Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 187–199. http://www.scpe.org
ISSN 1895-1767
© 2012 SCPE

POLICY-BASED SCHEDULING OF CLOUD SERVICES

FARIS NIZAMIC*‡ VIKTORIYA DEGELER *‡ RIX GROENBOOM† AND ALEXANDER LAZOVIK*

Abstract. Worldwide accessibility of clouds brings great benefits by providing easy access to resources. However, scheduling cloud resources for utilization among multiple collaborating cloud users is still often executed manually. To address this problem, we developed a scheduling service for cloud middleware that guarantees optimal resource utilization in terms of a total number of used resources in a given interval based on user-defined policies. In the paper, we introduce the scheduling algorithm, describe its supporting system architecture and provide the evaluation that proves the feasibility of the developed solution. The provided scheduling algorithm takes into account dependencies between individual services, and can enforce common use of shared resources that lead to the optimal resource utilization. By assuring continuous schedule optimality, costs caused by unnecessary usage of additional cloud resources are minimized.

Key words: Cloud Computing; Resource allocation; Optimisation

1. Introduction.

Today, Clouds are used for different purposes and have numerous application areas. Clouds are defined as a large pool of easily accessible virtualized resources (such as hardware, development platforms and/or services) which can then be dynamically reconfigured to adjust to a variable load, allowing for optimum resource utilization [1]. One role of Clouds is to influence development process of software as a service. Currently, software engineering practices are far from ideal, and often the quality of the final product suffers. Complexity of software development process is still very high, and many processes are still manually executed. An example of such process is the scheduling of hardware/software resources for multiple collaborating software engineering teams (e.g., development, system test, user-acceptance test/staging, etc.).

Infrastructures based on Service-Oriented Architecture (SOA) increase the severity of the problem. The flexible setup of SOA systems, and dependencies on other services, make the deployment of these systems complex in staged environments. Virtualization can help to reduce the cost of physical hardware, and simulation of application behaviour can reduce the dependency on back-end systems and external services [5]. These simulators can be deployed using cloud infrastructures to create a flexible platform to support the software development and test teams. However, to fully exploit the benefits of virtualization and virtualized services, one still needs to carefully manage dependencies between different parts of the system to ensure that all services are in place, resources that are not required are not launched (or being turned down) for optimal resource usage. Without automated scheduling this task is very complex and error prone, and potential savings cannot be achieved.

Currently, practice shows that in many large companies that are working with sensitive data, scheduling of resources is done manually. Nearly half of hundred of surveyed IT executives on Cloud technologies use manual processes to handle moves and changes in their infrastructure solution for the cloud [3]. For example, some companies perform their resource allocation and scheduling using collaborative tools, while in others, more primitive (in terms of automatization support) methods are used. Furthermore, requests coming from different resource requesters are not handled in a fair and optimal manner. Resource requester can be anyone requesting cloud resources, for example, a team leader responsible for provisioning of resources needed by their team members in order to perform their everyday job. Quite frequently, consensus about the actual priorities for resource utilization is not achieved, and the loudest requester gets a better working environment (set of hardware resources). This arbitrary decisions lead to a non-optimal resource utilization and therefore, it puts additional costs on a company. Moreover, as a consequence, various delays in project may potentially occur, with inevitable frustration in teams, what results in an overall lower quality of a product. For large companies that are using a huge number of cloud resources, other side-effects such as energy consumption should be mentioned as well. As an illustration of a possible magnitude of consumption, a Google data centre consumes as much power as a city the size of San Francisco [6]. Thus, it is also important to have optimal resource utilization in Clouds in order to be energy efficient to reduce their power consumption.

In this paper, we propose a domain independent *policy-based scheduling mechanism for cloud services* that

*Distributed Systems Group, Johann Bernoulli Institute, University of Groningen, Nijenborgh 9, 9747 AG Groningen, NL, ({f.nizamic, v.degeler, a.lazovik}@rug.nl)

†Parasoft Netherlands B.V., Lange Voorhout 70, 2514 EH Den Haag, NL, (rix.groenboom@parasoft.nl)

‡The first two authors have contributed equally to this paper, and should therefore be both considered as first authors.

guarantees optimal resource utilization with respect to a total usage of cloud resources in a predefined time interval. We propose a novel data model for describing the requests for resource utilization. Several policies for scheduling are provided, though the developed approach is not limited to presented policies, and can be easily extended to incorporate other types of constraints. Additionally, in our approach we take into account dependencies between individual services that are forming a complete system, and present how enforcing of common usage of shared resources can lead to optimal resource utilization. With the example, we show the feasibility of our approach and how the reference implementation of our cloud scheduler optimizes a schedule and makes significant savings of resource usage in a cloud. The scheduler performance is evaluated and it has been shown that it scales well for a typical size of the resource allocation problems we consider in the paper.

This paper is organized as follows. First, in Section 2, we state the motivation for this work. In Section 3, we describe overall system architecture and describe its each component in detail. In Section 4, we describe the logic that resides in the Scheduler component. In Section 5 we present the demonstrating example, and show the whole workflow, from the request to an optimal schedule. In Section 6, we evaluate and discuss the performance of the Scheduler. In Section 7, we overview existing approaches to scheduling and allocation of resources within the cloud environment. Finally, in Section 8, we draw conclusion and discuss potential future work.

2. Motivation. Consider a scenario where a service provider wants to optimize the usage of the cloud resources used within software development process of the service it delivers. The service provider has multiple teams working together in order to bring the new version of the service. The main service is a composition of other (sub-)services, where all sub-services are separate and independent. In this scenario, there are six collaborating teams and their roles are the following. Development team is a team responsible for programming (actual development of the service). Testing team is responsible for quality assurance of the product. Acceptance testing team is responsible for final approval for software validity before the product is released (or not) to Production. Performance testing team is a team that ensures that quality of service (QoS) is in satisfactory limits. Proof of concept (POC) team is a team that does the research on new concepts before decision for the same to be implemented is made. Finally, Training team is a team that provides a training for employees that will use the developed service.

Each team has its own needs for resources on which they could deploy appropriate versions of software in order to perform their tasks. For instance, Acceptance and Performance testing teams need to have resources that are most similar to one in Production environment and minimal number of resources that can be shared with other teams. Reason for having high number of resources that are almost a mimic of Production environment is simple: before exposing developed system to end-users, rigorous tests should be executed on Production-like environment, that will minimize chance for surprises when the same code goes to Production. Reason to use as much as possible resources exclusively is that both teams simply do not want to have other teams interfering their tests. Excellent example of an exclusive resource usage demand is Performance team. In order to have precise results on on performance tests, they cannot share the same environment with other teams as it may affect the results of their performance measurements of the service. From the other side, Training team uses small number of resources for purposes of training of employees. For Training team, performance and accuracy of data is not particularly important. Therefore, they may share their environment and deployed services with other teams whenever it is possible. Note that different teams may use different versions of services. For example, Development team may already work on the second version of Service X while the Performance test team still executes tests and collects performance data from the previous ("first") version of the same Service.

For a human, this kind of scheduling is highly time-consuming, error-prone and costly in terms of required efforts.

3. System architecture. In this section, we describe cloud resources and specifics of requests for the cloud resources. Moreover, we describe behaviour of the system, system architecture and the responsibilities and functionality of each component of the system.

3.1. Cloud Resources. A Cloud can be seen as a set of computational and storage resources. In this work, we will refer to the smallest unit of a cloud infrastructure as a *resource*, be it CPU, memory or a single, stand-alone server. Resource is an abstraction that represents one unit/instance of a cloud used as computing or storage capacity. Resources can be shared or exclusively used, and can have different services deployed onto them. All resources in a cloud that are available for usage are located in a resource inventory. We assume that the number of available resources is limited by company's planned budget for the infrastructure (cloud

resources). In some cases, companies may want to limit a number of used resources to a number of free cloud resources offered by cloud resource providers*. In this work we focus on resource scheduling, and, without loss of generality, we will use hardware agnostic approach. In other words, in this paper hardware specific details such as CPU, memory and I/O will not be considered.

3.2. Requests for resources. The request for resource utilization come from *resource requesters*. To request the resource, one needs to know which services make a chain needed to fulfil a complete functionality, what type of resources are required, and for how long and in which way those resources will be used. These parameters represent input parameters for the scheduling service.

A *request for resources* is composed out of three following elements: a resource demand, a policy and a request. A resource demand contains: resource type, required number of resources, and information whether the resource can be shared or it must be exclusively used (e.g., resource requester needs two instances of Service X that can be shared or invoked by more internal service consumers). There can be more resource demands under one request for resources. A policy contains an amount of time for which resource is required, and a parameter which defines how resource need to be used (e.g., Service X is used for five consecutive days). From the perspective of Performance test team, one request for resources could be: in order to perform a load test on Service X which invokes Service Y, we need to demand two resources of types X and Y, which will be used exclusively for five consecutive days. This request for resource defines dependency between two services and that way embodies it into a form of request.

3.3. System behaviour. In Fig. 3.1, we propose a system architecture to provide cloud resources to resource requesters, taking into account the above-mentioned limitations. The sequence of actions and flow of information is the following. The resource requesters submit their requests to the *Resource request Service*. Then, the Scheduler Service is invoked. The *Scheduler Service* provides an optimized schedule as an output. The schedule defines which resources are assigned to which time slot. Subsequently, when the *Deployer Service* receives the schedule, it physically deploys the services to the resources, per information proposed in the schedule. After the deployment process is finalized, testing scripts are executed in order to define the status of the services, or/and to execute the initial preparations of the services. When services are up and running, *Distributed Configuration Service* keeps the track of physical locations (endpoints) of the services, and gives an input to *Monitoring Service* which shows the current status of each individual deployed service. In case that additional requests are submitted re-scheduling can be dynamically invoked, while a number of used cloud resources would stay within the limitations given by resource requester.

3.4. Resource Request Service. Resource Request Service is responsible for communication with Scheduler Service and preparation of requests in a form understandable by Scheduler Service itself.

3.5. Scheduler Service. *Scheduler Service* is responsible for provisioning of fully optimized *Schedule* as the output for structured requests for resources as an input. Provided schedule maps the requests for resources to the available time slots in optimal manner. In the following section, the scheduler service will be described in more detail.

3.6. Deployment Service. The deployment service is responsible for physical deployment of requested services to appropriate cloud instances. Input to the deployment service is a previously generated schedule for requests.

At the heart of the deployment service is *Apache Whirr*† which in turn relies on the *Jclouds library* which specializes in abstracting the connection and deployment to various on-line cloud services like *Amazon* and others. Development of Whirr is ongoing and is soon capable of supporting *Openstack*‡ which is used to create private clouds.

The goal of the Deployment service is to deploy the requests and configure the services defined to run. For example, if Service X depends on Service Y, once deployed, Service X should have knowledge of where Service Y is located. Configuration and scripting is done by relying on Apache Whirr's module for *Puppet*§. Using Puppet, we can easily automate installation and configuration of a wide array of services. Along with Puppet,

*aws.amazon.com/free
†whirr.apache.org
‡openstack.org
§puppetlabs.com

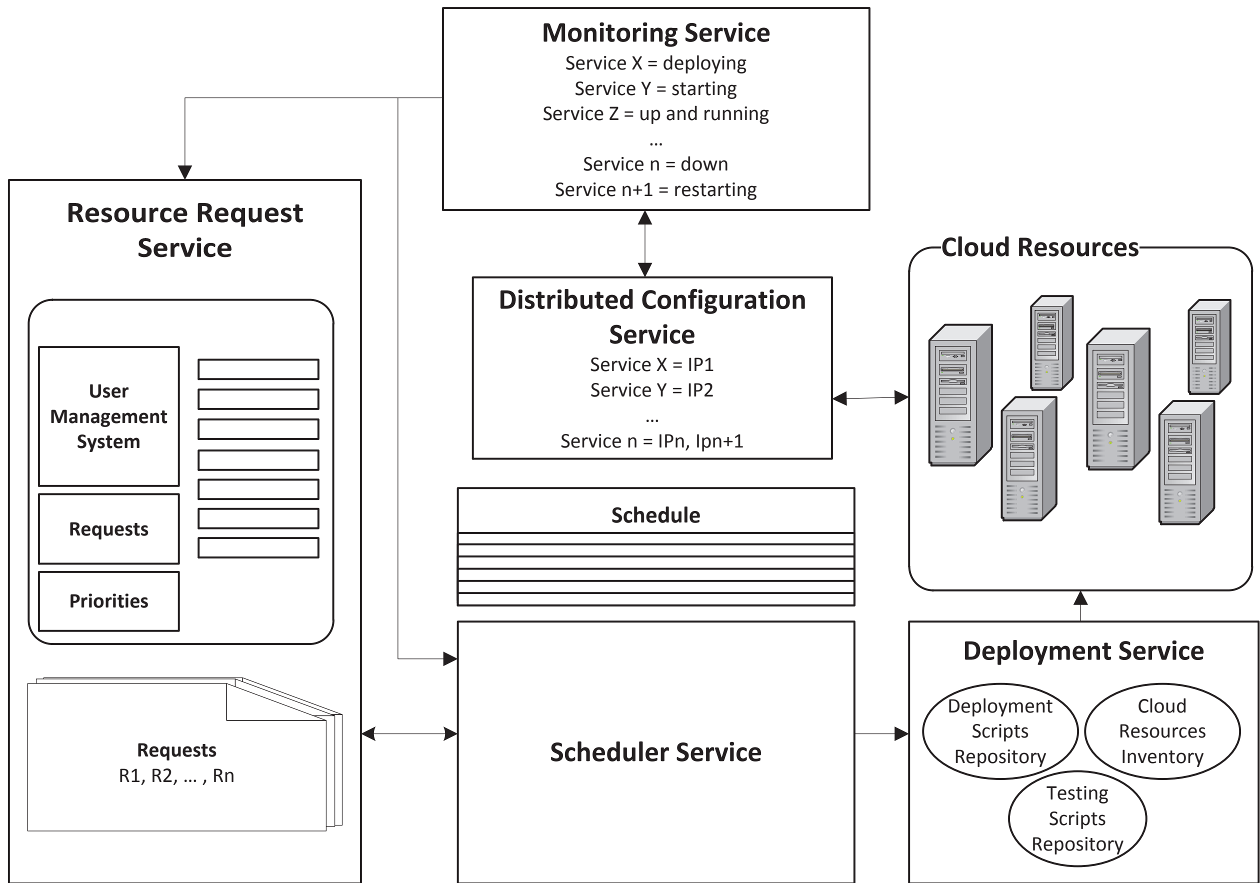

Fig. 3.1: System Architecture

Apache Whirr provides modules to directly configure a desired service instead of writing scripts for Puppet. All modules are available for selection in the deployment service.

Additionally the Deployment service is responsible for updating the Distributed Configuration Service (DCS). In our setting, we use *ZooKeeper*[¶] as a distributed configuration service. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. The goal of deployer is then to update ZooKeeper entries about exact service location (endpoint) and the status information. For example, after the Service X is deployed on a Cloud instance, in DCS table information that Service X can be accessed through certain endpoint is added.

3.7. Distributed Configuration Service. The Distributed Configuration Service (DCS) is responsible for maintenance of information about the physical location of each cloud instance. Initially, when services on instances are deployed, it will send the update to DCS which will store the information about its location. Location information will be represented in form of endpoints that will point to specific instances. Endpoints are composed of an IP address or a hostname and a port.

The Distributed Configuration Service is implemented as a Zookeeper service, and it uses its built-in support for group management. A client registers itself in a Zookeeper by creating a so called *ephemeral* node, that is automatically removed if the client does not send a heart beat within a given timeout. Zookeeper also provides features like replication and automatic fail-over. Therefore, clients maintain a TCP connection through which they:

1. send the requests for update of information about their physical location,

[¶]zookeeper.apache.org

2. get the responses which contain the physical location of other clients,
3. send a heart beat (that is also used by Monitoring Service).

If the TCP connection to Distributed Configuration service server breaks, the clients (system component instances) can connect to a different Distributed Configuration service server (replica) which contains the same information. This way we implement fault tolerance and avoid to have a single point of failure. While disconnected though, such component instances are not visible to other components until they are re-registered within DSC.

3.8. Monitoring Service. The Monitoring Service is responsible to represent the current state of each service deployed on cloud instance (up and running, down, instantiating, deploying, restarting, etc.). Additionally, information on performance of individual services is being collected.

4. Scheduler Service. The core logic of the system resides in the Scheduler service. It is responsible for optimization of the total number of resources required to satisfy the requests. The Scheduler service has a REST interface and is invoked at the beginning of the time interval to be scheduled, with all requests for the following time interval. The requests are stored and passed in *Google Protocol Buffers format*‖.

The Scheduler service has two parts: Cloud Schedule Interface (CSI) and Scheduling Core. Scheduling Core is a domain-independent scheduling algorithm, and can be used within any domain, as long as constraints of scheduling are specified in similar policy types. CSI is a wrapper on top of Scheduler Core and is specific to the cloud scheduling optimization problem. The task of this component is to transform cloud schedule requests to the domain-independent representation within Scheduling Core, and to transform back the resulting schedule to the required form.

4.1. Cloud Schedule Interface. As input to the optimization task, the Scheduler receives a list of *requests*.

Definition 4.1 (Request). *A* request *is a full specification of the number and type of* resources *needed to satisfy a certain task, together with the* policy *of resources usage.*

The informal examples of requests may be "five cloud instances are needed for a total of eight hours running time to execute Services X, Y, and Z", or "three cloud instances are needed to run continuously for twelve hours to execute Service K as a shared service", etc. To satisfy the request, the requested number of resources should be available for the required amount of time. Partial satisfaction of a request is not possible, since partially satisfied request means an incomplete task. A request should either be satisfied fully, or not at all.

Thus, to fully define the request, we first need to define its two most important parts: the list of *resource demands*, and the execution *policy.*

Definition 4.2 (Resource demand). *A* resource demand *is the full specification of a resource that is needed to complete the task, which includes the specification of the type of service which should be running, the number of services, and whether the service can be shared with other tasks, or must be run exclusively.*

The data model uses Google Protocol Buffers format, where all variables in a message are described as a tuple: *"modifier, type, variable name = parameter id"*. In the code presented in this paper we omit *parameter id* for clarity purposes.

The data for resource demand written as follows:

```
message ResourceDemand
  optional uint32 resId;
  optional uint32 number;
  optional bool   shared;
```

Here the "*resId*" uniquely represents the service to run, "*number*" represents the number of such services that should be deployed, and a boolean value "*shared*" tells whether the service can also be used by other requests, or should be run exclusively by this request.

While by using a list of resource demands, we can specify all the resources that we need, we also should specify the time frame for them to run, and the execution policy. For example, one task may require for services to be run for twelve consecutive hours, while another task may require them to run for twenty four hours, while not caring whether those hours are consecutive, or split apart.

‖code.google.com/apis/protocolbuffers

In this work we use five predefined types of policies. All policies are formally described below. For all policies, we define T as the total number of time slots, p_{ij} as the scheduled status of request r_i at time slot j ($p_{ij} = true$, if request r_i is scheduled for the time slot j, and $p_{ij} = false$ otherwise).

Total. The policy has an additional parameter d ("*duration*"), and assumes that resources should be available for the total number of time slots, equal to the "*duration*" value. How the time slots are split over the whole scheduling period is not important, thus the task can be split, and, for example, it can run on Monday, Wednesday, Friday, or on Monday to Wednesday.

$$\forall r_i,\ r_i.policy = TOTAL:\ \sum_{j=1}^{T} (p_{ij} = true) = d$$

Continuous. This policy is stricter, and guarantees that once the request is started, it will run *uninterrupted* for the required number of time slots, also specified by the parameter d ("*duration*").

$$\forall r_i,\ r_i.policy = CONTINUOUS:\ \exists k:\ 1 \le k \le T - d + 1\ s.t.\ \forall j = 1..T\ :\ p_{ij} = true\ \Leftrightarrow\ k \le j < k + d$$

Multiple. The policy allows for more than one job to be scheduled within the same request. Each job must have resources within uninterrupted period of time, but jobs themselves may be split in time, for example, one job can be executed on Monday, and two more on Thursday. In addition to the d ("*duration*") of the job parameter, the policy also has a n ("*number of jobs*") parameter.

$\forall r_i,\ r_i.policy = MULTIPLE:\ \forall l = 1..n\ \exists k_l\ :\ 1 \le k_l \le T - d + 1,\ \nexists (k_s, k_t)\ :\ |k_s - k_t| < d\quad s.t.\quad \forall j = 1..T\ :\ p_{ij} = 1 \Leftrightarrow\ \exists k_x : 0 \le j - k_x < d$

Repeat. The policy has two parameters: c ("*cycle duration*") and d ("*total time to be scheduled within a cycle*"), and assumes that a resource should be available cyclically with a certain periodicity. Example are regression tests that must be run for an hour every day (to test nightly builds).

$$\forall r_i,\ r_i.policy = REPEAT:\ (\sum_{j=1}^{c} p_{ij} = d)\ \bigwedge\ (\forall j = c + 1..T\ :\ p_{ij} = p_{i,j-c})$$

Strict. The policy firmly defines the specific schedule for certain resource requests. Thus these resource requests cannot be moved to different time slots, but the knowledge about them allows Scheduler to schedule other requests to share resources with the strictly defined requests, whenever possible.

$$\forall r_i,\ r_i.policy = STRICT:\ \exists F_i(t)\ s.t.\ \forall j = 1..T\ :\ p_{ij} = F_i(j))$$

Thus, the data model to specify the policy is the following:

```
enum PolicyType
  TOTAL;
  CONTINUOUS;
  MULTIPLE;
  REPEAT;
  STRICT;
```

```
message Policy
  required PolicyType type;
  optional uint32     duration;
  optional uint32     numberJobs;
  optional uint32     cycleDuration;
  repeated uint32     strictTimeOn;
```

Now that we have specified both the policy data model and the resource demands data model, we can fully specify the request:

```
message Request
  required uint32 reqId;
  optional Policy policy;
  repeated ResourceDemand demand;
```

Note that each request can contain a list (specified by keyword '*repeated*') of different resource demands, and to satisfy the request, all resource demands must be satisfied at the same moment of time.

In order to create a schedule from requests, besides a list of requests, additional information is also required. First of all, a number of available time slots over the whole scheduling period should be given (e.g., 24 hours times 5 working days equals 120 available time slots). Furthermore, the total number of resources available at

Algorithm 4.2.1 Scheduler Core searching algorithm high-level overview

1: $q \leftarrow PriorityQueue[search_node]$
2: $q.add(\langle 0; 0; [] \rangle)$ *//initialise queue with empty schedule*
3: **while** !q.isEmpty **do**
4: $\quad \langle c; t; ps \rangle \leftarrow q.pop()$
5: $\quad R_f \leftarrow resources$ s.t. for the next time unit $t+1$: $isFeasible(R_f, t+1, ps)$
6: $\quad$ **for** $r_f \leftarrow PowerSet(R_f)$ **do**
7: $\quad\quad$ **if** $!isAlternative(r_f)$ **then**
8: $\quad\quad\quad q.add(\ \langle c + cost(r_f);\ t+1;\ ps + r_f \rangle\ )$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end while**

the same time should be specified (e.g., 50 cloud instances). If resources represent a single instance in a cloud, it is usual for the cloud providers to charge more per instance, if many instances are used at the same time. Additional costs can be avoided in case we limit our execution by not using more than a certain number of instances at each moment of time. Thus, the schedule request data model is the following:

```
message ScheduleRequest
  repeated Request reqList;
  required uint32  numSlots;
  required uint32  numResources;
```

"*reqList*" is the list of requests, "*numSlots*" is the number of available time slots (usually an hour, but can also be any other time interval), "*numResources*" is the maximum number of resources that can be used at the same time.

As a result of the Scheduler execution, we obtain a full schedule of requests distributed over available time slots. For each time slot, the Scheduler presents a list of request IDs to show which requests should run at this time. The data model for the Scheduler response is the following:

```
message ScheduledTimeSlot          message Schedule
  repeated uint32 reqList            repeated ScheduledTimeSlot schedule
```

We can optimize the resource usage by maximizing the reuse of shared resources. If requests require same shared resources, placing them at the same time slots will enable maximum reuse.

4.2. Scheduler Core. The Scheduler Core is the actual implementation of the scheduling algorithm. It is domain-independent, and can be used for other domains, as long as they can be specified using similar policies as constraints to the schedule optimization. For example, we also use the Scheduling Core to schedule the working time of home appliances (such as fridge, boiler, printer, etc.), to reduce the cost of energy consumption [7]. The Schedule Interface is different for that case, and also transforms that task to the same data structure.

For solving our task, as a searching strategy within the Scheduler Core we implement a priority queue with *Breadth-First Search (BFS) algorithm* [8]. Using this algorithm over other possible conventional search strategies [8] allows us to minimize the search space, since we use the cost of intermediate solution as a prime factor for search expansion. The high-level overview of the search can be seen in Algorithm 4.2.1. We create a priority queue with *a search node* that corresponds to a partially fulfilled schedule. Each search node has the following structure and is prioritized by its cost:

$search_node = \langle cost, time_units, partial_schedule \rangle$

$partial_schedule$ is a state matrix $partial_schedule = T \times R$, where $T \in 1..time_units$, and R is a set of resources. The matrix shows, for each time slot, in which state the resource was at this time slot. For the purpose of our paper we treat this matrix as boolean (resource is either scheduled at a time slot, or not), though in general we assume more possible states for each resource (can be useful in other domains).

The queue starts with empty schedule. During each search step it takes the schedule with the least cost and tries to add possible distribution of resources to the next time slot.

Our main contribution to scheduling strategies lies in definition of policies in such a way to drastically reduce search space. Since we know the constraints that policies impose on a possible solution, we can restrict in advance many solutions that will violate those constraints at the end. By doing this we prevent many "dead-end" partial solutions from further expansion, thus saving time. In the algorithm this is defined by two functions: *isFeasible*, which prevents from searching all schedules that breach at least one policy, and *isAlternative*, which finds if several different partial schedules actually both have the same outcome, which means that we only need to continue searching one of them, and safely drop all others.

4.3. Feasibility check. We decrease search space by extensive usage of policy restrictions. For example, if a request has the policy *total*, it means it should have the available resources for a certain number of time slots, so we automatically restrict the search space to only those schedules that have this request satisfied for exactly this number of time slots, and remove those that have a request satisfied for more or less. Because having a request satisfied for fewer days means the request is not fully fulfilled. While having it satisfied for more days means we unnecessarily schedule more resources for usage, thus such a schedule is intrinsically not optimal. Thus, for the resource with a policy *total*, we have two constraints. The first one is that the current number of time slots with scheduled resource should not exceed total expected time for resource scheduling. The second constraint is that the number of time slots left unscheduled should not exceed the difference between total expected time and current scheduled time. So, for a current time slot t the following formal description holds.

$$C_{TOTAL}: \sum_{j=1}^{t} p_{ij} \leq d \bigwedge T - t \geq d - \sum_{j=1}^{t} p_{ij}$$

For the *continuous* policy, while searching for the optimal schedule we remove all partial schedules that assume a number of continuously used slots not equal to the total number of required time slots. All restrictions of the policy *total* are also applied to the policy *continuous*.

$$C_{CONTINUOUS}: C_{TOTAL} \bigwedge (p_{it} = 0) \Rightarrow ((\sum_{j=1}^{t} p_{ij}) = 0 \vee (\sum_{j=1}^{t} p_{ij}) = d)$$

For *multiple* the total uninterrupted time should be divisible to the duration of one job. For example, if a job lasts two hours, and we found a partial schedule that proposed to schedule the request for three hours, we can immediately see, that one hour the request will unnecessarily occupy resources. Restrictions of the policy *total* are applicable here as well.

$$C_{MULTIPLE}: C_{TOTAL} \bigwedge (p_{it} = 0) \Rightarrow ((\sum_{j=1}^{t} p_{ij}) \ mod \ d = 0)$$

Repeat policy is checked as *total* within the first cycle, and for all time slots after the first cycle, the full periodicity is applied.

$$C_{REPEAT}: (t \leq c) \Rightarrow \sum_{j=1}^{t} p_{ij} \leq d \bigwedge T - t \geq d - \sum_{j=1}^{t} p_{ij}; \ (t > c) \Rightarrow (p_{it} = p_{i,(t-c)})$$

Strict policy does not need any feasibility checking, because it is already strictly defined. On the other hand, there is only one way to satisfy a strict policy, which means it does not add complexity to the search space.

$$C_{STRICT}: p_{it} = F_i(t)$$

4.4. Alternatives check. Let us say that we have two different partial schedules for a request with a "*total*" policy. If the total cost of these two schedules is the same (which may be or not be the case, depending on sharing resources with other requests), for the purpose of finding the schedule for the next time period those two schedules for this request are identical, as both schedules assigned the same number of time slots for a request. Which means we should only continue searching one of the schedules, and we can safely drop the other, as it will not produce better result. Similar techniques can be used for other policies as well.

We can only drop one of two partial schedules if (1) they have the same total cost; (2) they have the same number of scheduled time slots; (3) for each request we determine that both schedules arrive to the same current situation. The way to determine it differs per policy.

For the *total* policy only the number of already assigned time slots matters, but not their exact position. For example, if after 30 time slots we determine, that both schedules schedule a certain request for 6 times, we can regard them as the same for this request, no matter when were the exact times when this resource was scheduled. The *continuous* policy and the *multiple* policy are the same as *total*, we only check the total assigned time slots. The additional restrictions to the schedule are already checked at the feasibility check point, so we already know that both schedules are feasible.

We can only regard two schedules for a request with the *repeat* policy as similar in case the distribution of the assigned time within a cycle is completely the same. The reason is the distribution may matter later in the schedule, but it cannot be changed, once created during the first cycle. So two schedules for the request with repeat policy are regarded as the same when they really have the same assignment distribution within a cycle.

Finally, the *strict* policy is always the same, because there is only one way to satisfy this policy.

5. Demonstrating example. In order to show that Scheduler provides optimization, we have defined a demonstrating example that represents a common situation for companies developing software as a service. Let as assume that *International Phonebook Company* has five different teams that work on development of one product, *Phonebook Service* (see Fig. 5.1), and one team that provides training to employees.

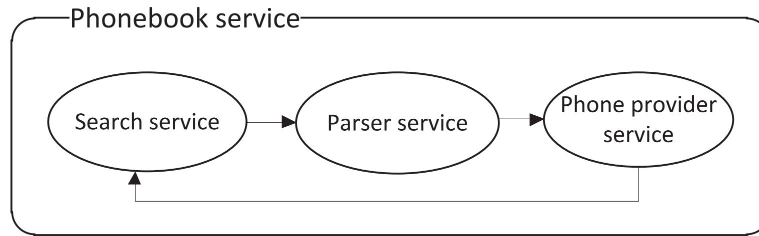

Fig. 5.1: Phonebook Service

Phonebook service is composed out of three independent services: *Search service*, *Parser service* and *Phone provider service*. Each service provides unique function. *Search service* takes free form text (*name, surname, address*) as an input and forwards it to *Parser service*. *Parser service* parses the free form text and formats it to XML form, so values of a *name*, a *surname* and an *address* are assigned to corresponding XML fields. Subsequently, *Parser service* forwards the XML formatted input to *Phone provider service* which forms a query based on the XML input and returns a phone number as an output to the *Search service*. Set of these services provides the full functionality of *Phonebook service*, and that is, for a set of user inputs (name, surname, address) it provides a corresponding phone number. That way, one request embodies a specification of needed complete working environment that provides the full functionality of the system. Also, one request links more services and that way implicitly defines dependencies among them. List of the requests for resources required by each of the teams is presented in Table 5.1.

Table 5.1: List of requests

Request	**Dev.**		**Test**		**UAT**		**Perf.**		**Train.**		**POC**	
Shared?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Search Service v1						8		8	4			
Search Service v2		4	4								4	
Parser Service v1					4			4	1			
Parser Service v2	2		2									
Parser Service v3												2
Phone Provider Service v1								2				
Phone Provider Service v2	1		1		2						1	
Duration (hours)	72		2		48		24		2		24	
Cycle duration	-		24		-		-		-		-	
Number of jobs	-		-		-		-		6		-	
Policy	Cont.		Repeat		Cont.		Cont.		Multi		Total	

Each team has different needs for resources. Those needs are reflected in description how and how long resources will be used; if resources can be shared with other teams or not, and if they need to be used continuously, repeatedly or if some other policy should be implemented. For example, Performance team needs to run their tests without interruption (continuously) in order to reach wanted load, while testing team wants to run their regression test repeatedly after every deployment of a new build. According to those requirements, scheduling of resources needs to be done. In most cases, scheduling is done manually by person responsible for scheduling of environments (environment administrator). Manual scheduling usually leads to non-optimal usage resources and it is something that we want to avoid.

Given the total of 120 hours (5 working days 24 hours each), and the limit of maximum 25 simultaneously used resources, schedule for one working week produced by the Scheduler is the following.

Table 5.2: Optimized schedule

	Mon	**Tue**	**Wed**	**Thu**	**Fri**
Development	20:00-23:59	00:00-23:59	00:00-23:59	00:00-19:59	
Test	22:00-23:59	22:00-23:59	22:00-23:59	22:00-23:59	22:00-23:59
UAT			00:00-23:59	00:00-23:59	
Performance					00:00-23:59
Training				08:00-19:59	
POC	20:00-23:59	12:00-23:59		20:00-23:59	20:00-23:59

Total number of used resources provided by this schedule is 1680 server-hours, and it is optimal in regard to number of resources used in one working week. Every other schedule would lead to less or in best case equally good solution.

6. Evaluation. Finding the optimal schedule is an expensive task in terms of computational resources, as it is NP-hard problem [16]. While the current scheduler is designed to schedule the reasonably low and stable number of requests (such as a request per software engineering team), the number of time slots of the schedule can and expected to be reasonably high. Nevertheless, we also ensured that the Scheduler can sustain a certain level of demand increase, and remain practical for higher number of requests.

As the scalability is important characteristic of the cloud computing, the performance evaluation in this section investigates the ability of the scheduler to scale with the increase in the number of time slots, and also shows the usability of the scheduler with the increase in the number of requests.

6.1. Number of time slots. In many situations the scheduling of resources in a cloud should be done on an hourly basis. Thus, if we take into account a working week, the number of time slots can be up to 40 (8 hours times 5 days). Thus the ability of the scheduler to scale with respect to time slots is important. We performed an experiment to run the Scheduler with 5 randomly generated requests and schedule them on a period from 5 to 50 time slots. The results can be seen in Fig. 6.1. As can be seen, even the scheduling for 50 time slots takes only about 2.8 seconds. Given into account that this scheduling is done for the distribution of resources over the full coming week, the performance is within perfectly acceptable bounds.

6.2. Number of requests. The number of requests causes much bigger strain on a scheduler, because at each time slot it needs to regard 2^{nReq} possibilities. As mentioned before, the scheduler is optimized to work well and to find optimal solution under small and stable number of requests. However, since we assume the possibility of requests increase, we implemented the dynamic relaxation of the optimality requirement, and instead we try to search fast for a "good enough" solution. The dynamic relaxation is done by implementing a gradual approach in the following way: if the number of requests is higher than a certain predefined number (in the experiment it was set to 8), the requests are split on several groups. We run the Scheduler for the first group, obtain the optimal schedule for this group, and than freeze the already scheduled requests in their time slots, and begin to schedule a second group, taking into account the already scheduled requests, and so on. Note that while this approach is "greedy", thus not guaranteed to return the optimal solution for the full number of requests, the returned solution is still effective enough, because if the next group of requests contains resources that can be shared with those in the previous groups, this situation is always detected and it automatically gives preference to those time slots that allow for maximum sharing of resources with already scheduled requests. Figure 6.2 shows the time needed to schedule up to 100 requests.

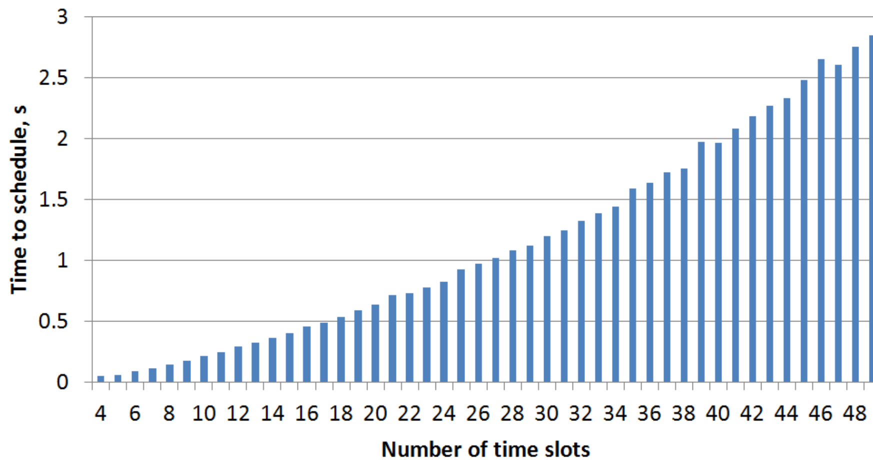

Fig. 6.1: Scheduler performance based on the number of time slots.

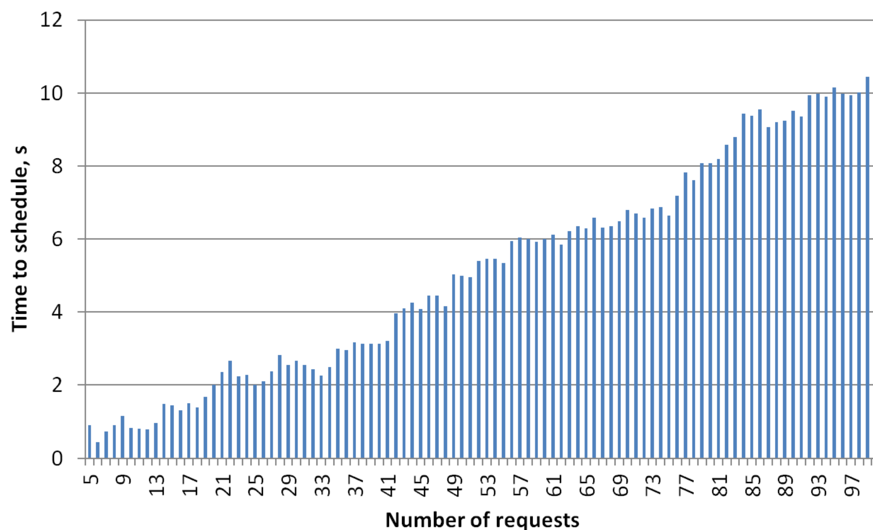

Fig. 6.2: Scheduler performance based on the number of requests.

7. Related work. The problem of scheduling of cloud resources has been addressed in a number of papers. We present some of the related work and compare it to our approach. Cloud scheduler described in [9] manages user-customized virtual machines in response to a user's job submissions. Its main motivation is to provide computing resources to the research community. Similarly, in [10], solution is oriented toward the same application area by providing a scheduling scheme for scientific applications which require large-scale computing resource for long term execution period. Contrary to this, the motivation of our work is to provide scheduling mechanism for highly demanding requests for resources of multiple collaborating teams inside software development companies. Moreover, our scheduler guarantees optimality of schedule in regard to number of used resources for a defined interval of time, that was not tackled neither in [9] nor in [10]. In [11], different metrics such as the change of load are used to dynamically schedule cloud resources. By real-time monitoring of performance parameters of virtual machine, scheduling of cloud resources is being done using *ant colony* algorithm to bear some load on load-free node. On the other hand, our scheduler as an input has user-defined metrics, such as resources specification, requested usage duration and policies.

Scheduling of grid applications on clouds is presented in [12] where not only resource demands are taken

into account, but also software requirements of the applications. This approach is similar to ours in sense of taking a content of resources into consideration. Difference is that our approach is focused on service-oriented systems, whereas in [12] they are using grid application of image processing. Besides that, we introduce dependencies among services and the way to manage them. Work done in [13] proposes a scheduler which makes scheduling decision by evaluation the entire group of tasks in job queue. The preliminary simulation results show that scheduler can get shorter "make span" for jobs and achieve better balanced load across all the nodes in the cloud. Instead, our scheduler enables control of maximum number of used resources per a given interval of time. Additionally, there are two papers, [14] and [15], which are focusing on inter-cloud scheduling (scheduling for cloud federations). That is completely different problem, but both papers provide useful insight into specifications, scheduling, and monitoring of services. There are a couple of industry white papers that present how usage of cloud resources can support Agile Software development [2], [4]. The main idea of these papers is that realization of automated builds, testing and production deployment in clouds can accelerate feedback mechanism that is crucial for Agile software development methodology. Current implementations presented in these papers are quite promising and it is left to us to research how this developments can be exploited.

8. Conclusion and future work. Proper scheduling of Cloud resources used in development process for software as a service can save time, effort and money. In this work, we have developed a scheduling mechanism of a Cloud middleware that guarantees optimal resource utilization in terms of total number of used resources in a given interval of time. In addition to optimization, our Scheduler provides fair scheduling for multiple collaborating cloud users that have highly demanding requests for cloud resources. We introduced dependencies between individual services and introduced the way how those services can be composed. By forming data models with user-defined inputs for scheduler, we have developed scheduling policies and created a good basis for additional extensions. We have shown that with our solution, cloud resources are used in optimal manner. This implies that beside making additional resources free for use, possibility of occurrence of project delays, the most expensive event in software development, is minimized. Additionally, by limiting maximum number of used resources, free usage of some resources on a cloud can be enabled without risk of having unwanted costs.

We have left enough space for additional improvements in future work. First of all, system can be enriched by taking into account priorities between different requests. Definition of priorities would give additional possibility for distinction of requests coming from different teams which have different importance in a certain time intervals. The model could be additionally improved by introducing reconfigurability of system parameters (e.g. time slot size, earliest start time of request, etc.), and expanding existing policies. Moreover, introduction of simulated services could eliminate a need for some of exclusively used resources. Furthermore, the load of the cloud resources should also be taken into account to fine-tune the scheduler. Finally, to multiply effect, future work should tackle efficient model for resource scheduling for geographically distributed software development teams working around-the-clock.

Acknowledgement. The authors would like to thank to Michiel van der Waaij and Rudi van Drunen for general advice and providing of examples from their day-to-day work, and Master student Werner Buck for useful inputs regarding deployment service. Additionally, the authors would like to thank to prof. dr. Marco Aiello for providing useful comments about this work.

REFERENCES

[1] Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M., *A break in the clouds: towards a cloud definition.*, SIGCOMM Comput. Commun. Rev. 39 (2008) 50–55

[2] CollabNet, I., *Reinforcing agile software development in the cloud.*, (2011)

[3] CIO Custom Solusions Group, *Physical infrastructure: a critical factor in cloud deployment success*, white paper (2012)

[4] Dumbre, A., Senthil, S.P., Ghag, S.S., *Practicing agile software development on the windows azure platform.*,(May 2011)

[5] Nizamic, F., Groenboom, R., Lazovik, A., *Testing for highly distributed service-oriented systems using virtual environments.*, In: Proceedings of 17th Dutch Testing Day. (2011)

[6] Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I., *Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility.*, Future Gener. Comput. Syst. 25 (June 2009) 599–616

[7] I. Georgievski, V. Degeler, G. A. Pagani, T. A. Nguyen, A. Lazovik, and M. Aiello, *Optimizing Energy Costs for Offices Connected to the Smart Grid.*, IEEE Transactions on Smart Grid, 2012.

[8] Russell, S.J., Norvig, P., *Artificial Intelligence: A Modern Approach*, 2nd Ed. Prentice Hall, Englewood Cliffs, NJ (2002)

[9] Armstrong, P., Agarwal, A., Bishop, A., Charbonneau, A., Desmarais, R.J., Fransham, K., Hill, N., Gable, I., Gaudet, S., Goliath, S., Impey, R., Leavett-Brown, C., Ouellete, J., Paterson, M., Pritchet, C., Penfold-

Brown, D., Podaima, W., Schade, D., Sobie, R.J., *Cloud scheduler: a resource manager for distributed compute clouds.*, CoRR (2010)
[10] Kim, S., Kim, Y., Song, N., Kim, C., *Adaptable scheduling schemes for scientific applications on science cloud.*, In: IEEE Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), (Sep 2010) 1–3
[11] Lu, X., Gu, Z., *A load-adapative cloud resource scheduling model based on ant colony algorithm.*, In: Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on. (Sep 2011) 296 –300
[12] Chaves, C., Batista, D., da Fonseca, N., *Scheduling grid applications on clouds.*, In: IEEE Global Telecommunications Conference (GLOBECOM). (dec. 2010) 1–5
[13] Ge, Y., Wei, G., *Ga-based task scheduler for the cloud computing systems.*, In: Int. Conf. Web Information Systems and Mining (WISM). (Oct 2010) vol.2, 181–186
[14] Larsson, L., Henriksson, D., Elmroth, E., *Scheduling and monitoring of internally structured services in cloud federations.*, In: Computers and Communications (ISCC), 2011 IEEE Symposium on. (2011) 173 –178
[15] Sotiriadis, S., Bessis, N., Antonopoulos, N., *Towards inter-cloud schedulers: A survey of meta-scheduling approaches.*, In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2011 International Conference on. (Oct 2011) 59 –66
[16] Chen, B., Potts, C.N., Woeginger, G.J., em A review of machine scheduling: Complexity, algorithms and approximability, D.Z. Du & P. Pardalos, Handbook of Combinatorial Optimization, Kluwer Academic Publishers, 1998.

Edited by: José Luis Vásquez-Poletti, Dana Petcu and Francesco Lelli
Received: Sep 20, 2012
Accepted: Oct. 15, 2012