, D. Theotokis, and A. Katifori, "An active ontology-based blackboard architecture for Web service interoperability," in *Services Systems and Services Management, 2005. Proceedings of ICSSSM '05. 2005 International Conference on*, vol. 1, June 2005, pp. 573–578.

[13] L. Nolle, K. Wong, and A. Hopgood, "DARBS: A Distributed Blackboard System," in *21st SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, December 2001. [Online]. Available: http://oro.open.ac.uk/5568/

[14] M. Tang and L. Ai, "A Hybrid Genetic Algorithm for the Optimal Constrained Web Service Selection Problem in Web Service Composition," in *WCCI 2010: IEEE World Congress on Computational Intelligence*. IEEE, July 2010, pp. 1–8.

[15] T. Fürle, O. Jorns, E. Schikuta, and H. Wanek, "Meta-ViPIOS: harness distributed I/O resources with ViPIOS," *Iberoamerican Journal of Research "Computing and Systems", Special Issue on Parallel Computing*, vol. 4, no. 2, p. 124–142, 2000. [Online]. Available: www.ejournal.unam.mx/cys/vol04-02/CYS04205.pdf

[16] E. Schikuta and H. Wanek, "Parallel I/O," *International Journal of High Performance Computing Applications*, vol. 15, no. 2, p. 162–168, 2001.

[17] E. Vinek and F. Viegas, "Composing Distributed Services for Selection and Retrieval of Event Data in the ATLAS Experiment," in *Journal of Physics: Conference Series*, October 2010, accepted for publication.

[18] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press, 1992.

[19] M. Nowostawski and R. Poli, "Parallel Genetic Algorithm Taxonomy," in *Proceedings of the Third International*. IEEE, 1999, pp. 88–92.

[20] E. Alba and C. Cotta, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 443–462, 2002.

[21] D. Corkill, "Blackboard Systems," *AI Expert*, vol. 6, no. 9, January 1991.