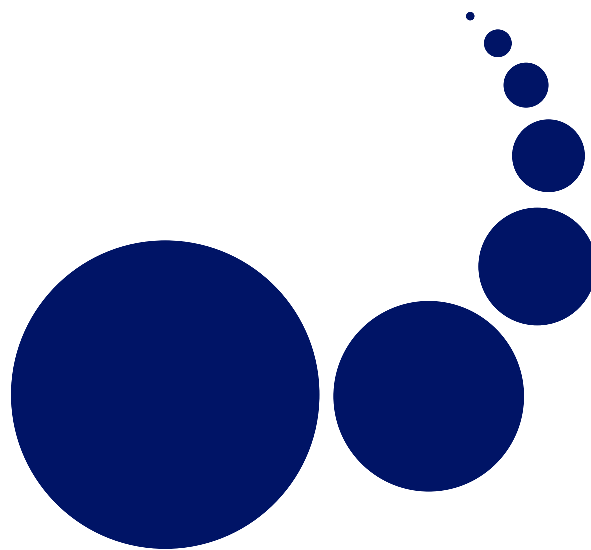
SCALABLE COMPUTING
Practice and Experience

Special Issue: Selected Papers from the International Workshop on Clouds for Business and Business for Clouds

Editors: José Luis Vázquez-Poletti, Dana Petcu and Francesco Lelli



Volume 13, Number 3, September 2012

ISSN 1895-1767

Editor-in-Chief

Dana Petcu
Computer Science Department
West University of Timisoara
and Institute e-Austria Timisoara
B-dul Vasile Parvan 4, 300223
Timisoara, Romania
petcu@info.uvt.ro

Managinig and
TEXnical Editor

Frîncu Marc Eduard
Computer Science Department
West University of Timisoara
and Institute e-Austria Timisoara
B-dul Vasile Parvan 4, 300223
Timisoara, , Romania
mfrincu@info.uvt.ro

Book Review Editor

Shahram Rahimi
Department of Computer Science
Southern Illinois University
Mailcode 4511, Carbondale
Illinois 62901-4511
rahimi@cs.siu.edu

Software Review Editor

Hong Shen
School of Computer Science
The University of Adelaide
Adelaide, SA 5005
Australia
hong@cs.adelaide.edu.au

Domenico Talia
DEIS
University of Calabria
Via P. Bucci 41c
87036 Rende, Italy
talia@deis.unical.it

Editorial Board

Peter Arbenz, Swiss Federal Institute of Technology, Zürich, arbenz@inf.ethz.ch
Dorothy Bollman, University of Puerto Rico, bollman@cs.uprm.edu
Luigi Brugnano, Università di Firenze, brugnano@math.unifi.it
Bogdan Czejdo, Fayetteville State University, bczejdo@uncfsu.edu
Frederic Desprez, LIP ENS Lyon, frederic.desprez@inria.fr
Yakov Fet, Novosibirsk Computing Center, fet@ssd.sscc.ru
Andrzej Goscinski, Deakin University, ang@deakin.edu.au
Janusz S. Kowalik, Gdańsk University, j.kowalik@comcast.net
Thomas Ludwig, Ruprecht-Karls-Universität Heidelberg, t.ludwig@computer.org
Svetozar D. Margenov, IPP BAS, Sofia, margenov@parallel.bas.bg
Marcin Paprzycki, Systems Research Institute of the Polish Academy of Sciences, marcin.paprzycki@ibspan.waw.pl
Lalit Patnaik, Indian Institute of Science, lalit@diat.ac.in
Boleslaw Karl Szymanski, Rensselaer Polytechnic Institute, szymansk@cs.rpi.edu
Roman Trobec, Jozef Stefan Institute, roman.trobec@ijs.si
Marian Vajtersic, University of Salzburg, marian@cosy.sbg.ac.at
Lonnie R. Welch, Ohio University, welch@ohio.edu
Janusz Zalewski, Florida Gulf Coast University, zalewski@fgcu.edu

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 13, Number 3, September 2012

TABLE OF CONTENTS

Special Issue Selected Papers from the International Workshop on Clouds for Business and Business for Clouds:

Introduction to the Special Issue **iii**
José Luis Vázquez-Poletti, Dana Petcu and Francesco Lelli

Policy-Based Scheduling of Cloud Services **187**
Faris Nizamic, Viktoriya Degeler, Rix Groenboom and Alexander Lazovik

Challenges for the comprehensive management of Cloud Services in a PaaS framework **201**
Sergio García-Gómez, Miguel Jiménez-Gañán, Yehia Taher, Christof Momm, Frederic Junker, József Biro, Andreas Menychtas, Vasilios Andrikopoulos and Steve Strauch

On Engineering Cloud Applications - State of the Art, Shortcomings Analysis, and Approach **215**
Yehia Taher, Dinh Khoa Nguyen, Francesco Lelli, Willem-Jan van den Heuvel and Mike P. Papazoglou

Approaches to Aggregate Price Models to Enable Composite Services on Electronic Marketplaces **233**
Frederic Junker, Jürgen Vogel and Katarina Stanoevska

Datastores supporting Services Lifecycle in the framework of Cloud Governance **251**
Adrian Copie, Teodor-Florin Fortiş, Victor Ion Munteanu and Viorel Negru

Regular Papers:

A Distributed Program Global Execution Control Environment Applied to Load balancing **269**
Janusz Borkowski, Damian Kopa, Eryk Laskowski, Richard Olejnik, Marek Tudruj

© SCPE, Timişoara 2012

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

INTRODUCTION TO THE SPECIAL ISSUE ON SELECTED PAPERS FROM THE INTERNATIONAL WORKSHOP ON CLOUDS FOR BUSINESS AND BUSINESS FOR CLOUDS

Dear SCPE readers,

The present issue of Scalable Computing: Practice and Experience is devoted to the use of cloud computing for business and the application of business in the cloud computing area.

It's clear that cloud computing has acquired enough maturity to expand its field of application to business. The number of institutions that are moving their production lines onto the cloud is increasing year after year, and so are the emerging companies that are offering services that follow the cloud provision model.

The International Workshop on Clouds for Business and Business for Clouds (C4BB4C) held at the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA2012) (Madrid, July 2012) put together efforts done from both service producers and consumers. The objective was to identify the best cloud practices that produce an added value for the economy of any kind of institution. Technologies, policies and heuristics were shared, without discarding those coming from other areas that would benefit the cloud computing paradigm.

All contributions to the present issue of SCPE except one are extended version of accepted papers presented at the Workshop. In general, these papers intend to focus on how services are delivered through the cloud platform as a popular strategic technology choice for business. They remark cloud computing as a consistent platform for flexible and ubiquitous service provision from anywhere and at any time.

Also, the interface of software services on the cloud provides a rich area for research. As it will be seen, the editorial board for this issue encouraged the submission of contributions from a wide range of areas, comprising cloud computing, service engineering and business process management, coming from both the Academia and Industry.

José Luis Vázquez-Poletti
Distributed Systems Architecture Group
Facultad de Informática, Universidad Complutense de Madrid

Dana Petcu,
Computer Science Department
West University of Timisoara
and Institute e-Austria Timisoara, Romania

Francesco Lelli,
European Research Institute in Service Science

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

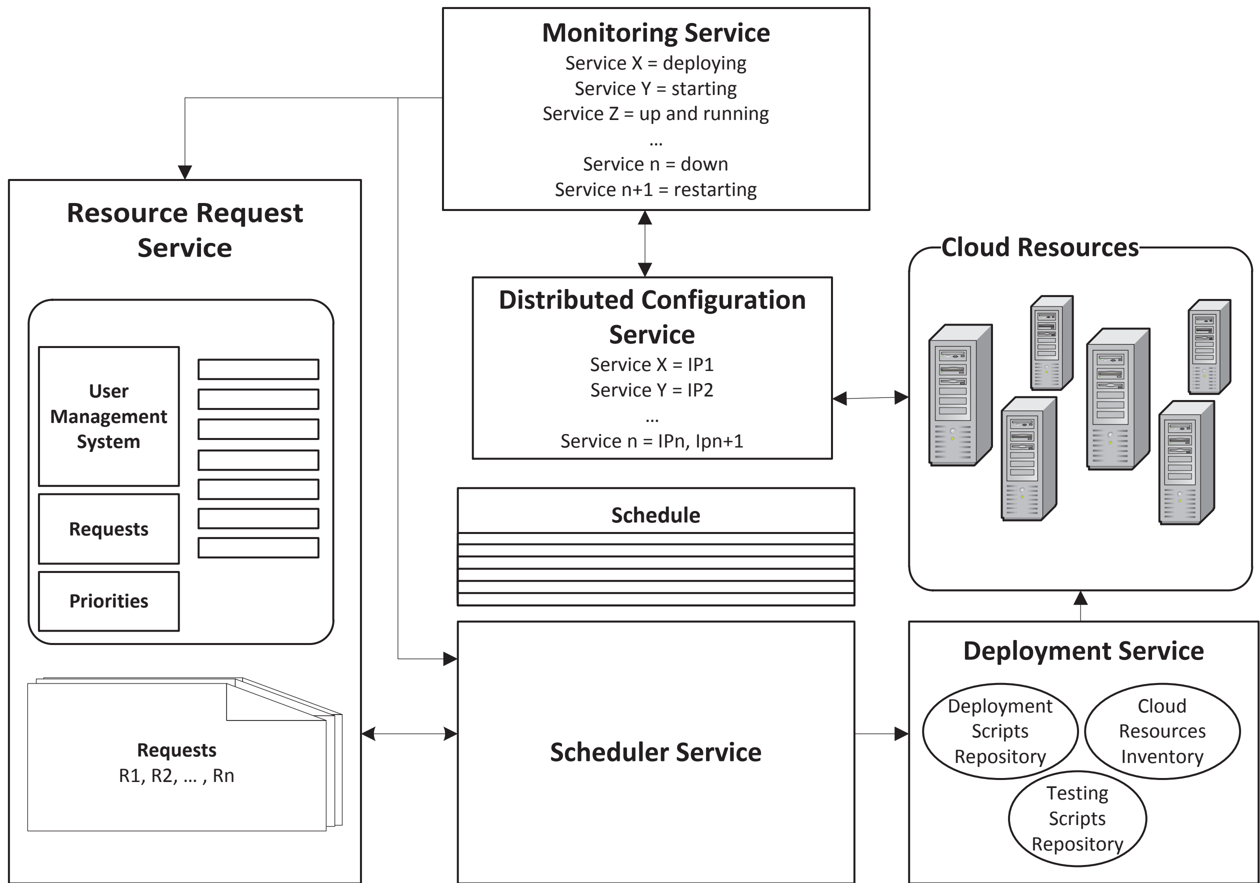


Fig. 3.1: System Architecture

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.

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

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.

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

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.

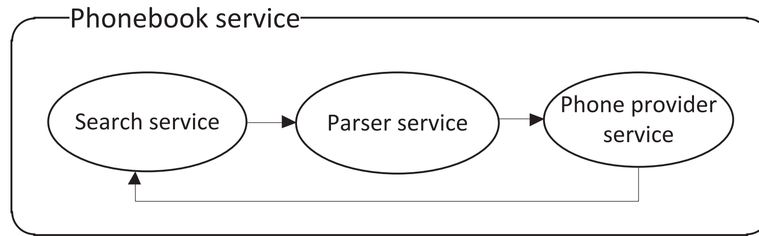


Fig. 5.1: Phonebook Service

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.

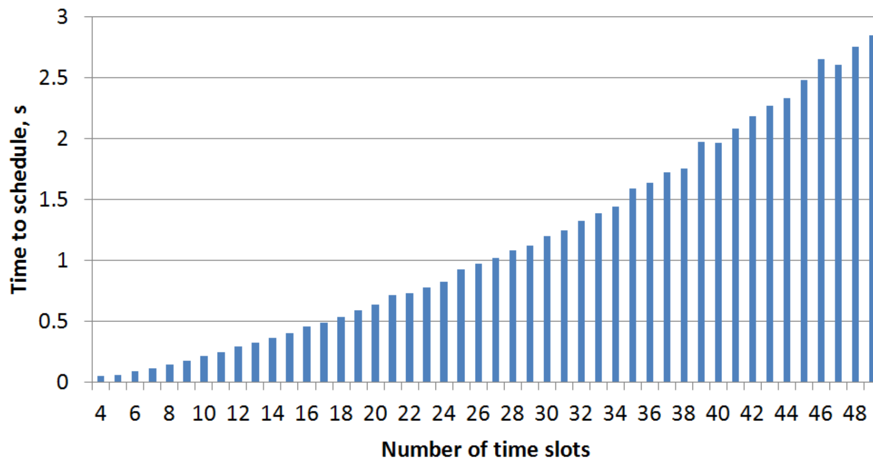


Fig. 6.1: Scheduler performance based on the number of time slots.

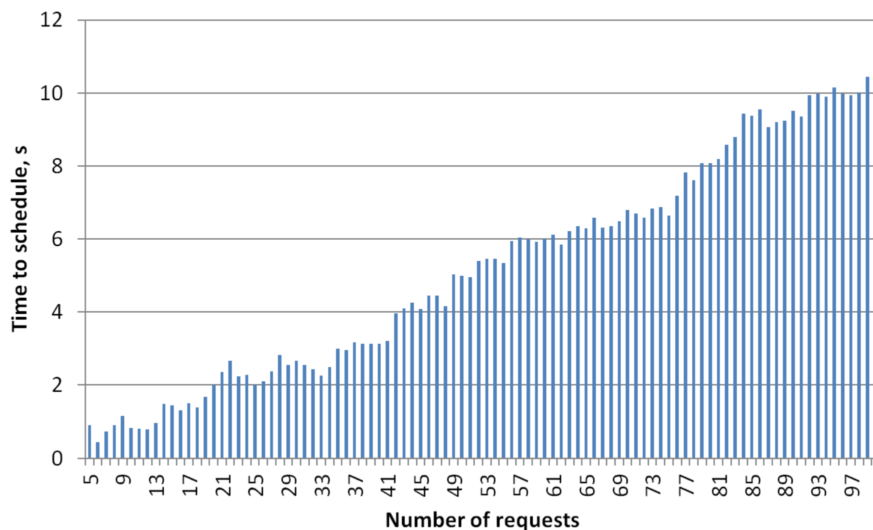


Fig. 6.2: Scheduler performance based on the number of requests.

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

Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 201–213. http://www.scpe.org

ISSN 1895-1767
© 2012 SCPE

CHALLENGES FOR THE COMPREHENSIVE MANAGEMENT OF CLOUD SERVICES IN A PAAS FRAMEWORK

SERGIO GARCÍA-GÓMEZ [*], MIGUEL JIMÉNEZ-GAÑÁN [†], YEHIA TAHER [‡], CHRISTOF MOMM [§], FREDERIC JUNKER [¶], JÓZSEF BIRO [‖], ANDREAS MENYCHTAS [**], VASILIOS ANDRIKOPOULOS [††] AND STEVE STRAUCH [‡‡]

Abstract.
The 4CaaSt project aims at developing a PaaS framework that enables flexible definition, marketing, deployment and management of Cloud-based services and applications. This paper describes the major challenges tackled by 4CaaSt for the comprehensive management of applications and services in a PaaS. These challenges involve the blueprint language to describe applications in the cloud and its lifecycle management, as well as a one stop shop for Cloud services and a PaaS level resource management featuring elasticity and advanced Network as a Service capabilities. 4CaaSt also provides a portfolio of ready to use Cloud native services and Cloud enabled immigrant technologies. The evaluation process followed to assess 4CaaSt progress is also described.

Key words: Cloud, Platform as a Service, Service Composition

AMS subject classifications. 68M14, 68U35

1. Introduction. Cloud computing is transforming the way applications and services are created, provided and consumed. The virtualization of infrastructures has lowered the barriers of entry such as cost and provisioning time for many providers, especially SMEs [1] [2]. Through the virtualization of platforms, SMEs can compete in an almost equal basis with the established players [2].

The European Union-funded 4CaaSt project[1] aims to create an advanced PaaS implementation, which supports the optimized and elastic hosting of Internet-scale, multi-tier applications and enabling the creation of a true business ecosystem [7]. Applications coming from different providers can be tailored to different users, integrated, mashed up and traded together.

In [3], the 4CaaSt value proposition is highlighted as:

- A higher level of abstraction regarding applications and services deployment, hiding the operational complexity while providing a resource efficient solution.
- A broad set of built-in programming libraries, building blocks and specific functionalities, as well as common facilities beyond what is offered and fostered by State-of-the-Art PaaS Clouds, easing development of killer applications showing the value of the 4CaaSt platform.
- An attractive business ecosystem supporting facilities to promote and monetize applications and services, as well as create an active community of users, providers and developers.
- The necessary tools to monitor the execution and manage the lifecycle of applications.

This paper focuses on the description of the research challenges that are being tackled in the 4CaaSt project, highlighting the main benefits for service developers and providers and explaining how this benefits are going to be assessed. The 4CaaSt platform revolves around the innovative concept of blueprint, an abstract description of an application or service that decouples what they offer from the resources required from the various layers of the Cloud stack. The blueprint leverages a great flexibility for the creation, deployment and marketing of applications and services in the Cloud.

The paper is structured as follows. The different usage models supported by 4CaaSt are presented in Section 2. Section 3 describes the most important innovations of the project. Section 4 explains the 4CaaSt validation scenarios and a simplified example to illustrate 4CaaSt platform process, and finally Section 5 summarizes the conclusions and the most important benefits to be obtained from the platform.

[*] Telefónica Digital, Spain (sergg@tid.es).
[†] Universidad Politécnica de Madrid, Spain (mjimenez@fi.upm.es)
[‡] ERISS, University of Tilburg, The Netherlands (Y.Taher@TilburgUniversity.edu)
[§] SAP Research Center Karlsruhe, Germany (christof.momm@sap.com,)
[¶] University of St. Gallen, Switzerland (frederic.junker@unisg.ch)
[‖] Nokia Siemens Networks, Budapest, Hungary (jozsef.biro@nsn.com)
[**] National Technical University of Athens, Greece (ameny@mail.ntua.gr)
[††] University of Stuttgart, Germany (steve.strauch@iaas.uni-stuttgart.de)
[‡‡] University of Stuttgart, Germany (vasilios.andrikopoulos@iaas.uni-stuttgart.de)
[1] http://www.4caast.eu/

2. Usage Models. According to the most cited architectural concepts for Cloud computing, Platform as a Service is an important part of Cloud computing architecture. PaaS represents the middle layer connecting the IaaS and the SaaS layer, see for example [5] [4] . However, this reflects a very simplified view on Cloud architecture. While 4CaaSt concentrates on the Platform as a Service layer of the Cloud stack, the way in which the project deals with the combination of services from the different layers, benefits from an analysis of how different roles can use them. First of all, the following roles are identified:

Service Provider , who markets deployed services based on existing software, e.g. for SaaS, PaaS, IaaS, and any other XaaS layer.

Software Provider , who provides new application software or platform (middleware) ready to be deployed on the IaaS/PaaS layer.

Customer , who contracts a software to be deployed on Cloud resources, or any service in general (SaaS, PaaS, IaaS).

Beyond those external users, it must be taken into account that there is also a Cloud platform manager. The different cloud stack layers can be used and combined in different ways by the roles specified above, leading to an abundance of deployment scenarios for applications and services over Cloud Computing resources.

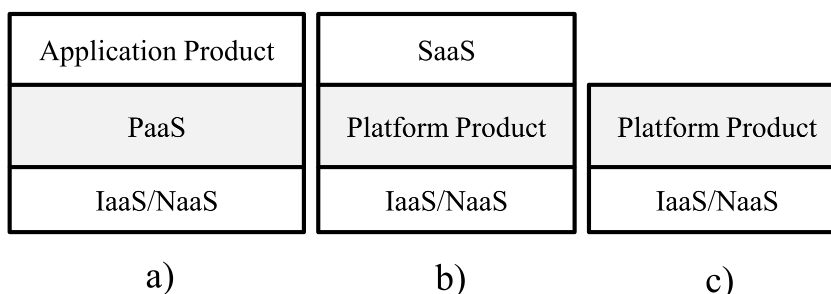


Fig. 2.1: Examples of cloud usage models.

Figure 2.1 shows three examples of scenarios of such deployments. *Application over a platform (a)*: A software provider can develop an application (e.g., a content management system) and publish it so that it can be deployed using some platform resources (a web container, a RDBMS) offered as a service (PaaS). *SaaS over a platform (b)*: A service provider can develop and/or deploy an application (e.g.: a billing application) on top of several platform products (deployed over IaaS resources) to offer a service to any external customer that contracts it (SaaS model). *Pure platform over IaaS (c)*: A development company can contract several platforms (e.g. a web container, a RDBMS, etc.) to be deployed over an IaaS and use them for their own developments. They can also contract PaaS services (e.g., a Non-SQL Data Store) as part of the development/deployment environment of their applications.

4CaaSt concentrates on the usage models that make use of the Platform Software and the PaaS, providing mechanisms to support the application software and SaaS lifecycles when deployed over the 4CaaSt platform. That includes the description of applications and their dependencies on the platform and infrastructure layers, the decisions about which specific resources to use, the deployment and configuration of platforms and application components and their management and lifecycle, the monitoring of the different layers the application consist of, and on top of that, the services required to trade with all of these usage models through appropriate business models.

Since the automatic provisioning and management of resources is one of the key features of Cloud Computing, managing the end-to-end lifecycle of any of those scenarios is a challenge. The innovative concept of the 4CaaSt blueprint, described in the following section, in combination with the marketplace functionality and the integrated management of software, services and resources, enable the reification of many usage models and business models as described above. This flexibility constitutes a key differentiator of the 4CaaSt platform in relation to major competitors both from the market and academia.

3. The 4CaaSt Innovations.

3.1. Blueprints. In 4CaaSt, every application, service or component is described by a blueprint, a description that specifies the various aspects that are linked to such resource and are required to manage its lifecycle, set up the runtime environment, and support the business transactions. Cloud Blueprinting is a powerful solution that aims at providing next-generation software developers with significant methods and tools that enable them to easily aggregate, configure and deploy virtual service-based application payloads on virtual machine and resource pools on the Cloud [8] .

The long-term benefits of Cloud Blueprinting will address concerns at the heart of the Enterprise of the Future and global service marketplaces by:

- Enabling novel geography spanning, end-to-end service applications to be built.
- Encouraging innovation through novel integrative service/Cloud development.
- Empowering service developers to better meet changing application requirements and develop customized service applications.
- Allowing new, innovative business models to be developed through the use of on-demand service platforms, infrastructure and supporting services.

To achieve its aim, Cloud Blueprinting promotes autonomous services at all levels of the Cloud stack that adhere to the same principles of separation of concerns to minimize dependencies. This solution allows any service at any layer to be appropriately combined with a service at the same level of the Cloud stack or swapped in or out without having to stop and modify other components elsewhere. At the same time Cloud Blueprinting allows multiple (and possibly composed) resource/infrastructure or implementation options for a given service at the application-level. This enables forming service aggregations on demand at any level of the Cloud stack that may potentially involve various SaaS/PaaS/IaaS providers by breaking up the current SaaS/PaaS/IaaS monolithic approach.

After having studied relevant literature [16], a proliferation of solutions for Cloud service development has been observed [10] [11] [12] [13] [14]. But, such methods have clearly shown considerable shortcomings to provide an efficient solution to deal with important aspects related to Cloud service-based applications. Some of these aspects are the elasticity and multi-tenancy of SaaS applications used to compose service-based applications. Current Cloud service offerings are often provided as a monolithic one-size-fits-all solution and give little or no opportunity for further customization. As a result, these stand-alone Cloud service offerings are more likely to show failure in meeting the business requirements of several consumers due to a lack of flexibility and interoperability.

The Cloud blueprinting approach introduces a series of Blueprint templates used to abstract and describe the components of Cloud Blueprinting-based applications. The use of templates provides a fast and simplified method for provisioning and automating Cloud services. It can be seen as a way for providing an understanding of the features used to deliver reliable and scalable Cloud deployments, and achieving better interconnection between physical and virtual infrastructures. To better manage Blueprint templates, the Blueprint framework interlaces several inter-related components [9]: (1) a declarative Blueprint Definition Language (BDL) that provides the necessary abstraction, constructs to describe the operational, performance and capacity requirements of Cloud services; (2) a Blueprint Constraint Language (BCL) that specifies any explicitly stated rule or regulation that prescribes any aspect of Cloud service; (3) a Blueprint Manipulation Language (BML) which provides a set of operators for manipulating, comparing and achieving mappings between blueprints, and (4) a simple Blueprint Query Language (BQL) for querying collections of blueprints.

The Blueprint model helps managing services when they transit through lifecycle stages: design, deployment, testing and monitoring. As illustrated by 3.1, after a provider has created the components of a service, the software provider begins the process of making it available to Cloud consumers by creating a source Blueprint model that defines the content of, and the interface to the service. Initially, during design each provider describes all relevant aspects of an offered service in a structure called a source blueprint. A service provider (might be distinct from the software provider) customizes the source blueprint templates to create a service offering for consumption by one or more consumers.

During design, an interim *target blueprint* model is created by combining a set of source blueprint models that a developer has selected. Combining source blueprints to satisfy the functional and non-functional requirements of the target blueprint relies on the blueprint resolution technique. A Cloud service developer normally starts designing a new and unresolved target blueprint that captures his to-be services. During the resolution process, in order to fulfill all resource requirements in the target blueprint, he relies on the offerings of other available third-party source blueprints that can be queried and purchased from the marketplace. The

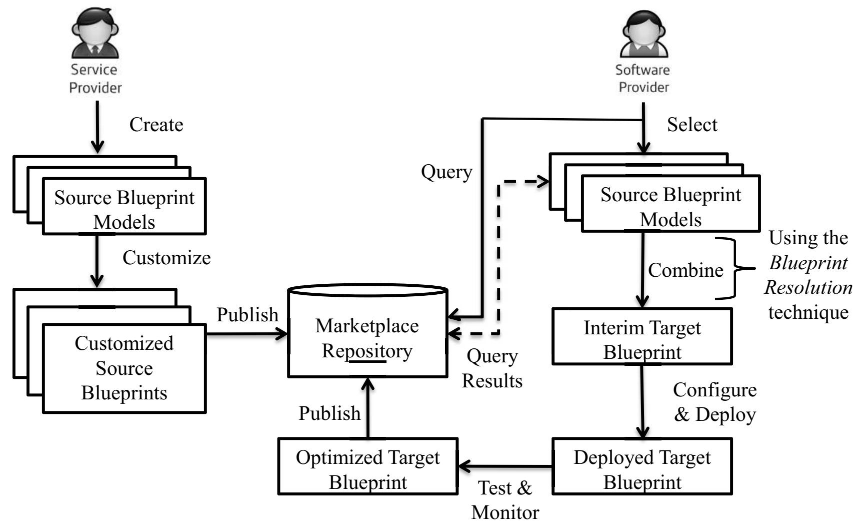


Fig. 3.1: Blueprint support for the Cloud service lifecycle.

blueprint resolution is an iterative process that ends up with a set of blueprints that fulfill all the requirements and constraints needed for actual deployment; the whole solution is referred to as *abstract resolved blueprint*. Subsequently, a deployment plan with configurability points is generated. This plan drives platform resources and virtual machine placement and network configuration.

3.2. Cloud eMarketplace. 4CaaSt provides a cloud *One Stop Shop* marketplace that supports the trading of all types of XaaS (SaaS, PaaS, etc.) services in a unified way, via a single point of access both for consumers and providers of services. This uniform specification of commercial offerings for any type of service is enabled by the marketplace's tight relationship with the 4CaaSt service engineering layer (by means of blueprints). Based on the model depicted in fig. 3.2, the 4CaaSt marketplace manages all phases of publishing and purchasing a service: information of products and stakeholders; negotiation and resolution of products; contracting and settlement of services; money flows; and analytics. Furthermore, the marketplace is tightly integrated with the 4CaaSt platform, so information sources can be leveraged by the marketplace. For instance, monitoring data from the service execution can be used for market analysis purposes.

The 4CaaSt marketplace may support different usage models, as discussed in Section 2: service providers can contract platform resources and even third-party software in order to provide and provision their own services, either by themselves or through the 4CaaSt marketplace.

The 4CaaSt infrastructure supports multiple contracting models: on one hand, customers can contract access to a public instance of a service, which is used by multiple customers and runs already before customers start using it; on the other hand, software developers can enable applications to be deployed on demand. Customers can contract a private instance of an application that will be deployed once the contract has been established.

The concept of the blueprint and its lifecycle allows 4CaaSt to provide fully-automated support for these different types of trading software, services and resources. This variety of deployment and contracting models makes software and service providers more flexible in implementing and applying diverse business models in 4CaaSt.

The 4CaaSt platform providers themselves can thereby develop dynamic business ecosystems. As a result, an intensified emergence of innovative applications for end- and corporate users can be expected on the 4CaaSt marketplace.

Service providers can define the price models for their services, which are then stored in machine-readable documents for automated processing (based on the Universal Service Description Language, USDL). When customers start using a service, their quantitative consumption is monitored by the 4CaaSt infrastructure, so

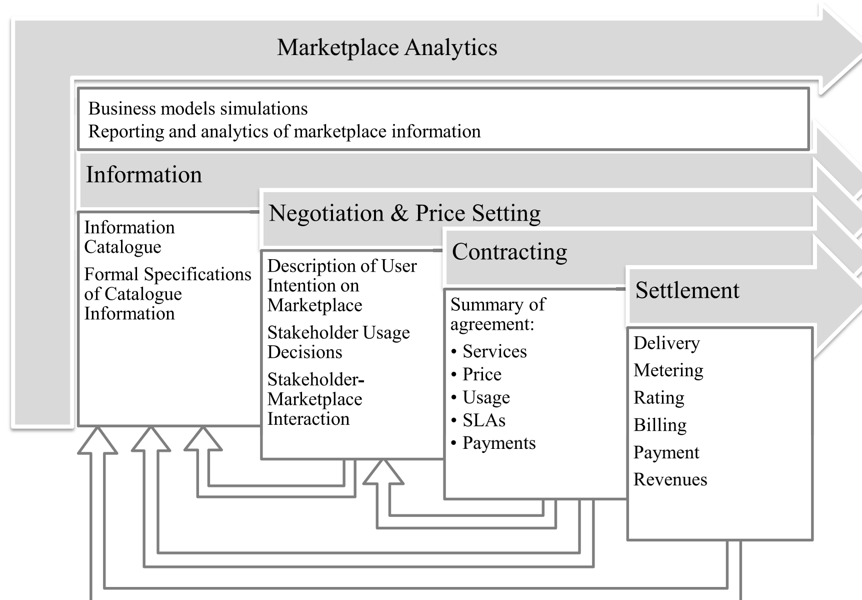


Fig. 3.2: Marketplace Processes.

the marketplace can automatically compute billing amounts and issue bills. Despite their machine-readable design, price models are very versatile and allow service providers to define the most appropriate price models, including subscription, pay per use, revenue sharing and advertisement-based models. The price model of a service can be defined either in advance or dynamically after the *blueprint resolution* by considering the aggregated cost the provider incurs by using third-party services.

The 4CaaSt marketplace also includes a feature called *business resolution*, which allows customers to select the most appropriate service for their needs. In addition to the technical requirements resolved by the blueprint resolution, the customer can specify business requirements such as business model, cost or workload. The solutions fulfilling the customer's technical requirements are customized and evaluated by the individual customer's business requirements. In this process, advanced methodologies are applied that combine the customer request and the available service offerings with various information sources such as data on market analysis customer behaviour. The quality of each solution is examined with the marketplace's analytics services and the outcomes are exploited to fine-grain future resolutions and selections.

In addition, the agents operating on the 4CaaSt marketplace, i.e. buyer and seller agents, can communicate via social tools. The data originating from this interaction is called *social data*, which describes the relationship between the agents on the marketplace. Social data is recorded and analysed to enable *socially enhanced market analysis (SEMA)*. The goal of SEMA is to employ social data to more accurately *(a)* forecast demand for given products and functionalities, and *(b)* predict the behavior of other market participants in the eMP. Market analysis tools without social enhancement are typically inaccurate, e.g. due to a lack of high-precision data input for analysis. Therefore, social enhancement can support or even replace the statistical estimation approaches currently employed by market analysis tools used by electronic marketplaces. To enable SEMA in the first place, we identify the characteristics of social data emerging in business social environments and define procedures for its acquisition, quality assurance, preparation and analysis. Analysis methods include:

- measuring the importance of individual nodes in the social graph of agents operating on the 4CaaSt marketplace.
- approaches from artificial intelligence to perform *plan recognition*, i.e. predicting the actions of agents based on partial knowledge of their environment.

Beyond SEMA, further *social enhancements* can be explored and implemented in the future, which are used to *(a)* improve features already existing in other cloud marketplaces, and *(b)* enable entirely new features, which could not exist before. In particular, these social enhancements include:

1. Socially enhanced search
2. Socially enhanced ratings
3. Socially enhanced user adaptation

These social enhancements are employed within the B2B and B2C areas. The C2C area is not covered, particularly with respect to *ratings* and *consumer adaptation*. Social data and social enhancements in C2C are out of the scope of this seminar paper as per the research objectives of the 4CaaSt project.

After analysing the most prominent Cloud services marketplaces (Windows Azure Marketplace, SuiteApp, Zoho Marketplace, Google Apps Marketplace, Google Play (formerly known as Android Market), and Force.com AppExchange), it has been realised than no other platform supports the full suite of functionalities envisioned by the 4CaaSt integrated marketplace outlined above [15]. The most comprehensive offering is provided by Force.com, although it only allows the trading of applications based on their exposed development APIs. Furthermore, it has been detected a trend for IaaS providers to include PaaS capabilities in their offerings, or SaaS providers to enable the application development over their platform [17] . However, no other platform supports in the same way a combined offering of services in the different levels of the Cloud stack. This can therefore be considered as a unique feature of 4CaaSt as compared to other Cloud marketplaces.

3.3. PaaS Deployments and Elasticity. Current PaaS offerings, like the Google App Engine, Amazon Beanstalk, Force.com and Windows Azure rely on a dedicated, homogenous set of infrastructure resources and middleware components. They are able to automatically provision resources but are limited to their choice of technology. 4CaaSt adds support for automatically generating and executing different elastic PaaS Deployments using different middleware and infrastructure components/services. This approach works as follows:

1. **Blueprint creation + resolution.** 4CaaSt component or service providers have to create a blueprint that includes all information to perform the deployment design generation and execution. In addition to the basic blueprint content this includes: (a) Definition of elasticity constraints for each artifact, in particular whether horizontal and/or vertical scalability are supported in principle. (b) Optional definition of elasticity rules for the blueprint based on monitoring KPIs + adaptation actions defined for blueprint itself or required blueprints. To define these rules we use the open Rules Interchange Format (RIF) [31]. (c) "Hints" defining quality/capacity requirements of required blueprints for achieving certain qualities, e.g. 1000 4CaaStAppServerPowerPoints required for serving 100 users in parallel with good responsiveness. (d) References to Chef-based Installation + configuration scripts for the included artifacts to enable an automated installation of the software stack. Using the 4CaaSt marketplace the customer selects a blueprint and defines his non-functional requirements (based on the hints). The blueprint resolution described before then resolves all functional dependencies by creating the abstract resolved blueprint (ARB).
2. **Deployment Design Generation.** An ARB can be considered as structured bill of material defining the required components and services for a solution. However, an ARB still leaves many degrees of freedom how these sets of components are concretely deployed, e.g. put all components on one single virtual machine (VM) or create a distributed, elastic deployment. The deployment design generation step therefore accounts for creating the best deployment design for a given ARB based on the specified customer requirements, i.e. the design that fulfills the customer requirements with a minimum set of resources. If a solution should support a certain workload range (e.g. 100-1000 users) these designs may be elastic, i.e. supporting a vertical or horizontal scaling of certain nodes. However, a prerequisite for this is that elasticity rules are available and the given elasticity constraints permit such a scaling. Since it is possible to define a set of (consistent) elasticity rules for each blueprint, it may happen that for the ARB more than one set of rules is available. In this case we use the rules defined for the "highest" blueprint in the ARB tree, e.g. the application component.
3. **Resource Provisioning.** The final deployment designs are captured using extended OVF [30] specifications (OVF++), which contains all information for "executing" the deployment, in particular the VM specifications, the list of "products" that have to be installed on the VM including deployment scripts and the selected elasticity rules. Using this information, first all required VMs are created, then for each VM the specified software stack is installed and configured (including monitoring!) and finally the machines are started.
4. **Service Operation + Adaptation.** Having completed the provisioning process, all VMs are running within the specified infrastructure cloud, e.g. Flexiscale. In addition to this, the services are registered

in the 4CaaSt runtime infrastructure, which supports monitoring, accounting as well as elastic scalability for enforcing certain quality constrained using the predefined elasticity rules.

Fig. 3.3 provides an overview to the architecture implementing this approach.

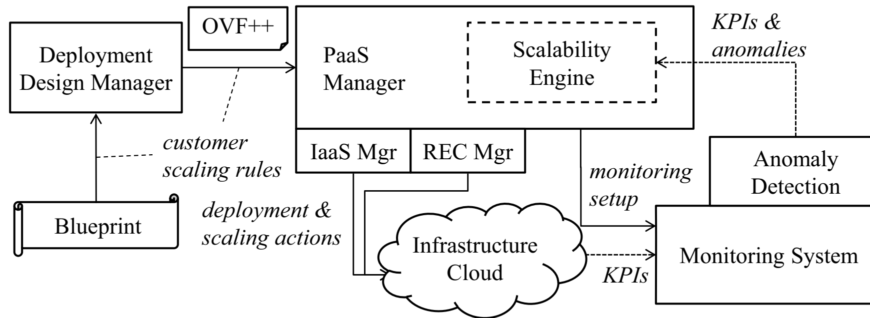


Fig. 3.3: Architecture of the PaaS deployment and elasticity mechanism.

The whole deployment design generation step is performed by the Deployment Design Manager Component, while the PaaS Manager accounts for both the resource provisioning and the elastic scaling during service operation. For the resource provisioning the PaaS Manager first interacts with a IaaS Manager for creating all required VM. After that, the so called Runtime Execution Container (REC) Manager is used to configure the individual software stacks within the machines. To this end, each machine per default includes an agent that can be remote controlled by the REC Manager. The implementation of this infrastructure is based on Chef [29]. To perform the automated elasticity the PaaS Manager includes a scalability engine, which is configured with the rules and actions defined in the OVF++. The necessary KPIs are thereby delivered through the 4CaaSt Monitoring System. This requires the configuration of monitoring probes within the different VMs, which is handled by the REC Manager during the resource providing phase. In contrast to existing solutions, the PaaS Manager supports scaling in the PaaS layer, using KPIs not only from the IaaS layer (CPU usage, free memory, etc.) but also from the PaaS layer (number of transactions per second, number of tenants, etc.). In this way, it is possible to provide PaaS architectural vertical and horizontal scaling, by decoupling middleware and components from the virtual machines in which they are hosted and deploying scalable architectures (load balancers, shared/not-shared execution containers, stateless components replication, etc.).

3.3.1. Example. In the following we explain how the approach works based on a simplified sample scenario, namely a web app that requires a Tomcat servlet engine and a MySQL database. Figure 3.4 illustrates how the blueprinting and the deployment design work for this scenario.

The “hints” used for the sizing are relations defined on a discrete set of qualities or attributes. What needs to be underlined is that 4CaaSt defines a strictly layered requirements translation approach based on blueprints. This means that every component that is being described by a blueprint, allows only requirements of the same layer or directly depending ones to be defined as hints. In the example mappings are defined between each of the offered workload values (100, 1000, 3000) and the required DB and app server (AS) capacity defined in terms of 4CaaStDBPoints / 4CaaSTASPoints, e.g. 100 Users require 100 DB and 100 AS points. In addition to this the hints defines the effect of adding more instances of a blueprint, e.g. linear behavior: Two 100 point DB / AS instances would serve 200 Users. Whether it is possible to create multiple instances depends on the elasticity constraints. In the example, the database is not scalable. Thus, as soon as the database is part of a node, there can only be 1 instance of it. For this reason, the minimal deployment design comprising only one node is not sufficient for satisfying the customer requirements. Instead, the deployment design manager proposes the maximum elastic design with arbitrary many Tomcat instances and one DB instance (due to the elasticity constraint). The initial sizing thereby is determined based on the available hints, which in this case would be 2 small Tomcat instances and one big database instance. If the workload changes at runtime, the elasticity engine executes the available rules. For the example, only rules for the Tomcat blueprint are defined as shown on Figure 3.4. So there is no need for choosing between several alternative rule sets. This Tomcat

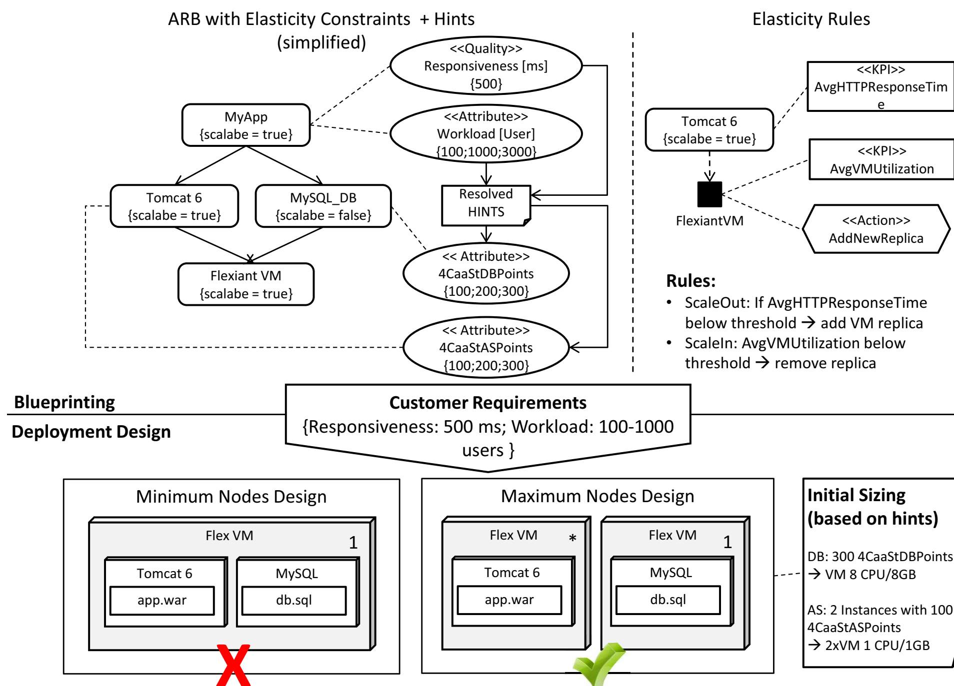


Fig. 3.4: Blueprinting and Deployment Design by Example.

rule set is included in the OVF++ and can be executed by the PaaS Manager. To install the corresponding monitoring probes, we assume a Chef-based installation/configuration script is referenced in the blueprint for every KPI.

3.4. Network as a Service. NaaS (Network as a Service) is a very popular emerging topic in the field of cloud computing. Consequently, the term is extremely overloaded, covering a wide range of networking related services and a variety of underlying technologies. The meaning of the term applied in 4CaaSt context is derived from the classic NIST definition of Infrastructure as a Service, according to which infrastructure is understood as the trio of computing, storage and networking resources. Yet, typical state-of-the-art IaaS services still focus more on computing and storage than on networking. Therefore, the major 4CaaSt objective concerning NaaS is to emancipate networking resources in a typical IaaS portfolio and give more control to clients over networking options.

Early IaaS offerings provided very simple L3 networking options: single connection to the internet with automatically preallocated IP addresses and no possibility of separated networks. These were sufficient for simple applications but not for more complex enterprise or telco application suites, formed by a set of closely or loosely cooperating VMs, possibly in multiple tiers, possibly at different data centers. Admittedly, the cloud networking scene is evolving rapidly today with various more advanced offers both at L3 and L2 (support for multiple virtual networks, secure connections towards customer premises, some control over addressing, etc.), but further work is needed to create a mature and sufficiently feature-rich networking environment.

4CaaSt tries to contribute to the ongoing networking evolution by addressing the following important aspects. The first requirement is support for network quality, i.e. the ability (of the customer) to request quality attributes of the required networking services. This requires fine-grained control over the network elements forwarding the traffic. The second requirement is the integration of the network control subsystem with the cloud control subsystem and the presentation of the networking options in the IaaS APIs. This requires a definition of a well-thought, modular architecture with interfaces providing well-thought future proof networking abstractions, allowing the continuous seamless adaptation of new networking technologies. Finally 4CaaSt also aims to integrate the NaaS offer vertically into its PaaS concept. Network resources can be requested/offered in blueprints and the resource management layer will deploy/control networking resources in an orchestrated

way alongside with other IaaS and PaaS resources.

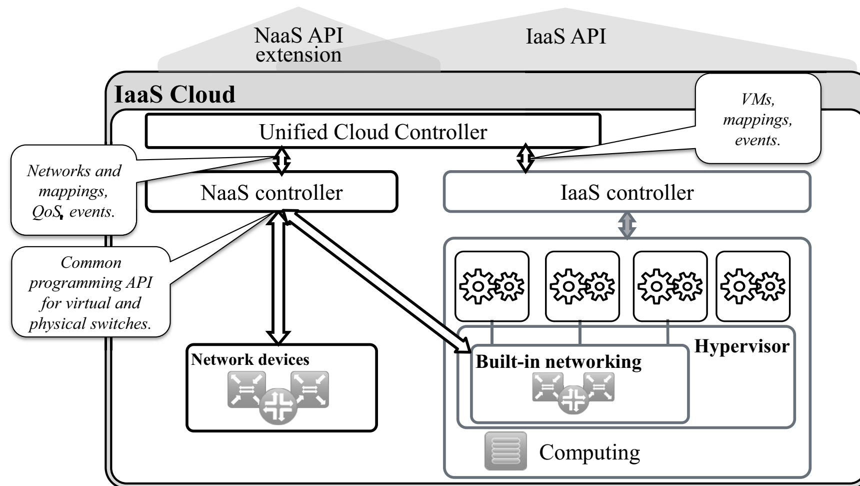


Fig. 3.5: Network as a Service Architecture.

In order to address the aforesaid requirements 4CaaSt proposes a new NaaS Manager component. This component aims to take over (fully or partially) the network management responsibility from other existing entities such as Data Center network operators or IaaS. NaaS Manager will support programmable control over the network and provide better information about the network state. Furthermore it will provide new features like traffic Engineering, QoS, multicast, anycast, etc.

NaaS Manager will provide a northbound interface towards the 4CaaSt resource management layer for the management of networking resources belonging to the deployed/managed 4CaaSt applications/entities. Its southbound interface will directly access the network elements (both physical and virtual) forming the local network under control. The architecture will enable flexible deifinition of networks boundaries and consider the requirement of close but modular integration of the NaaS Manager with different existing cloud manager platforms and possibly with peer network managers. Thus, The 4CaaSt NaaS Manager will implement the Software Defined Network paradigm for Data Center networks and integrate this concept with the 4CaaSt Paas paradigm.

3.5. Service Enablers. Service Enablers represent a common view of the integrated services inside the platform, allowing the Cloud platform to manage such services in a homogeneous way. Service Enablers have been defined so as to provide in a homogeneous way heterogeneous services, and are used to provide both Immigrant and Native PaaS technologies. Management of the services spans different actions, namely deploying the components and provisioning the services after a purchase, monitoring the correct working of the services, and accounting the usage of resources, so as to charge for them in the marketplace. Since each of these actions poses different requirements, different mechanisms and technologies have been used.

Service provisioning and deployment are built using Chef recipes, which are scripts in a domain specific language based on Ruby. Recipes are created by technology owners so as to enable the component deployment, configuration, and provisioning. Pre-defined sets of recipes, called cookbooks, are created by technology owners and invoked by REC Manager so as to provision new tenants in the services and deploy technology stacks.

Service monitoring is based on two different monitoring technologies, currently JMX and collectd, for which collectors have been created. Each technology component implements monitoring probes providing several Key Performance Indicators (KPIs) as declared in its Blueprint, and the same deployment technology installs the probe providing the needed KPIs.

Service accounting is done in an active way, providing the Marketplace with periodic reports about the resources used by a specific application. These reports, called Service Detailed Records (SDRs), provide information about the concepts declared in the price model of the service, so the Marketplace can charge the customer.

There is also a Software as a Service provisioning API used to notify an application offered as SaaS in the marketplace, the addition of removal of a client allowed to invoke that service. Those applications must provide a simple RESTfull interface so as to notify the users allowed to use the service. Basic authentication processes occurs behind the scenes so that the user purchasing a service can then log on the purchased service and start using it wit her marketplace credentials.

Native Service Technologies include technologies developed specifically for the Cloud or that happen to be very suitable for it; therefore they are likely to be involved in Cloud-oriented applications. 4CaaSt will offer several native Cloud technologies as services, such as Context aaS, Network Enablers aaS or Cloud Data Store Capabilities that can be used and deployed in an on-demand fashion according to a client's workload and requirements. These technologies are packaged as services offered by 4CaaSt in a uniform fashion, following the concept of technology enabler described above, and thus offering a common interface used by the platform in different phases of applications lifecycle, namely deployment, configuration, monitoring, management and billing.

3.6. Immigrant PaaS technologies. Immigrant PaaS technologies refer to a set of tried and proven technologies that were available before the advent of Cloud computing and now need to be adapted for the new era. These technologies form a series of building blocks for the 4CaaSt platform and can be used on demand, either in conjunction or independently, when building applications and services based on the 4CaaSt platform. 4CaaSt focuses on the provisioning of infrastructure components for composite applications and services, and in particular on providing Cloud-ready databases, application servers and service composition engines, working when required in tandem through an integration layer that handles the communication with services external to the 4CaaSt platform.

Our investigation of the State of the Art showed that existing Cloud solutions for data storage, application hosting, service composition and integration are either immature or focused exclusively on one feature like availability [18]. Furthermore, merely deploying existing non Cloud-native technologies on the Cloud as part of VM images fails to utilize the full potential of the Cloud computing paradigm in terms of on-demand elasticity and scalability on PaaS level [27]. For these reasons, in 4CaaSt we look into how these technologies can be instead immigrated to the Cloud, adding to them mechanisms and capabilities in order to make them Cloud-aware, offering out-of-the-box capabilities like multi-tenancy, scalability and PaaS-oriented management and configuration. These technologies can then be provided as installable software, or as services, similar to the Native Service Technologies.

Cloud-enabling existing technologies creates a number of research and implementation challenges to be overcome. An example of such challenges is the case of immigrating a critical middleware component like the Enterprise Service Bus (ESB) [20] to the Cloud. The concept of ESB as the messaging hub between applications addresses the fundamental need for application integration and in the last years it has become ubiquitous in enterprise computing environments. ESBs control the message handling during service invocations and are at the core of each Service Oriented Architecture (SOA) [20]. Enabling, for example, multi-tenancy at the ESB level allows multiple service consumers to use the same application offered as a service by a provider in a fully costumizable per consumer manner, both on the level of tenants, and that of users belonging to a particular tenant. Apart therefore from allowing organizations to outsource the development, deployment, operation and management of such applications to a service provider, this solution also maximizes the benefits on the provider side.

Making an ESB solution multi-tenant however requires fulfilling the following requirements [28], as identified among others by [21], [25], [24]:

- providing for *tenancy awareness*, i.e. tenant-based identification and hierarchical access control for tenants and their users,
- offering *tenant-based facilities* like management interfaces providing tenant- and user-specific deployment and configuration options, and
- ensuring *tenant isolation* and *security*, making sure that no tenant data and computational resources can be accessed by other tenants or unauthorized third parties.

In order to satisfy these requirements, in 4CaaSt we extend the open source ESB solution Apache ServiceMix [19] which is based on the OSGi Framework [26] implementing the JBI specification [22] as discussed in [28]. Beyond extending the ServiceMix components responsible for processing and transforming messages to support multi-tenant aware message exchanges, we also implemented an OSGi-based management service and the respective

Web GUI and Web Services API offering tenant-based deployment, configuration and administration of service endpoints. As the management components of the underlying resources are implemented as EJB components, we use container-managed transaction demarcation, which allows the definition of transaction attributes for whole business methods, including all resource changes [23]. A performance evaluation of our Apache ServiceMix extension is currently being performed, with encouraging results.

Given the disparity of technologies involved (databases, application servers, orchestration engines and integration technologies), the extracted requirements for enabling multi-tenancy and scalability demand a significant amount of work in order to fulfill the goals of the project.

4. Validation. 4CaaSt requirements and business view are being validated through three different scenarios. These three scenarios are implementing application prototypes which make use of some of the native cloud services and immigrant technologies provided in 4CaaSt, and are being integrated with the overall 4CaaSt platform in order to evaluate the different features provided. Below, a description of the three scenarios, together with the validation proposal, is shown. Additionally, a simplified generic scenario is described in order to illustrate the most important steps in 4CaaSt lifecycle.

4.1. A marketplace for SME's applications. The first scenario aims at developing a web application (a taxi fleet management application) that is delivered on demand on cloud resources provisioned for the customer, i.e., the resources are booked, deployed, and configured so that the software stack and the web application can be properly deployed. This means that the application is not offered as a service (it is not a multitenant SaaS), but a COTS software that is automatically installed on the cloud.

This scenario is used in the current platform release to validate a number of important features:

- Marketplace products and price models definitions.
- Contracting and business resolution.
- Blueprint resolution.
- Flexible deployment on cloud resources.
- Immigrant technologies.

4.2. A marketplace for mass market applications. This scenario is based on the creation, deployment and offering of SaaS solutions over PaaS/IaaS resources, and how to sell them as services for the mass market. In this case, a multitenant tourism application is deployed and offered as a service to final users (tourists). In the current release, it is used for the validation of the following features:

- Blueprints definition.
- Price models definition based on accounting capabilities.
- Provisioning of services.
- Accounting, charging and settlement.
- Native services.

4.3. A hybrid cloud. This scenario implies the cloud bursting of resources at PaaS level of an application that is running on a private cloud. The evaluated features are:

- Marketplace offering and contracting.
- SLAs definition and deployment design and sizing.
- Flexible cloud deployment and elasticity management.
- Network as a Service.
- Advanced IaaS features.

4.4. A simplified scenario. In order to illustrate the value proposition of 4CaaSt platform and validate the proposed approach of the project, this section describes a simple scenario where the main actors are involved, and their interactions with the system depicted:

1. The *PaaS (4CaaSt) provider* wants to have multiple technologies (software or services) in a 4CaaSt platform so that developers have a wider range of technological options to develop their applications. For instance, Context as a Service or a PostgreSQL-based service. For each of them, he must provide a blueprint with the offering and the requirements, and publish it in the marketplace according to a price model.

   External *Service Providers* could follow the same approach in order to provide services in a 4CaaSt compliant Cloud Provider, enabling them to do business in the 4CaaSt ecosystem.

2. The *Software provider* creates the application using his preferred tools (e.g., Eclipse). When he wants to integrate an existing service, he can browse through the 4CaaSt One Stop Shop in order to get information and download client libraries or other artifacts. He could also contract virtual machines or platform services (like a Tomcat deployment) to deploy and test his own application. Once the application is ready, he can create and publish the application blueprints and artifacts.
3. The application is now available and the 4CaaSt platform has all the information required to deploy and manage it. A *Service provider* can create a commercial offer of the application in the marketplace. He selects the application blueprint and defines the price model and the application is ready to be contracted. Any *Customer* will be able to obtain his own instance of the application.
4. When the *Customer* logs into 4CaaSt marketplace, he can select one out of many existing application and services and customize it if such action is allowed for the service.
5. Given the functions, price, performance, availability, or even the customer profile, the *4CaaSt Blueprint Resolution process* decides the best combination of services (blueprints) to deploy or provision for this particular *Customer*. *4CaaSt platform* decides also about which Cloud resources to use, including computing, networking, platforms and services. The application is deployed and the *4CaaSt platform* starts monitoring and accounting its usage.
6. The monitoring information is used by the 4CaaSt platform to take runtime elasticity decisions according to the constraints defined by the *Customer*.
7. Finally, the *4CaaSt marketplace* charges the *Customer* and distributes incomes among the various *Service providers* involved, including revenue sharing policies when they are defined.

5. Conclusion. The 4CaaSt project aims at creating an innovative framework for creating, marketing, deploying and managing applications on the Cloud, both over platform products and platform as a service. 4CaaSt introduces the concept of blueprint, a technical description of an application or a service that decouples the various dependencies it has along the Cloud layers.

Thanks to the blueprint innovation and how applications and services are traded and provisioned/deployed, 4CaaSt may support multiple usage and business models, giving to software and service providers the flexibility to use the resources and services as they prefer.

As a summary, 4CaaSt allows software and service providers to focus on their business (both the software and the service monetization), leaving the underlying complexity of infrastructure and platforms out of their concerns. Among the benefits that 4CaaSt provides for practitioners or IT managers, the following ones can be highlighted:

- 4CaaSt provides a large portfolio of Cloud-aware, ready to use, mainstream technologies (ESBs, RDBMS, BPM engines, etc.)
- 4CaaSt offers an easy mechanism to integrate off the self-services (context, telco, mashups, etc.).
- 4CaaSt allows developers to decouple the development and specification of applications from their actual deployment.
- 4CaaSt enables the trading of applications and services based on multiple usage models and business models, providing integrated settlement of revenue flows among providers.
- 4CaaSt is capable of on-demand deploy, configure and provision software and services into virtual machines to offer applications ready to use, including complex configuration of the networking infrastructure.
- 4CaaSt scalability is not only based on adding more infrastructure resources, but also to vary the application architecture for a more efficient usage of resources.

To date, a first release of the platform has been delivered, implementing the main features for executing an end-to-end process, and the scenarios are being integrated and validated. Moreover, the focus of the future work will be in some of the most advanced and challenging features described in this paper, especially those related to PaaS elasticity and blueprint resolution, which will represent 4CaaSt unique value proposition.

Acknowledgments. The research leading to these results has partially received funding from the 4CaaSt project (http://www.4caast.eu/) from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 258862. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

REFERENCES

[1] Yefin V. Natis, Benoit J. Lheureux, Massimo Pezzini, David W. Cearly, Eric Knipp, and Daryl C. Plummer. *PaaS Road Map: A Continent Emerging*, Gartner (2011).
[2] IDC. *Market Analysis - Worldwide Public Application Development and Deployment as a Service: 2010 2014 Forecast.* (pp. 1 - 31), Framingham, MA, USA. 2010.
[3] Oliveros Díaz, E., García Gómez, S. *4CaaSt Value Proposition - A 4CaaSt Whitepaper.* (pp. 1-17) October 2011. Brussels. Available at http://www.4caast.eu.
[4] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Linder. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009.
[5] Peter Mell and Tim Grance, *The NIST Definition of Cloud Computing*, Tech. report, July 2009.
[6] Mitrabarun Sarkar, Brian Butler, and Charles Steinfield, *Cybermediaries in electronic marketspace: Toward theory building*, Journal of Business Research **41** (1998), no. 3, 215–221.
[7] James F. Moore, *Predators and prey: a new ecology of competition*, Harvard Business Review **71** (1993), no. 3, 75–86.
[8] Michael P. Papazoglou, Willem-Jan van den Heuvel. *Blueprinting the Cloud.* IEEE Internet Computing 15(6): 74-79 , 2011.
[9] Amal Elgammal, Oktay Tretken, Willem-Jan van den Heuvel, Mike P. Papazoglou. *Root-Cause Analysis of Design-Time Compliance Violations on the Basis of Property Patterns.* ICSOC 2010: 17-31
[10] Mietzner, R. *A method and implementation to define and provision variable composite applications, and its usage in Cloud computing.* Dissertation, Universitaet Stuttgart, Fakultaet Informatik, Elektrotechnik und Informationstechnik, Germany, August 2010.
[11] Galan. F, Sampaio. A, Rodero-Merino. L, Loy. I, Gil. V, Vaquero. L.M. *Service specification in Cloud environments based on extensions to open standards.* In: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE. COMSWARE '09, New York, NY, USA, ACM, 2009 19:1-19:12
[12] Chapman, C., Emmerich, W., Marquez, F.G., Clayman, S., Galis, A. *Software architecture definition for on-demand Cloud provisioning.* In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. HPDC '10, New York, NY, USA, ACM, 2010, 61-72
[13] Wei-Tek Tsai; Xin Sun; Balasooriya, J. *Service-Oriented Cloud Computing Architecture.* Seventh International Conference on Information Technology: New Generations (ITNG), vol., no., pp.684-689, 12-14 April 2010.
[14] Hyun Jung La and Soo Dong Kim. *A Systematic Process for Developing High Quality SaaS Cloud Services.* In Proceedings of the 1st International Conference on Cloud Computing (CloudCom '09). Springer-Verlag, Berlin, Heidelberg, 278-289, 2009.
[15] García-Gómez. S. (editor). *D3.1.1, Marketplace Functions: Scientific and Technical Report.* 4CaaSt Project Deliverable D3.1.1, 2011. Available at http://www.4caast.eu.
[16] Lelli. F. (editor). *D2.1.1, Blueprinting the Cloud: Scientific and Technical Report.* 4CaaSt Project Deliverable D2.1.1, 2011. Available at http://www.4caast.eu.
[17] Giessmann, A. (editor). *D9.1.2, Market and Competition Analysis.* 4CaaSt Project Deliverable D9.1.2, 2012. Available at http://www.4caast.eu.
[18] 4CaaSt Consortium, *Immigrant PaaS Technologies: Scientific and Technical Report D7.1.1.* Deliverable, July 2011.
[19] Apache Software Foundation, *Apache ServiceMix.*
[20] David A. Chappell, *Enterprise Service Bus*, O'Reilly Media, Inc., 2004.
[21] C.J. Guo, W. Sun, Y. Huang, Z.H. Wang, and B. Gao, *A framework for native multi-tenancy application development and management*, Proceedings of the 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'07), IEEE, 2007.
[22] Java Community Process, *Java Business Integration (JBI) 1.0, Final Release*, 2005, JSR-208.
[23] *Enterprise JavaBeans (EJB) 3.0, Final Release*, JSR-220, 2006.
[24] Rouven Krebs, Christof Momm, and Samuel Konev, *Architectural Concerns in Multi-Tenant SaaS Applications*, Proceedings of the 2nd International Conference on Cloud Computing and Service Science (CLOSER'12), SciTePress, April 2012 (Englisch).
[25] Ralph Mietzner, Tobias Unger, Robert Titze, and Frank Leymann, *Combining Different Multi-Tenancy Patterns in Service-Oriented Applications*, Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009), IEEE, September 2009 (Englisch).
[26] OSGi Alliance, *OSGi Service Platform: Core Specification Version 4.3*, 2011.
[27] L.M. Vaquero, L. Rodero-Merino, and R. Buyya, *Dynamically scaling applications in the cloud*, ACM SIGCOMM Computer Communication Review **41** (2011), no. 1, 45–52.
[28] S. Strauch, V. Andrikopoulos, F. Leymann, and D. Muhler, *ESBMT: Enabling Multi-Tenancy in Enterprise Service Buses*, in 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2012) - to appear, IEEE Computer Society, December 2012.
[29] Opscode Chef, http://www.opscode.com/chef/, 2011.
[30] Distributed Management Task Force *DMTF. Open virtualization format specification. Specification DSP0243 v1.0.0d. Technical report*, https://www.coinor.org/OS/publications/optimizationServicesFramework2008.pdf, 2008.
[31] W3C *Rule Interchange Format (RIF) Working Group* http://www.w3.org/2005/rules/wiki/RIF_Working_Group, 2010.

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

Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 215–231. http://www.scpe.org
ISSN 1895-1767
© 2012 SCPE

ON ENGINEERING CLOUD APPLICATIONS - STATE OF THE ART, SHORTCOMINGS ANALYSIS, AND APPROACH

YEHIA TAHER,* DINH KHOA NGUYEN, FRANCESCO LELLI, WILLEM-JAN VAN DEN HEUVEL, AND MIKE P. PAPAZOGLOU

Abstract. Recently, Cloud Computing has become an emerging research topic in response to the shift from product-oriented economy to service-oriented economy and the move from focusing on software/system development to addressing business-IT alignment. From the IT perspectives, there is a proliferation of methods for cloud application development. Such methods have clearly shown considerable shortcomings to provide an efficient solution to deal with major aspects related to cloud applications. One of these major aspects is the multi-tenancy of the Software-as-a-Service (SaaS) components used to compose Service-Based Applications (SBAs) on the cloud. Current SaaS offerings are often provided as monolithic *one-size-fits-all* solutions and give little or no opportunity for further customization. Monolithic SaaS offerings are more likely to show failure in meeting the business requirements of several consumers. In this paper, we analyze the state-of-the-art of the standardization, methodology, software and product support for SBA development on the cloud, identify some shortcomings, and point out the need of a novel approach for breaking down the monolithic stack of cloud service offerings and providing an effective and flexible solution for SBA designers to select, customize, and aggregate cloud service offerings coming from different providers [25].

Key words: Cloud Computing, Service-based Application (SBA), Service-oriented Architecture (SOA), Cloud Development Methodology

1. INTRODUCTION.

Software service technologies aim to support effective combination of independent software services, which are made available by diverse providers, into end-to-end service aggregations on a global scale and in much more powerful and innovative ways that respond to the needs of any kind of service consumers. Central to this aim is the concept of *Service-Oriented-Architecture* or *SOA* [29]. SOA is a philosophy of design that can be informally described as "the software equivalent of Lego bricks" where a collection of mix-and-match units (called "services") - each performing a well-defined task - can reside on different machines possibly under the control of a different service provider, and are ready to be used whenever needed. Enterprises typically use a single software service to accomplish a specific business task, such as billing or inventory control or they may compose several software services to create a value-added distributed Service-based Application (SBA) such as customized ordering, customer support, procurement, and logistical support.

The enterprise information system of the future will comprise of unbounded numbers and combinations of service eco-systems, which are network-structured, software-intensive, geographically dispersed, and have a global reach. A *service eco-system* is a system of systems [15], which depends on distributed control, cooperation, influence, cascade effects, orchestration, and other emergent behaviours as primary compositional mechanisms to achieve its purpose. The purpose, structure, and number of components in a service eco-system are increasingly unbounded in their development, use, and evolution. Service eco-systems support the development of end-to-end services and SBAs through the creation of alliances between service providers, through which each offered service can be used or syndicated with other services. For instance, an integrated logistics application, which entails the management of an entire logistics chain as a single application in the form of integrated services, could comprise many end-to-end services, such as procurement, order management, production planning and scheduling, inventory control, etc. Typical consumers of such service eco-systems are private organizations, public entities, user-communities, or any combinations of these. Flexible service infrastructures and technologies that can be used as the foundation for developing service eco-systems and applications are gradually providing the incentive and are paving the way for business and social innovation.

However, a serious limitation of an SOA is that it does not make any assumptions regarding service deployment and leaves it up to the discretion of the service developer to make this deployment choice, which is a daunting task and often leads to failure. A dangerous "*difficult-to-customize, one-size-fits-all*" philosophy permeates SOA development leading to brittle implementations where once an application is deployed it is bound to a particular infrastructure. In addition, traditional SOA-based application development concentrates on a kind of "*big design upfront*" where the prevailing belief is that it is possible to gather all of a developer's or customer's requirements, upfront, prior to coding a software solution. So despite its promises, SOA has so far failed to deliver promised benefits except in rare situations leading yet again to a software development crisis.

*European Research Institute in Service Science, Tilburg University, 5000 LE, Tilburg, The Netherlands, (Y.Taher@TilburgUniversity.edu).

To address these serious shortcomings it is normal to turn our attention to *Cloud Computing* as it aims to provide both the economies of scale of a shared infrastructure and a flexible delivery model that naturally complements the service orientation of the SOA paradigm. Cloud computing is a computing model for enabling convenient and on-demand network access to a shared pool of configurable and often virtualised computing resources (e.g., networks, servers, storage, middleware and applications as services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [28]. Cloud capabilities are defined and provided as services where users of cloud-related services are able to focus on what the service provides them rather than how the services are implemented or hosted. And this begins to explain why the service orientation provided by an SOA needs the "cloud" as a natural deployment medium. In fact, the two concepts can be paired to support service development and deployment and their merger can provide a complete services-based solution. Cloud computing is typically divided into three levels of hosting service offerings: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). These three levels support the virtualisation and management of different levels of the computing solution stack as follows:

- IaaS: is the delivery of hardware (server, storage and network), and associated software (operating systems virtualisation technology, file system), as a service. It is an evolution of traditional hosting that does not require any long-term commitment and allows users to provision resources on-demand. IaaS incorporates the capability to abstract resources as well as deliver physical and logical connectivity to those resources and provides a set of APIs that support the interaction with the infrastructure by consumers. The full power of IaaS can only be used if the flexibility of IaaS deployment and resource allocation has already been considered during the design and development of service-based applications; something that is not possible with today's IaaS approaches. Amazon Web Services Elastic Compute Cloud (EC2) and Secure Storage Service (S3) are well-known examples of current IaaS offerings.
- PaaS: is an application development and deployment platform delivered as a service to developers over the Web. PaaS facilitates development and deployment of applications without the cost and complexity of buying and managing the underlying infrastructure. PaaS offerings comprise of infrastructure software, and typically include a database, middleware and development tools for delivering Web applications and services from the Internet. The consumer's application, however, usually cannot access the infrastructure underneath the platform.
- SaaS: is an "on-demand" application delivery model over the Internet built upon the underlying IaaS and PaaS stacks. It provides a self-contained operating environment used to deliver the entire user experience including the content, its presentation, the application(s), and management capabilities. The SaaS consumer can only access the exposed functions of the application. A typical example is SalesForce.com that offers CRM applications accessible by subscription over the Web. SaaS provides the most integrated functionality built directly into the offering with no option for consumers' extensibility. It cannot handle application variabilities and does not follow the "true" spirit of the SOA paradigm.

The shortcomings and rigidity of current cloud computing service offerings highlighted above prohibit the development of flexible cloud-enabled SBAs. These limitations can be addressed when combining the SOA principles with cloud services resulting in a mixing and matching of external services with on-premise assets and services. This merger offers unprecedented control in allocating resources dynamically to meet the changing needs of SBAs, which is only effective when the service level objectives at the application level guide the cloud's infrastructure management layer. The combination of an SOA with cloud computing concepts brings together scalability, speed, modularity, reuse and the choice of the most appropriate implementation and deployment infrastructure for end-to-end services. Those basic principles are going to allow us to develop novel approaches that revolutionize the way service development is conducted by giving rise to the notion of smart Internet. Moving successfully into smart services poses challenges and opens up possibilities for pioneering research. Not only does it require novel concepts and techniques that will infuse cloud capabilities into an SOA but also requires that application appropriateness be tested and be optimized against both business and technical characteristics. From a business perspective, the application needs to be evaluated in terms of core functionality, service reuse, QoS, compliance requirements, and Service Level Agreement term stipulations. From a technical perspective, the application needs to be evaluated in terms of usage, performance, latency, service-level needs, execution environment and dependencies. Proper application partitioning and fit leads to better performance, scale, ease of change and efficiency that would not otherwise be possible. Our main contribution in this paper is twofold:

- An extensive state-of-the-art analysis on the supports for SBA development on the cloud, which also

leads to the identification of their shortcomings.
- A high-level description of the requirements of a novel uniform cloud service representation model which aims to realize the aforementioned view of a combination of SOA with cloud computing concepts.

The rest of the paper is organized as follows. Section 2 presents an extensive state-of-the-art evaluation. Then, Section 3 identifies the shortcomings of existing approaches and highlights some research challenges derived from the state-of-the-art evaluation. To target the identified shortcomings and research challenges, Section 4 presents a contemporary approach on supporting the SBA development on the cloud. Finally, Section 5 concludes the paper.

2. STATE-OF-THE-ART AND EVALUATION. An SOA promotes loosely-coupled and interoperable service components that are easily shared within and between enterprises via published and discoverable interfaces [29]. It has been suggested by many experts both in the academia and industry that the SOA paradigm promises many advantages over the other architecture paradigms in terms of reusability, business flexibility and agility, and interoperability. However, current SBA development following the SOA paradigm usually leads to a vendor lock-in approach, where the constituting monolithic service components are predominantly tethered to proprietary platforms and infrastructure of a vendor and thus not customizable, extendable and interoperable, cf. the left side of Fig. 2.1. That is because current SOA developments do not usually put focus on the deployment environment of the constituent service components.

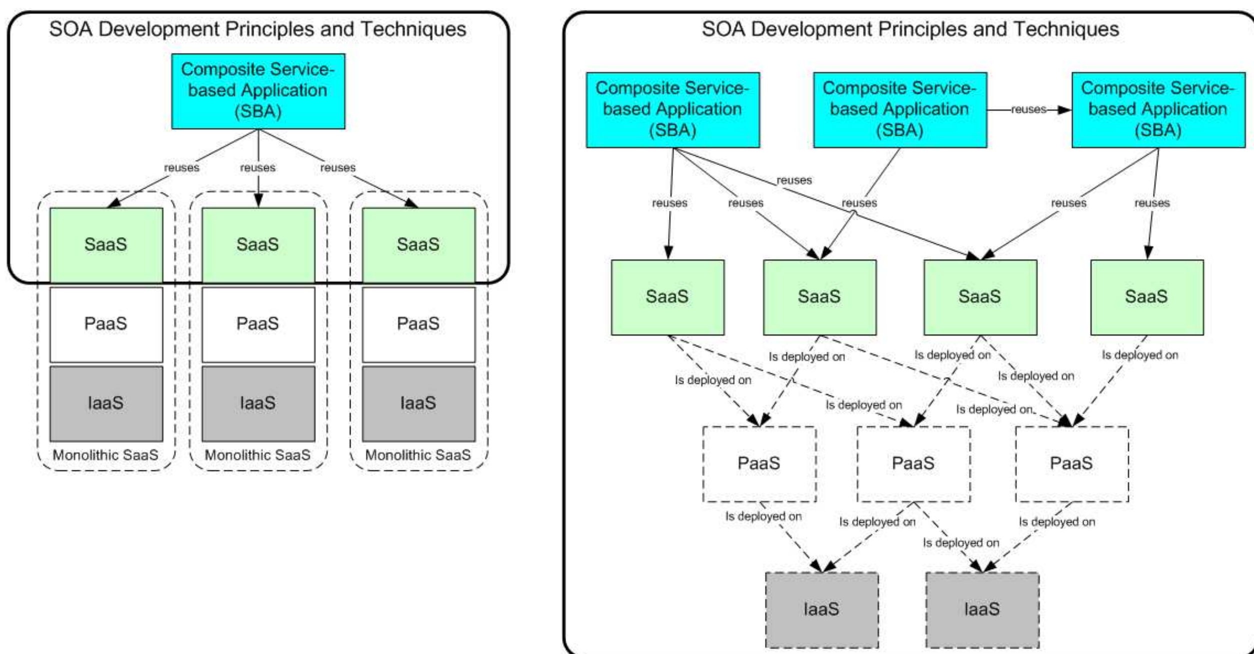


Fig. 2.1: Monolithic SBA Development vs. Cloud-based SBA Development following SOA principles and techniques

This limitation can be addressed by breaking the monolithic service offerings into cloud services (XaaSs) across cloud computing layers, i.e. SaaS, PaaS and IaaS, to enable the platform- and infrastructure-agnostic SBA development on the cloud. Following the SOA principles and techniques, SBA developers can reuse and combine distributed cross-layered XaaS functions. The expression and agreement of non-functional properties between XaaS components on the same layer or across layers must also be available so that cloud-based SBAs that have constraints, such as a maximum amount of time or cost, can be created with a guaranteed QoS[1] that is captured in a Service Level Agreement (SLA). These non-functional properties are referred to as XaaS

[1] http://www.s-cube-network.eu/km/terms/q/quality-of-service-qos

Quality Attributes[2] in this document to capture their wide-ranging nature and to illustrate that they can be used to describe any aspect of how XaaS functions are provided across layers.

As can be seen on the right side of Figure 2.1, an SOA-enabling cloud-based SBA development results in an amalgamation of on-premise and external XaaSs that promotes the reusability and composability of XaaSs across SaaS providers and PaaS/IaaS vendors. Our envisioned SBA development methodology should contain the following three features:

- (F1) Platform- and infrastructure-agnostic SBA development on the cloud, i.e. by using the XaaS offerings from multiple SaaS providers and PaaS/IaaS vendors
- (F2) Following the SOA paradigm to promote the reusability and composability
- (F3) Supporting the SLA specification, negotiation and monitoring regarding the XaaS Quality Attributes across layers.

However, cloud computing is a relatively new research area and only a few existing work supports our envisioned cloud-based SBA development methodology. In this section, we review and evaluate the existing approaches regarding the three desired features. Regarding (F1), Section 2.1 discusses the existing efforts on providing a standardized XaaS representation for supporting the platform- and infrastructure-agnostic, vendor-independent SBA development on the cloud. Section 2.2 targets both (F1) and (F2) from the methodological point of view by presenting the related approaches to develop SBA on the cloud and the existing SOA development methodologies that provide a set of fundamental SOA principles and techniques to be adhered to during the cloud-based SBA development. Lastly regarding (F3), Section 2.3 reviews the existing PaaS/IaaS software and products for platform- and infrastructure-agnostic SBA development on the cloud, putting emphasis on their SLA support.

2.1. XaaS Standardization Support for Cloud-Based SBA Development. While developing SBAs on the cloud, the absence of standardization across cloud vendors, results in unnecessary complexity to obtain interoperability, high switching costs and potential vendor lock-in. The main concerns of cloud-based SBA development are how to deal with the standardization and interoperability between different cloud platforms [33], since cloud computing promises to allow SBA developers to design and develop elastic and inexpensive applications independent of platforms [3]. However, current cloud vendors have different application models, many of which are proprietary, vertically integrated cloud stacks that limit the customizations of the underlying platform and infrastructure resources. There is currently little effort in supporting tools, techniques, procedures or standard data formats or service interfaces that could guarantee data, application and service portability. In [24] the vendor lock-in problem that prevents the interchangeability and interoperability between the SaaSs has been addressed and subsequently a state-of-the-art in both standardization efforts and on-going projects has been presented. Document [34] points out that concerning the vendor lock-in there are still many unsolved compatibility issues beside the API compatibility, such as the data format, billing, metering, error handling, logging, or cloud management and administration. In general, the current situation makes it difficult for SBA developers to migrate their data and service components from one cloud vendor to another or back to an in-house IT environment.

The ability to manipulate, integrate and customize XaaS across different cloud providers for SBA development has been studied in [18] that has IaaS, application and deployment orchestrators but falls short of proposing a solution for the problem at hand.

The Distributed Management Task Force (DMTF) group[3] has published standards such as the Open Virtualization Format (OVF) [12], to provide an open packaging and distribution format for virtual machines, and the Virtualization Management (VMAN) [11] specifications that address the management lifecycle of a virtual environment to help promote interoperable cloud computing service. The OVF is considered nowadays as a standardized means for describing single or multiple virtual machines. Using the OVF supports the specification of either the technical offering of an IaaS provider or the resource requirements of a SaaS or PaaS provider.

Similar to OVF-based approaches, the Solution Deployment Descriptor (SDD) template [26] proposed by OASIS defines an XML schema to describe the characteristics of an installable unit (IU) of software that are relevant for core aspects of its deployment, configuration, and maintenance. The benefits of this work include: the ability to describe software solution packages for both single and multi-platform heterogeneous environments,

[2] `http://www.s-cube-network.eu/km/terms/q/quality-attribute`
[3] `http://dmtf.org/`

the ability to describe software solution packages independent of the software installation technology or supplier, and the ability to provide information necessary to permit full lifecycle maintenance of software solutions.

Document [6] targets the interoperability between the federated clouds by providing a collection of proposals for "Inter-cloud" protocols and formats. However, this work is still in the early stage and targets only the interoperability between the data centers, i.e. only on the infrastructure level. To unlock the vendor lock-in problem concerning the APIs, the Open Grid Forum's Open Cloud Computing Interface (OCCI) working group (www.occi-wg.org) has been developing a uniform API specification for remote management of Cloud Computing infrastructure [27]. This will support the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. The scope of the specification will be all high-level functionality required for the life-cycle management of virtual machines (or workloads) running on virtualization technologies (or containers) supporting service elasticity.

Approaching the cloud-based SBA development from a different perspective, the Model Driven Engineering (MDE) research community has realized the benefit of combining MDE techniques with application development and suggested combining MDE with cloud computing [7]. As the article describes, there is no consensus on the models, languages, model transformations and software processes for the model-driven development of cloud-based applications. Following the MDE vision,the authors in [17] propose a meta-model that allows cloud users to design applications independent of any platform and build inexpensive elastic applications. From their point of view, a SaaS application should avoid the vendor lock-in problem concerning the underlying platforms. This meta-model support the description of the capabilities, technical interfaces, and configuration data for the virtualized infrastructure resources of the cloud application service. Similarly, the work in [8] presents a different customer-centric cloud service model. This model concentrates on aspects such as the customer subscription, capability, billing, etc., yet does not cover other technical aspects of the cloud services including the technical interfaces of the cloud services, the elasticity, the required deployment environment, etc. Other existing models, e.g. in [32], also lack a formal structure and dentitions (reducing their usability and reusability) or are not explicit and assume tacit knowledge.

In practice, an attempt to provide a template-based approach for using cloud services is available from Amazon through their AWS CloudFormation offering [1]. This template provides AWS developers with the ability to specify a collection of AWS cloud resources and the provisioning of these resources in an orderly and predictable fashion. Nevertheless, this template works only for AWS cloud platform and infrastructure resources and thus lacks interoperability.

2.1.1. Summary and Evaluation. In summary (cf. Table 2.1), existing approaches mostly target the infrastructure levels and cover only certain perspective of standardizations for cloud services, e.g. description format, APIs, protocol, definition models, protocols, SLA, etc. The state of the art analysis has shown that there is a lack of a uniform representation for cross-layered XaaSs that unifies all views on an XaaS, e.g. from the customer view on the APIs and SLA, to the developers that are responsible for deploying and maintaining that XaaS through the cloud.

2.2. Methodology Support for Cloud-Based SBA Development. Apart from the standardization supports for the interoperability and portability between cloud vendors, we are also interested in the existing methodologies that provide guidelines for developing composite SBAs on the cloud. As mentioned before, cloud-based SBA development may benefit from existing SOA development methodologies that aim to provide guidelines, models, best practices, standards and reference architectures necessary to construct a well-defined and well-structured SOA. Hence, Section 2.2.1 reviews some well-known SOA development methodologies. Since an SBA can be considered as a composite SaaS application, in Section 2.2.2 existing methodologies for SaaS development on the cloud are discussed, some of which were already developed based on the XaaS description standards mentioned in the previous Section 2.1.

2.2.1. SOA Development Methodologies. In general, an SOA methodology is either an industrial initiative or academic proposal. First, several industrial SOA initiatives have emerged over the past years, resulting in many methodologies, notably the Service-Oriented Modelling and Architecture (SOMA) [4], Service Lifecycle Process [31], and Service-Oriented Modelling Framework (SOMF) [5]. In addition, vendors such as SAP and BEA, and firms providing analysis and design consulting services such as Cap Gemini, and Everware-CBDI proposed their own methodologies. In academia, we have witnessed several activities that concentrate around SOA analysis, design and development. Most prominent approaches are the Service Development Lifecycle

Table 2.1: XaaS Standardization Support - Summary and Evaluation

Approaches	**SOA-based**	**SLA-aware**	**Maturity Level**			
			Academic Proposal	**Standard**	**Vendor proprietary**	**Industrial Adoption**
SaaS Application Meta-Model [17]		X	X			
Customer Centric Model for SaaS Development [8]		X	X			
OVF [12], [16], [9]		X		X	X (Originally by VMware)	X
SDD [26]				X (by OASIS)		
InterCloud [6]			X			
OCCI [27]				X (By the Open Grid Forum)		
Amazon CloudFormation [1]		X (partly)			X (by Amazon)	

(SLDC) methodology [29], SOA Analysis and Design/Decision Modelling (SOAD) [35] and SOA Framework for Service Definition and Realization (SOAF) [14]. It goes far beyond the scope of this article to review all the SOA methodologies in detail here. Hence, we refer to the work in [2], which provides detailed surveys, comparisons and evaluations of a number of existing approaches.

Although cloud-based SBA development may benefit from adopting the SOA principles and techniques by following the SOA development methodologies, none of these SOA methodologies have considered the appealing characteristics of the cloud as a deployment environment for the SBA as a whole, nor for its constituting SaaS components. However, if we look specifically at the development of each individual SaaS component, some of the contemporary approaches have already taken into account the cloud advantages for SaaS developments. As we consider an SBA as a composition of SaaS components following the SOA paradigm, it is worth to understand how these contemporary approaches have recognized the benefit of combining SOA with cloud computing. The next Section 2.2.2 reviews these approaches.

2.2.2. SaaS Development Methodologies. A systematic process for developing high-quality cloud SaaSs has been proposed by La et al. in [20], taking into considerations the key design criteria for SaaSs and the essential commonality/variability analysis to maximize the reusability. Although this approach claims to develop cloud-based SaaSs, it does not discuss about the cloud support for the deployment environment of the SaaSs.

Maximilien et al. introduces a cloud-agnostic middleware in [21] that can sit on top of many PaaS/IaaS offerings and enable a platform-agnostic SaaS development. They provide a meta-model for describing SaaS applications and their needed cloud resources, and APIs and middleware services for the deployment.

The connection between SOA and cloud computing has been established by the Service-Oriented Cloud Computing Architecture (SOCCA) proposed in [33]. Using the SOCCA, developers can build SaaS applications following an integrated SOA framework. Cloud platform and infrastructure resources will be discovered by a Cloud Broker Layer and a Cloud Ontology Mapping Layer for deploying the SaaS components. The multi-tenancy feature of cloud computing is also supported by SOCCA where multiple instances of SaaS applications or components can be provided to multiple tenants. Although the SOCCA is a useful reference architecture for developing cloud-based SaaSs following the SOA paradigm, a lot of supports are lacking here including a concrete definition language for the SaaS applications and components, a mapping approach for finding necessary cloud resources, and the ability to specify and resolve end-to-end constraints of SaaS applications that might affect

the underlying cloud resources.

The Cafe application and component templates in [22] are a relevant approach for cloud-based SaaS development that provides an ad-hoc composition technique for application components and cloud resources following the Service Component Architecture (SCA). However, this approach requires SaaS developers to possess deep technical knowledge of the application architecture and the physical cloud deployment environment to select and compose the right application components and cloud resources. Furthermore, application component and cloud resource discovery in Cafe uses WS-Policy matching from [23] and cannot retrieve components and resources that satisfy end-to-end quality-of-service constraints.

The work in [16] uses the OVF to define a service definition language for deploying complex SaaS applications in federated IaaS clouds. These SaaS applications consist of a collection of virtual machines (VMs) with several configuration parameters (e.g., hostnames, IP addresses and other application specific details) for software components (e.g., web/application servers, database, operating system) running on the VMs. The service definition language enables also the specification of SaaSs' Key Performance Indicators (KPIs) and the elasticity rules that prescribe what to do in case the KPIs do not meet the expected levels. The work in [9] extends this language into a service definition manifest to serve as a contract between a SaaS provider and the infrastructure provider. In this contract, architectural constraints and invariants regarding the infrastructure resource provisioning for an application service are specified and can be used for on-demand cloud infrastructure provisioning at run-time. KPIs monitoring mechanisms are also specified in the contract for ensuring the timely scaling of the provisioned infrastructure resources based on the specified elasticity rules. Nevertheless, the existing work related to the OVF targets the infrastructure level only, i.e., they support the specification of architecture constraints for deploying the applications directly on (federated) data centres but do not cover the holistic picture of a top-down development of SBAs on the cloud that can guide developers in selecting, resolving and composing cross-layered XaaS offerings. Using the service definition manifest to specify the structure of a SaaS application, i.e. the SaaS components and their required Virtual Execution Environments (VEE), the Reservoir architecture [30] can automatically provision the VEE instances that can run simultaneously without conflict on a federated cloud infrastructure of multiple providers. KPI monitoring mechanisms and elasticity rules in the manifest act as a contract that guarantees the required Service Level Agreement (SLA) between the SaaS provider and the Reservoir architecture.

Model-driven approaches are also employed for the purpose of automating the deployment of complex SaaSs on cloud infrastructure. For instance, the authors in [19] propose a virtual appliance model, which treats virtual images as building blocks for IaaS composite solutions. Virtual appliances are composed into virtual solution model and deployment time requirements are then determined in a cloud-independent manner using a parameterized deployment plan. In a similar way, [10] describes a solution-based provisioning mechanism using composite appliances to automate the deployment of complex SaaSs on a cloud infrastructure.

2.2.3. Summary and Evaluation. The state of the art analysis (cf. Table 2.2) has shown that there is a need for a methodology that guides cloud-based SBA developers to make informed decisions for selecting, customizing, and composing cross-layered XaaS offerings from multiple providers. The methodology should obey the SOA principles and techniques that promote the reusability, loose coupling and composability of the underlying XaaSs.

2.3. Cloud-Software & Product Support for the Cloud-Based SaaS Development. This section describes the current state-of-the-art of cloud software and IaaS and PaaS cloud offerings for SaaS development, with emphasis on their SLA specification, negotiation and monitoring supports. The expression and agreement of IaaS and PaaS quality attributes are a necessary prerequisite to allow SaaS applications with a minimum quality of service to be built using IaaS/PaaS components. The quality attributes for a service are gathered together in a contract between the customer and a provider, known as a Service Level Agreement (SLA). The section is split into three parts: first, we describe selected IaaS and PaaS cloud computing software (i.e., frameworks that can be used by cloud service providers to offer an IaaS/PaaS service) and show how most of these do not support the expression of quality attributes and SLAs. The second and third parts describe IaaS and PaaS cloud offerings that support SLAs and what quality attributes are used to guarantee the SLA offered. This section concludes with a summary of the findings. Please note that this section concentrates on some of the main products in this category and their features to illustrate the state-of-the-art and is not intended to be a comprehensive catalogue of all available cloud computing products and platforms.

Table 2.2: Cloud-based SaaS Development Methodology - Summary and Evaluation

Approaches	SOA-based	SLA-aware	Maturity Level			
			Academic Proposal	Standard	Vendor proprietary	Industrial Adoption
SOCCA [33]	X	n/a	X			
Café [22]	X	X	X			
Reservoir [30]		X	X			
Cloud-Agnostic Middleware [21]		X (partly)	X			
Virtual Appliance Model [19]			X		X (by IBM)	
Composite Virtual Appliance [10]			X			

2.3.1. Cloud Software Components & Frameworks. Many software frameworks, tools and components have been developed to support the deployment of clouds by third-party (i.e., cloud service providers), however most of them do not offer support for quality metrics or SLAs as described below. As each framework has different priorities they are difficult to classify, therefore we have used the broad criteria of "SLA-enabled" and "non-SLA enabled" to indicate frameworks supporting the expression, monitoring and/or support for quality attributes and those that require additional components to do so:

- **SLA-enabled Components & Framework**

SLA@SOI[4] is an EC-funded FP7 research program to provide "a business-ready service oriented infrastructure empowering the service economy in a flexible and dependable way". A main requirement of "business-ready" includes "Transparent SLA management Service level agreements (SLAs) for defining the exact conditions under which services are provided/consumed can be transparently managed across the whole business and IT stack". The goal of the architecture is to provide a framework such that integration with other Resource managers will quite trivial to achieve. Therefore, it might be the case that the quality metrics supported will be dependent on the metrics supported by the underlying fabric and associated monitoring system.

RESERVOIR[5], sponsored by the EC in the FP7 program, is a software framework for CTOs and CIOs to build cloud infrastructure. RESERVOIR has the goal to deliver better services for businesses and eGovernment with energy-efficiency and elasticity by increasing or lowering computing based on demand. The project "showcases an innovative open SLA management framework" that treats "SLAs as a specific concern within the overall service lifecycle management at infrastructure level". The RESERVOIR component responsible for SLA definition and management, the Service Manager (SM), has been released by Telefonica as the Claudia platform[6], though at the time of writing, the software is not yet finalized and is offered as v0.1.

Claudia[7] supports the TCloud API Self-Monitoring extensions TCloud Self-Monitoring extensions, a set of entities and operations to perform monitoring actions on cloud resources. The extensions allow the monitoring of: CPU, memory and disk usage, and input and output bandwidth for virtual machines, input and output bandwidth for virtual networks, disk usage for virtual disks and CPU, memory, disk and bandwidth usage for virtual data centers (aggregations of virtual machines).

- **Non-SLA-enabled Components & Frameworks**

[4] http://sla-at-soi.eu/
[5] http://www.reservoir-fp7.eu/
[6] http://claudia.morfeo-project.org/
[7] TCloud API v0.9 http://www.tid.es/files/doc/apis/TCloud_API_Spec_v0.9.pdf

Arjuna's Agility can be used by multiple, federated organizations which have their own services, resources and policies. Initially, these organizations are likely to be semi-independent departments within a single enterprise. Ultimately, Agility^{TM} can provide seamless access to external utility/cloud services and resources[8].

Cloud.com provides an open-source cloud-computing platform for building and managing private and public cloud infrastructure[9].

The **Enomaly Elastic Cloud Computing Platform (ECP)** is software that "is specifically designed to meet the requirements of carriers, xSPs, and hosting providers who want to offer an Infrastructure-on-Demand or IaaS service to their customers"[10].

enStratus^{TM} is a cloud infrastructure management solution for deploying and managing enterprise-class applications in public, private and hybrid clouds[11].

Eucalyptus is an open source cloud-computing framework for academic research that provides computational resources for experiment and research. Eucalyptus is tightly integrated with the Xen Hypervisor [13]. It does not provide "metering with centralized reporting, so it's difficult to fulfill Service Level Agreements [using it]"[12].

Kaavo offers software that can handle the scale and complexity of application lifecycle management in the clouds and manage applications running on physical resources from different IaaS providers[13].

Nimbus is an open source toolkit aimed at the science community that supports the conversion of a cluster into an Infrastructure-as-a-Service (IaaS) cloud[14].

OpenNebula is an open-source cloud computing toolkit for managing heterogeneous distributed data centre infrastructures[15]. It "provides an abstraction layer independent from underlying services for security, virtualization, networking and storage, avoiding vendor lock-in and enabling interoperability"[16].

OpenStack is an IaaS cloud-computing project started by Rackspace Cloud and NASA but now contains 60 other companies including Citrix Systems, Dell, AMD, Intel, Canonical and Cisco[17]. OpenStack Compute is open source software designed to provision and manage large networks of virtual machines, creating a redundant and scalable cloud computing platform[18], whilst OpenStack Object Storage is open source software for creating redundant, scalable object storage using clusters of standardized servers to store petabytes of accessible data[19].

VMWare is a developer of virtualization software used by IaaS providers to host operating system instances[20].

The **Xen Cloud Platform (XCP)** is the fully-open sourced and freely-available follow-on of the XenServer application developed by Citrix who provide software "for powering and managing scalable clouds"[21]. XCP provides "an open source enterprise-ready server virtualization and cloud computing platform, delivering the Xen Hypervisor with support".

2.3.2. IaaS Cloud Offerings with Support for Quality Attributes/SLAs. **Amazon EC2** offers a blanket "*one-size-fits-all*" SLA to its customers. The SLA specifies only that the service should be available 99.95% in a Service Year - 365 days from the date of an SLA claim. Any lower and the customer is eligible for a Service Credit[22]. Guaranteed minimum levels of performance and response times by the platform are not available.

AT&T provides a simple, elastic, secure and cost-effective cloud solution that offers storage and compute services. The compute service has a "service level agreement of 99.9% for availability of the infrastructure"[23]

[8] http://www.arjuna.com
[9] http://www.cloud.com/
[10] http://www.enomaly.com/Elastic-Computin.457.0.html
[11] http://www.enstratus.com/page/1/about-us-overview.jsp
[12] http://cnsa-cloud-project.wikidot.com/eucalyptus\#toc8
[13] http://www.kaavo.com/products-and-services/product
[14] http://www.nimbusproject.org/
[15] http://en.wikipedia.org/wiki/OpenNebula
[16] http://blog.opennebula.org/?p=683
[17] http://en.wikipedia.org/wiki/OpenStack
[18] http://openstack.org/projects/compute/
[19] http://openstack.org/projects/storage/
[20] http://www.vmware.com/solutions/cloud-computing/index.html
[21] http://www.citrix.com/English/ps2/products/product.asp?contentID=1681633&ntref=hp_cat_cloud
[22] http://aws.amazon.com/ec2-sla/
[23] https://portal.synaptic.att.com/caas

, whilst the storage service is “backed by a 99.9% service level agreement for availability of the web services API”[24].

GoGrid provides scalable cloud infrastructure in multiple data centres using dedicated and cloud servers, elastic hardware load balancing and cloud storage[25]. The SLA for these services covers the following elements of the service: server uptime, persistent storage, internal and external network performance, load balancing, cloud storage, server reboot, support response time, domain name services, physical security and 24 x 365 engineering support.

IBM Cloud Infrastructure allows its customers to run operating system instances (currently RedHat Enterprise, Suse Linux Enterprise and Microsoft Windows Server) and together with a set of Operations Support Systems (e.g. monitoring and co-ordination software) and optimization services (e.g. support for high performance computing and special processor architectures)[26]. The SmartCloud Enterprise infrastructure cloud “comes with a 99.5 per cent uptime service level agreement”[27].

Joyent provides a cloud platform for creating and deploying scalable applications[28]. The cloud platform contains Smart Machines (compute instances) and specializations called Smart Appliances (e.g., relational and key-value storage and load-balancing). The service level agreement provided by Joyent states that their goal is “to achieve 100% availability for all customers”[29]. However, the SLA provides several caveats for the provider including the ability to scheduled maintenance, emergency maintenance and upgrades at their discretion.

Netmagic Solutions are a managed hosting business based in India that offers the SimpliCloud IaaS to run operating system instances and provide a storage infrastructure[30]. The SLA provided with the product provides a guarantee of service credits if there is less than “99.99% uptime on customer cloud server instance over a calendar month of usage [where the] server instance availability = 100* (total minutes per month - unscheduled downtime minutes) / total minutes per month”[31].

Online Tech is the supplier of a private cloud hosting service packages[32]. The data centres hosting the IaaS are SAS 70 & SSAE 16 certified environments, which is “validation to our clients and potential clients that our security procedures, change management practices, and the quality of our services is satisfactory”[33]. The managed is advertised with “100% Uptime – High Availability”, but it is not clear if this is part of the SLA which comes with the package[34].

ReliaCloud offers a similar product to Amazon’s EC2 with their Cloud Servers, i.e., customers can launch instances of operating system templates to run their applications and are billed according to the rate for the template and the overall time it was active. The SLA for this offering is “a cloud platform with enterprise-grade reliability and security. All our cloud servers are persistent and highly available [and] if one part of our cloud platform fails, your servers will restart somewhere else in our cloud”[35]. A guarantee on the time between detection of a failure and a service restarting is not given.

Windows Server Hyper-V[36] is a private cloud offering from Microsoft. Unclear if the infrastructure offers a standard (or any) SLA - it is possible the SLAs are negotiated individually as part of the private cloud agreement. Note Microsoft’s Azure PaaS does offer an SLA.

2.3.3. PaaS Cloud Offerings with Support for Quality Attributes/SLAs. There are fewer PaaS cloud services available than IaaS because a PaaS service must provide a flexible yet secure programming/development environment in addition to providing a scalable underlying resource infrastructure.

Commensus provides to its customers a hosting and virtualization platform (C-VIP) in the V-Cloud Enterprise Private Cloud solution. The platform is backed by a “99.999% uptime SLA and consistently achieves

[24] `https://portal.synaptic.att.com/staas`
[25] `http://www.gogrid.com/`
[26] `http://www.ibm.com/cloud-computing/us/en/`
[27] `http://www.theregister.co.uk/2011/04/08/ibm_smartcloud_enterprise/`
[28] `http://www.joyentcloud.com/`
[29] `http://joyent.com/company/policies/cloud-hosting-service-level-agreement`
[30] `http://www.netmagicsolutions.com/cloud-hosting-services/`
[31] `http://www.netmagicsolutions.com/service-level-agreements.html`
[32] `http://www.onlinetech.com/cloud-computing-hosting/private-cloud-hosting-packages`
[33] `http://www.onlinetech.com/secure-hosting/certified-data-centers/sas-70`
[34] `http://www.onlinetech.com/cloud-computing-hosting/managed-cloud-hosting`
[35] `http://www.reliacloud.com/cloudservers/`
[36] `http://www.microsoft.com/en-us/server-cloud/windows-server/server-virtualization.aspx`

100% network uptime"[37].

The **Google App Engine**'s SLA is described as "a draft of proposed SLA terms only. These proposed terms are not applicable to existing Google products and services in any way. Google reserves the right to modify these SLA terms at any time at its discretion" and provides generous exclusions for the provider[38]. However, the agreement does describe SLA metrics for "error rates" (number of Error Requests divided by the total number of user requests to a particular Customer application during a given five minute period), "error requests" (any user request to Customer's application that results in an error, which is caused by the Service, from the application server infrastructure or application datastore infrastructure in response to a valid read or write request) and "Monthly Uptime Percentage" (100% minus the average of the Error Rates from each five minute period in the monthly billing cycle for a particular Customer application).

OrangeScape[39] offers a PaaS that builds on top of Google's App Engine (see above) and provides an environment for building rich applications. Does offer a feature to define application-level SLAs on individual activities (steps in the application workflow accomplished through completing one or more actions), but no SLA for the platform is provided.

Windows Azure has separate SLAs for compute and storage instances, SQL platform and application fabric[40]. The compute SLA offers a monitoring assurance that "99.9% of the time we will detect when a role instance's process is not running and initiate corrective action" and a network availability guarantee "that two or more role instances[41] in different fault and upgrade domains will have external connectivity at least 99.95% of the time". For storage instances, Microsoft "guarantees that at least 99.9% of the time we will successfully process correctly formatted requests that we receive to add, update, read and delete data and that your storage accounts will have connectivity to our Internet gateway". The SQL platform has the guarantee that it will maintain a 'Monthly Availability' of 99.9% during a calendar month. 'Monthly Availability Percentage' for a specific customer database is the ratio of the time the database was available to customer to the total time in a month. Time is measured in 5-minute intervals in a 30-day monthly cycle. Availability is always calculated for a full month. An interval is marked as unavailable if the customer's attempts to connect to a database are rejected by the SQL Azure gateway". The application fabric SLA is the same as that of the compute and storage instances.

WOLF, offered by Wolf Frameworks, is an Online Database Application Platform architected to help the user design, deliver and use Software-as-a-Service (SaaS) database applications[42]. The platform offers "a Service Level Assurance of 99.97% and maintains a strict Business Continuity Service"[43].

2.3.4. Summary and Evaluation. The state of the art in cloud software and product support has shown that several IaaS and PaaS providers support quality metrics and their definition in SLAs. However, in most of these cases, none are negotiable or machine-readable (and mainly concentrate on guaranteeing availability or the uptime of the infrastructure or platform) and there is a lack of support for more advanced quality metrics, for example only Google App Engine includes error rates in the SLA. The survey has also highlighted the lack of support for quality attribute monitoring and management and SLA support in cloud computing components and frameworks (cf. Sect. 2.3.1) that can be used to create new clouds.

3. SHORTCOMINGS OF EXISTING APPROACHES. This section summarizes the shortcomings we identified while evaluating and comparing the different approaches in the state-of-the-art (cf. Table 3.1).

As a summary, most of the research activities contributing to the state-of-the-art described in the previous section concentrate on platform/infrastructure resource provisioning and attempt to combine and optimize interrelated cloud platform (PaaS) and infrastructure (IaaS) resources. However, we observe little research work on the cloud application (SaaS) level that supports the development of SBAs by utilizing distributed SaaS components that are deployed on a federation of elastic and heterogeneous PaaS and IaaS resources. Unfortunately, current approaches for cloud-based SBAs development cannot meet this expectation and usually

[37] `http://www.commensus.com/About-Us/Cloud-Platform`
[38] `http://code.google.com/appengine/sla.html`
[39] `http://www.orangescape.com/`
[40] `http://www.microsoft.com/windowsazure/sla/`
[41] A role instance is a.NET program that works with IIS, for example ASP.NET or WCF (Web services).
[42] `http://www.wolfframeworks.com/`
[43] `http://www.wolfframeworks.com/faq.asp`

Table 3.1: Summary of existing gaps and innovations needed to address them

Shortcoming (S)	Research Challenge (RC)	Relevant State-of-the-art Survey
S-1: Lack of uniform representation of cross-layered XaaSs	**RC-1**: The state of the art analysis has shown that there is a lack of a uniform representation for cross-layered XaaSs that unifies all views on an XaaS, e.g. from the customer view on the APIs and SLA, to the developers that are responsible for deploying and maintaining the XaaS through the cloud.	XaaS Standardization (cf. Sect. 2.1)
S-2: Lack of considering the appealing characteristics of the cloud as a deployment environment for the SaaSs	**RC-2**: Although cloud-based development may benefit from adopting the SOA principles and techniques by following the SOA developments methodologies, none of these methodologies consider the appealing characteristics of the cloud as a deployment environment for the SaaSs.	SBA development methodology (cf. Sect. 2.2)
S-3: Lack of a concrete definition language for SaaS applications	**RC-3**: Although there are some useful reference architectures for developing cloud-based SaaSs following the SOA paradigm, a lot of supports are lacking here including a concrete definition language for the SaaS applications and components, a mapping approach for finding necessary cloud resources, and the ability to specify and resolve end-to-end constraints of SaaS applications that might affect the underlying cloud resources.	XaaS Standardization (cf. Sect. 2.1)
S-4: Lack of controlled support and optimization for end-to-end services	**RC-4**: The cross-organizational nature of service systems and the potential composition of services across organizational boundaries requires that services are appropriately designed and effectively managed end-to-end for operational and performance effectiveness. In service eco-systems end-to-end services should be configured or re-configured according to QoS parameters, service preferences and requirements declared either by software developer or contained in the terms of an agreed upon SLA.	SLA support of Cloud Software and Product (cf. Sect. 2.3)
S-5: Lack of matching service design options with infrastructure	**RC-5**: The volatile requirements of service-based applications place demands that the execution infrastructure be appropriately configured in response to application characteristics, end-to-end QoS requirements, or when further functional optimisation is required. Research is required on virtualization techniques for cross correlating service components at the application-level with the most appropriate platforms and infrastructure.	SBA development methodology (cf. Sect. 2.2) and SLA support of Cloud Software and Product (cf. Sect. 2.3)
S-6: Lack of achieving scalability in service eco-systems	**RC-6**: Smart Internet service eco-systems require a scalable infrastructure that can be scaled up or down based on application demand, levels of QoS and availability of resources, dynamically evolving workloads, while maintaining critical architectural constraints.	SLA support of Cloud Software and Product (cf. Sect. 2.3)
S-7: Lack of a unified service/cloud service engineering methodology	**RC-7**: This item is the common denominator of all previous open research problems. There is a clear necessity for modern service engineering approaches to infuse cloud computing concepts and functionality into service-oriented systems. In this way it is possible to take a unified holistic view of the complete service-system solution lifecycle that causally connects high-level decisions at the application-level down to the level of resource virtualisation and provisioning of physical resources.	SBA development methodology (cf. Sect. 2.2)

leads to a vendor lock-in approach, where the constituting monolithic SaaS components are predominantly tethered to proprietary platforms and infrastructure of a cloud vendor; i.e., existing SaaSs from providers like SalesForce, Google or SaaSDirectory are often not customizable, extendable or interoperable. This approach fails to follow the true spirit of an SOA that promotes the reuse of loosely coupled services, thus makes it difficult for SBA developers to migrate the SaaS components from one cloud vendor to another or back to an in-house IT environment.

Moreover, existing approaches hardly address end-to-end non-functional requirements and are not a closed-feedback loop, thus partitioning service systems that involve many providers thereby increasing mean time to resolution of errors. For these methodologies scalability, optimal use of resources and continuous improvement of services are hardly considered. Further, they do not address the nature of the execution environment that automates the end-to-end services and their subsequent operation. None of the current methodologies considers the appealing characteristics of the cloud as a deployment environment. This is where our vision differs by deriving also the research challenges in the Table 3.1 to address the identified shortcomings. Through the research challenges, the Table 3.1 summarizes the open areas of future research priority with large potential for major breakthroughs.

To give an idea of how to target the research challenges pointed out in the Table 3.1, the next Section 4 presents an ongoing approach for engineering SBAs on the cloud: the blueprinting approach. Following this

approach, SBA developers can easily syndicate cross-layered XaaSs from multiple cloud vendors to address their application needs, whilst still adhering to the fundamental SOA design principles.

4. BLUEPRINTING APPROACH. As we already noted, developers at the SBA level need to couple their applications in whole or part with external SaaS offerings to provide opportunities related to other areas of their clients' business. To allow full automation and optimization of SBA level actions requires a causal connection of application-level process operations to configurable supporting platforms and infrastructure services. This approach promotes autonomous services (at all levels of the cloud stack) that adhere to the same principles of separation of concerns to minimize dependencies. It allows any service at any layer to be appropriately combined with a service at the same level of the cloud stack or swapped in or out without having to stop and modify other components elsewhere. At the same time, it should allow multiple (and possibly composed) resource/infrastructure or implementation options for a given service at the application-level. Such considerations lead to a syndicated multi-channel delivery model, where it is contrasted the monolithic cloud stack solutions that permeate the cloud today.

This section introduces an approach to target the aforementioned challenges in building cloud-based SBAs that have been derived in the previous Section 3. By decomposing the usual monolithic SaaS offerings and proposing the concept of *Blueprint* as an abstract, uniform representation of cross-layered XaaSs, the Blueprinting Approach is a novel powerful solution that allows SBA applications to dynamically run on top of multiple alternative cloud platform and infrastructure virtualization solutions. Figure 4.1 illustrates how cross-layered components of an SBA solution can be abstracted and described in a series of SaaS, PaaS, and IaaS blueprints to provide a fast and simplified method for provisioning them as cloud services and composing them to build an SBA. The Blueprinting approach also seeks to simplify the SBA deployment by hiding away the complexity of managing all configurations of the underlying middleware and integrating with optimal PaaS/IaaS options. It also achieves portability across clouds and cloud providers to leverage the benefits of elasticity and scale.

Blueprinting supports a flexible top-down continuous closed-feedback loop service refinement and improvement approach. Application-level decisions regarding virtualized end-to-end services are correlated with and are used to drive resource provisioning and adjust the workload and traffic to automate the dynamic configuration and deployment of application instances onto available cloud resources. From the architecture point of view, the blueprinting approach promotes the two fundamental characteristics of a SOA: the reuse and composition of independent, loosely-coupled XaaSs. It supports the independent layering of components within a typical cloud stack, i.e. the XaaS building blocks of an SBA are now composable and interchangeable. For example, a developer can choose to compose SaaSs from multiple SaaS providers into a coherent and integrated SBA, which the developer can then synthesize with PaaSs from one or more PaaS providers, and deploy on different alternative IaaS clouds.

The blueprint framework interlaces the following inter-related elements:

- A declarative Blueprint Definition Language (BDL), which provides the necessary abstraction constructs to describe the operational, performance and capacity requirements of cross-layered XaaSs.
- A Blueprint Manipulation Language (BPML), which provides a set of operators for manipulating, comparing and achieving mappings between blueprints that are defined in BDL.
- A Blueprint Constraint Language (BCL), which specifies any explicitly stated rule or regulation that prescribes any aspect of cloud service.
- A simple blueprint query language for querying collections of blueprints.

In the following, we will briefly introduce the most important elements of the blueprint framework.

4.1. Blueprint Definition Language. To understand the use of blueprinting consider an external developer (e.g., a virtual service operator or provider) that provides mash-up solutions bundling together different types of interactive telecommunications services (wireless home broadband, multi-channel video programming services, including video-on-demand) and enterprise-grade backend services (rating and broadband billing for cable TV services or fixed broadband services, billing processes for mobile services, invoicing, accounts receivable, and so on). The mashing up of these services creates an assortment of offerings that improve business efficiency and customer appeal. The turnkey service solutions are implemented in an end-to-end fashion and involve multiple service providers.

Service providers publish their offerings, e.g., video-on-demand or interactive TV/open cable applications, using a source blueprint model in a telecommunications marketplace - an efficient online platform that manages the distribution of services and the management of bids, thereby streamlining the procurement process.

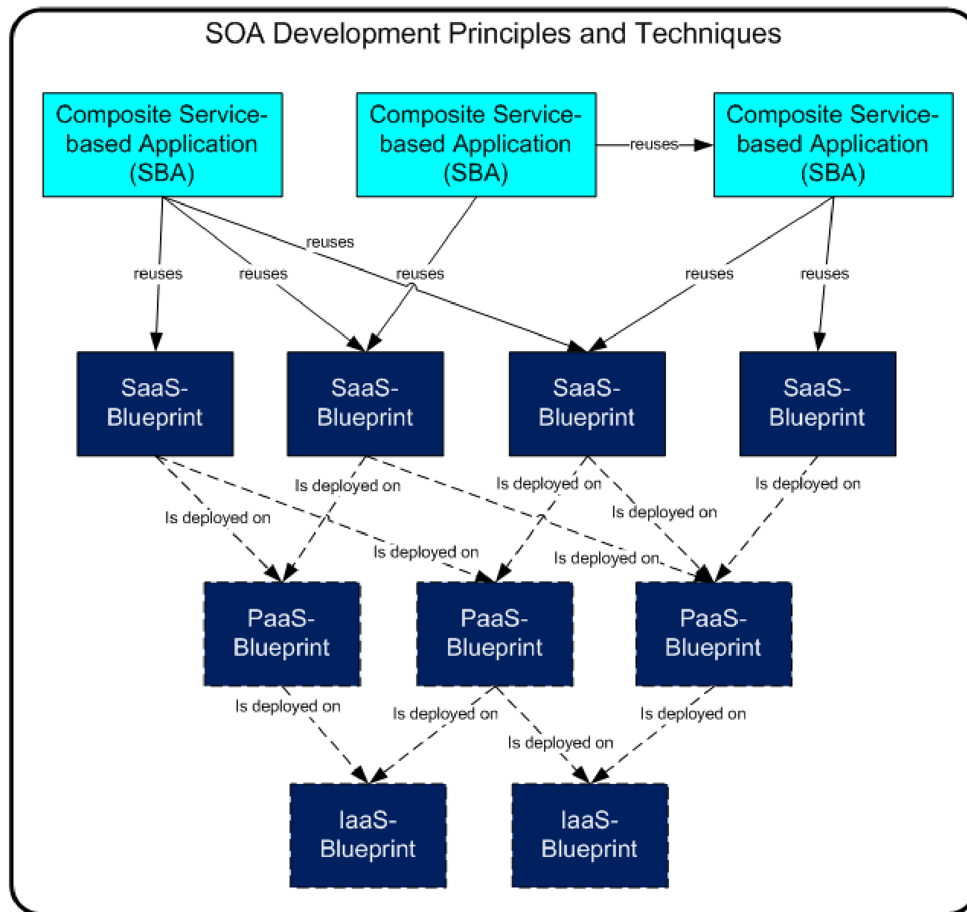


Fig. 4.1: SBA Development using Blueprints

Developers, such as a virtual service provider, can then choose offerings, compose, extend and customize this blueprint model to develop full-featured service-based applications.

A blueprint model is defined in BDL and is based on a clear separation of service processing concerns and is minimally distilled in the following number of inter-related templates:

- Operational service description: This template focuses on the description of functional characteristics of service such as service types, messages, interfaces and operations, namely, the service's signature.
- Performance-oriented service capabilities: This template includes key performance indicators (KPIs) associated with services. Typical examples of quantifiable KPIs are upper and lower performance response time ranges and service availability, throughput, delivery, latency, bandwidth, MTBF (Mean Time Between Failure), MTRS (Mean Time to Restore Service), and so on.
- Resource utilization: This template describes physical infrastructure and resources that are required to run a particular service described in the blueprint model. In general, this template expresses the workload profile including average and peak workload requirements. For instance, a service provider may declare specific technical features that must be taken into account for its service to operate properly, e.g., the server (disk I/O and network) bandwidth required for true on-demand delivery of streaming media, such as video and audio files. This template can express information packaged using the DMFT?s Open Virtualization Format (OVF).
- Policies: This template prescribes, limits, or specifies any aspect of a business agreement that is required by service application developers to use a particular service. It is typically annotated with service level agreements (SLAs) and compliance rules and includes amongst other things security, privacy and

compliance requirements.

In [25], the blueprint XSD template has been released as a first version of the BDL. Within the EC's 4caaSt FP7 project[44] that involves industrial key players in cloud computing such as Telefonica, SAP, Erricson, etc., the blueprint template has been extensively used as a standard, uniform and implementation-agnostic description for XaaS offerings. It has also been validated among the 4caast partners that the blueprint XSD template is capable to capture all the necessary aspects of an industrial cloud service and simple enough to be used by the key industry players in cloud computing.

4.2. Blueprint Constraint Language. The purpose of BCL is to formally express diverse types of policies, such as SLA terms, deployment constraints, data residency constraints, auditability constraints, security constraints, that are related to cloud services and captured by the policy template of the BDL. After a consumer or developer and a provider agree to a set of constraints, these constraints will govern how the services of the provider act.

The BCL is based on a formal foundation to facilitate reasoning and verification by ensuring that services comply with regulations and rules that demarcate their operational behaviour. For this purpose we may use Linear Temporal Logic (LTL) as the formal foundation of BCL and associated compliance patterns. By using automated verification tools that are associated with LTL (e.g. the SPIN model-checker), we can detect compliance violations and provide compliance support for cloud configurations, deployment models and plans that are generated by the blueprint framework prior to execution.

BCL could be used for instance to express that interactive telecommunications services involve high traffic spikes that require automatic provisioning of service instances to accommodate seasonal, e.g., summer period, peaks in demand. The combination of appropriate information from the policy and resource utilization templates will therefore result in claiming resources in advance to accommodate such high traffic spikes.

4.3. Blueprint Manipulation Language. The blueprint model exposes its information in a manner, which facilitates comparison and simple composition of blueprints to express end-to-end offerings from various providers. The BML is based on a set of model-management algebraic operators, such as match, merge, compose, extract, delete and so on. These operators accept source blueprint templates as input and return a new blueprint template as result.

To exemplify the use of BML consider the use of the merging operator on four source blueprints describing high definition IPTV, video on demand, broadband Internet and VoIP services to create an end-to-end triple play service. This operator will yield a target blueprint model that aggregates all four-blueprint models (and their underlying templates) by defining mappings between them. The operator will ascertain that the four blueprint models can be matched to each other and those constraints or capacity requirements are not violated by the target blueprint model.

5. CONCLUSION. This paper provided a survey on existing support for Service-based Application (SBA) development on the cloud. As a summary, the survey has shown that the current cloud solutions are mainly fraught with shortcomings:

- They introduce a monolithic SaaS/PaaS/IaaS stack architecture where a one-size-fits-all mentality prevails. They do not allow SBA developers to mix and match functionalities and services from multiple application, platform and infrastructure providers and configure it dynamically to address their application needs.
- They introduce rigid service orchestration practices tied to a specific resource/infrastructure configuration for the cloud services at the application level.

The above points hamper the (re)-configuration and customization of cloud-based SBAs on demand to reflect evolving inter-organizational collaborations. There is clearly a need to mash up services from a variety of cloud providers to create what has been termed a cloud ecosystem. This type of integration supports the tailoring of SBAs to specific business needs using a mixture of SaaS, PaaS and IaaS.

To deal with the identified shortcomings, we pointed out the need of an abstract and uniform representation for cloud service offerings across cloud computing layers, i.e. SaaS, PaaS, and IaaS. By using this uniform description for cloud service offerings, SBA developers can reuse, customize and combine distributed SaaSs for the SBAs in a seamless manner.

[44]EC's 7th Framework project 4caaSt: http://4caast.morfeo-project.org/

Acknowledgments. The research leading to this result has received funding from the Dutch Jacquard program on Software Engineering Research via contract 638.001.206 SAPIENSA; and the European Union's Seventh Framework Programme FP7/2007-2013 (4CaaSt) under grant agreement n° 258862.

REFERENCES

[1] Amazon Web Services, *Aws cloudformation http://aws.amazon.com/de/cloudformation/*, 2011.
[2] V. Andrikopoulos and et al., *State of the art report on software engineering design knowledge and survey of HCI and contextual knowledge*, Project deliverable PO-JRA-1.1.1, S-Cube Network of Excellence, July 2008.
[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia, *Above the clouds: A berkeley view of cloud computing*, Tech. report, 2009.
[4] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley, *Soma: a method for developing service-oriented solutions*, IBM Syst. J. **47** (2008), no. 3, 377–396.
[5] Michael Bell, *Service-oriented modeling: Service analysis, design, and architecture*, Wiley Publishing, 2008.
[6] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, *Blueprint for the intercloud - protocols and formats for cloud computing interoperability*, Proceedings of the 4th ICIW'09, IEEE, 2009.
[7] H. Brunelière, J. Cabot, and J. Frédéric, *Combining model-driven engineering and cloud computing*, Proceedings of the 4th edition of Modeling, Design, and Analysis for the Service Cloud, June 2010.
[8] H. Cai, K. Zhang, M. Wang, J. Li, L. Sun, and X. Mao, *Customer centric cloud service model and a case study on commerce as a service*, Proceedings of the IEEE CLOUD'09, 2009.
[9] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis, *Software architecture definition for on-demand cloud provisioning*, Proceedings of the 19th ACM HPDC '10 (NY, USA), ACM, 2010, pp. 61–72.
[10] T.C. Chieu, A. Mohindra, A. Karve, and A. Segal, *Solution-based deployment of complex application services on a cloud*, Proceedings of the IEEE SOLI'10, 2010.
[11] DMTF, *Dmtf to develop standards for managing a cloud computing environment, http://www.dmtf.org/standards/cloud.*
[12] *Open Virtualization Format (OVF), http://www.dmtf.org/standards/ovf.*
[13] Patrcia Takako Endo, Glauco Estcio Gonalves, Judith Kelner, and Djamel Sadok, *A survey on open-source cloud computing solutions*, Tech. report, Universidade Federal de Pernambuco, 2010.
[14] Abdelkarim Erradi, Sriram Anand, and Naveen N. Kulkarni, *Soaf: An architectural framework for service definition and realization.*, IEEE SCC'06, 2006, pp. 151–158.
[15] D. A. Fisher, *An emergent perspective on interoperation in systems of systems*, Tech. report, Software Engineering Institute, Carnegie Mellon University, 2006.
[16] Fermín Galán, Americo Sampaio, Luis Rodero-Merino, Irit Loy, Victor Gil, and Luis M. Vaquero, *Service specification in cloud environments based on extensions to open standards*, Proceedings of the 4th COMSWARE '09 (NY, USA), ACM, 2009, pp. 19:1–19:12.
[17] M. Hamdaqa, T. Livogiannis, and L. Tahvildari, *A reference model for developing cloud applications*, In proceedings of CLOSER'11, 2011.
[18] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, *Sky computing*, IEEE Internet Computing **13** (2009), no. 5, 43–51.
[19] Alexander V. Konstantinou, Tamar Eilam, Michael Kalantar, Alexander A. Totok, William Arnold, and Edward Snible, *An architecture for virtual solution composition and deployment in infrastructure clouds*, Proceedings of the 3rd workshop VTDC '09 (NY, USA), ACM, 2009, pp. 9–18.
[20] Hyun Jung La and Soo Dong Kim, *A systematic process for developing high quality saas cloud services*, Proceedings of the 1st CloudCom '09 (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 278–289.
[21] E. Michael Maximilien, Ajith Ranabahu, Roy Engehausen, and Laura C. Anderson, *Toward cloud-agnostic middlewares*, Proceeding of the 24th ACM SIGPLAN OOPSLA '09 (NY, USA), ACM, 2009, pp. 619–626.
[22] Ralph Mietzner, *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*, Dissertation, Universität Stuttgart, Germany, August 2010, p. 369.
[23] Ralph Mietzner, Tammo van Lessen, Alexander Wiese, Matthias Wieland, Dimka Karastoyanova, and Frank Leymann, *Virtualizing services and resources with probus: The ws-policy-aware service and resource bus*, Proceedings of the ICWS'09, 2009, pp. 617–624.
[24] A. Monteiro, J.S. Pinto, C. Teixeira, and T. Batista, *Cloud interchangeability - redefining expectations*, Proceedings of CLOSER'11, 2011.
[25] Dinh Khoa Nguyen, Francesco Lelli, Mike P. Papazoglou, and Willem-Jan van den Heuvel, *Blueprinting approach in support of cloud computing*, Future Internet **4** (2012), no. 1, 322–346.
[26] OASIS, *Solution deployment descriptor specification 1.0, http://docs.oasis-open.org/sdd/v1.0/os/sdd-spec-v1.0-os.pdf*, Tech. report, OASIS, September 2008.
[27] OCCI-Working Group, *Open cloud computing interface - infrastructure*, April 2011.
[28] Michael P. Papazoglou and Willem-Jan van den Heuvel, *Blueprinting the cloud*, IEEE Internet Computing **15** (2011), no. 6, 74–79.
[29] Mike P. Papazoglou and Willem-Jan van den Heuvel, *Service-oriented design and development methodology*, Int. J. Web Eng. Technol. **2** (2006), no. 4, 412–442.
[30] B. Rochwerger and et al., *The reservoir model and architecture for open federated cloud computing*, IBM Journal of Research and Development **53** (2009), no. 4.
[31] SOA Practitioners Guide Part 3, *Introduction to service lifecyle*, 2006.
[32] R.W. Thrash, *Building a cloud computing specification: fundamental engineering for optimizing cloud computing initialtives*,

CSC Whitepaper, August 2010.
[33] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya, *Service-oriented cloud computing architecture*, Proceedings of the 7th ITNG'10, Ieee, 2010, pp. 684–689.
[34] William Vambenepe, *Reality check on cloud portability*, SD Times, June 2009.
[35] O. Zimmermann, *An architectural decision modeling framework for service oriented architecture design*, dissertation.de, 2009.

(All links in the reference and footnotes were last visited on 11/10/2012)

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

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 13, Number 3, pp. 233–249. http://www.scpe.org

ISSN 1895-1767
© 2012 SCPE

APPROACHES TO AGGREGATE PRICE MODELS TO ENABLE COMPOSITE SERVICES ON ELECTRONIC MARKETPLACES *

FREDERIC JUNKER†, JÜRGEN VOGEL‡, AND KATARINA STANOEVSKA§

Abstract. Marketplaces for cloud services, referred to as electronic marketplaces (eMPs), have been introduced for a number of purposes, including comparability and transparency of service offerings. Since the complexity of services and their provisioning is increasing, eMPs need to handle the dynamic and automated creation of composite services, i.e. services which are yielded in part by using third-party services available on the marketplace. To meet this requirement, eMPs need a standardized machine readable description of service offerings. This paper presents an approach to define price models for services traded on eMPs, using a subset of the Unified Service Description Language (USDL). To enable composite service pricing, this paper introduces an approach on the aggregation of several price models into one price model and analyses the time complexity of the price aggregation algorithm presented.

Key words: price model, USDL, composite service, price aggregation

AMS subject classifications. 68-04 68U35

1. Introduction.

An eMP is a place on the Internet where services are offered by *service providers* and consumed by *service consumers*. A number of eMPs with varying functionalities and target audiences exist today, including Microsoft Windows Azure Marketplace, Google Apps Marketplace, AppExchange (salesforce.com), Android Marketplace, SuiteApp.com and Zoho Marketplace [1].

While most applications and services traded on those eMPs are small, monolithic and gadget-style, deploying applications and services on a large scale will lead to an increasing complexity of the underlying software and infrastructure. Accordingly, furthering the pervasion of eMPs requires research on the following topics:

1. Composite services, i.e. services which are provisioned in part by employing third-party services available on the marketplace.
2. Complex price models allowing the service provider of a composite service to cover the cost incurred by using third-party services.
3. Integrated service pricing, i.e. storing technical and pricing aspects together in a machine readable format.

Each of the third-party services on which a composite product relies has its own price model. Still, the composite product shall be analyzed, offered, used and charged like an elementary product from the user's perspective. So far, the structural design and the quantification of price models for services are chosen manually, without support from the marketplace the product is traded on. However, in case of a complex composite product using hundreds of third-party services, this approach does not scale well (1.) due to the large number of combinations of price models to be evaluated, and (2.) due to the tedious integration of price models with heterogeneous structural design.

1.1. Example: Composite Service.

One example of a composite service is a cell phone service, which is visually displayed in cf. Fig. 1.1. When providing the service S, the cell phone service provider has to pay for using $S_1, ..., S_5$, each of which has its own independent price model:

1. Price model of S_1: \$3.000.000 per month for the right to erect cell towers on 3rd party land property.
2. Price model of S_2: \$0.1 per call minute for transmitting calls via the 3rd-party cable network operator's resources (connection-oriented service).
3. Price model of S_3: \$0.05 per text message for transmitting text messages via the 3rd-party cable network operator's resources (connectionless service).
4. Price model of S_4: (i.) \$0.05 per text message for saving the log of text messages in the 3rd-party database. Applies only for the first 5.000.000 text messages per month. (ii.) \$1.000.000 per month for using the database in the first place.

*This paper bases on the paper 'Aggregating Price Models for Composite Services in Cloud Service Marketplaces' presented at the C4BB4C Workshop at the 10th IEEE ISPA, Madrid, July 2012, http://www.dsa-research.org/c4bb4c/

†University of St. Gallen and SAP Research St. Gallen, Blumenbergplatz 9, CH-9000 St. Gallen, Switzerland, frederic.junker@unisg.ch.

‡SAP Research Zürich, Regensdorf/Switzerland, juergen.vogel@sap.com.

§University of St. Gallen, St. Gallen/Switzerland, katarina.stanoevska@unisg.ch.

5. Price model of S_5: \$4 per month staff costs.

Before offering the cell phone service on the marketplace, the provider needs to determine the service's price model. The price model shall be defined such that the cell phone service provider can pay for using the services $S_1, ..., S_5$. Accordingly, the service provider needs to join the price models into one single price model which constitutes the cost incurred by using 3rd-party services[1].

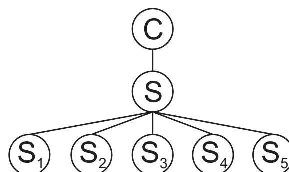


Fig. 1.1: The *product graph* of the example cell phone service.

1.2. Machine Readable Price Models. The Web Services Description Language (WSDL) "only target[s] the description of technical characteristics of services" [2]. In contrast, the Unified Service Description Language (USDL) is an effort to complement WSDL as a "specification language to describe services from a business, operational and technical perspective. USDL plays a major role in the Internet of Services to describe tradable services which are advertised in [eMPs]" [2].

1.3. Research Contribution. This paper presents an approach to define price models using a subset of USDL, including the computation of the *payment*, i.e. the actual amount of money charged to the service consumer, given the service's price model. To support composite service pricing, this paper presents two algorithms for the aggregation of several price models into one price model: aggressive and gentle deinterleaving. We prove both aggregation algorithms have a complexity of $O(n \cdot log(n))$. Aggressive and gentle deinterleaving have individual advantages and shortcomings, and the choice needs to be made based on application requirements.

The remainder of this paper is organized as follows: Sect. 2 presents related work and outlines the gap this paper intends to fill. Sect. 3 defines terms used in this paper as well as the structure of a price model. Sect. 4 describes how to compute the *payment*. Sect. 5 presents and analyses approaches to aggregate the price models of several services.

2. Related Work. Aggregating services is well understood from both technological and business perspectives. [4] analyses the aggregation of web services on a technical level. Research also exists on aggregating Service Level Agreements (SLAs) across several services [5] and aggregating services in business networks [6].

From a business perspective, [7] and [8] examine "determining the optimal composition [of business services] in terms of its short- and long-term profitability" and pricing models for composite mobile services. Beyond, [9] analyses usage metering for e-services, leading to an e-service taxonomy.

Eurich et al [3] research on revenue streams and business model innovation of cloud-based platforms, as well as "the design of the commercialization of PaaS solutions". The revenue streams of several real-world cloud-based platforms are discussed. However, approaches to quantified aggregation of price models and/or revenue streams are not presented. Wang [12] presents an approach towards profit-optimizing selection of third-party services in service value networks. The approach bases on brokers, i.e. instances mediating between customers and providers, and is therefore to be distinguished from our approach. However, the broker has some common features with electronic marketplaces mentioned in this paper, such as charging customers and awarding payments to providers. Another approach towards cost-based selection of third-party services is presented by [16]. An auction system is used to select third-party services to minimize the cost a service provider incurs, while keeping the computational complexity of identifying optimal third-party services low. In contrast, our approach assumes the third-party services have already been selected, and intends to minimize the complexity of the resulting aggregate price model that denotes the cost incurred by using all third-party services. Therefore, [16] and this paper are different in objective, and the two approaches may well be combined to realize efficient automation of service composition, pricing and billing on electronic marketplaces.

[1]Then, the service provider uses marketing approaches to define the price model of service S. This step is covered by existing marketing research.

Kantere et al [13] present an approach to optimally price 'data management services' offered via the cloud. Their approach relies on an economic model that computes the optimal price for cloud cache/data management services based on demand. In contrast, our research work 1) attempts to structure and handle price models for any kind of service offered in the cloud, and 2) deals with service value networks in which composite services rely on third-party services. The problem statement is therefore merely distantly related.

Literature from microeconomics, marketing and pricing, such as Nagle et al [10] and Homburg [11], deal with determining prices and price models under the supply and demand conditions of a free market. Such literature generally aims to determine prices and price models to maximize the profit the provider of a product or service makes. In contrast, we intend to determine a minimum price or price model, which is able to cover the cost a service provider incurs by using third-party services to offer the service.

Approaches from cost-earnings accounting compute indexes and ratios based on costs and earnings realized in the past. In contrast, our approach is based in investment and strategy, i.e. which decision needs to be made to generate certain costs and earnings. SLAs generally determine penalties the provider of a service pays to the customer if a given service level is not met. However, our approach merely determines a price model for a service, i.e. the payment the customer makes to the provider in return for using the service. Service levels and penalties are not considered in this paper, and may be added in future work. Due to the fundamental differences between those fields and our approach, no specific literature from those fields has been included in this overview of related work.

Existing work outlines requirements for the machine readable definition of price models of electronic services and introduces USDL (Unified Service Description Language) [2]. However, the automated aggregation of such machine readable price models has not been examined yet.

3. Price Models. The *price model* defines how much the consumer is charged for consuming a service. The service provider determines the price model before offering the service on the marketplace.

In contrast to the price model, the *payment* is the actual amount of money charged to the service consumer. The *payment* is computed by the payment calculation method described in Sect. 4.

To illustrate the definition of price models presented in this section, we pick the example cell phone service provider who defines the following price model for its service S, as seen in cf. Fig. 1.1: The consumer pays $10 per month as a basic fee, $0.1 per call minute and $0.1 per text message. In case of more than 50 text messages a month, the price drops to $0.05 per text message. Additionally, the actual cost charged to the consumer shall not be higher than $30 each month. In the following sections, we use a subset of USDL (presented in [2]) to transform this textual representation of the price model into a machine readable definition. Also, the price model can be modelled as an ontology.

3.1. Billing Unit. The *billing unit* is the unit per which the consumer is charged ("per what"). For example, the consumer can subscribe to a service and can be charged per the billing units *Day*, *Week*, *Month*, *Quarter* or *Year*.

3.2. Payment Assessment Metric (PAM). A PAM determines the basis by which the consumer gets charged for using a service (i.e. "for what"). Each PAM possesses one or several billing units per which the consumer can be charged. Table 3.1 presents several PAMs and their associated billing units. This list is not exhaustive and can be amended in future work.

3.3. Price Model Component. A price model component (PMC) is a linear function which maps a number of consumed billing units to a *payment*. A PMC specifies a *price* which applies per billing unit, as well as the billing unit itself and a PAM. The term *price* is not to be confused with *price model* and *payment*. Beyond, a PMC applies only during the validity time frame, determined by two dates named t_{VFrom} and t_{VTo}, and within the price fence, which is a range of the number of billing units consumed, represented by the integers u_{Min} and u_{Max}.

Formal definition: A PMC is a 4-tuple *c = (pam, bu, price, condition)*:

- *pam*: A payment assessment metric.
- *bu*: One of the billing units associated with the PAM *pam*.
- *price*: The price charged by the PMC per billing unit.
- *condition* $= (t_{VFrom} \leq t \leq t_{VTo}) \wedge (u_{Min} \leq u \leq u_{Max})$: A logical expression determining whether the PMC is valid for given values of the variables t and u. t denotes the time, u denotes the consumed units, i.e. the number of billing units the consumer has consumed. $t_{VFrom}, t_{VTo} \in \mathbb{N}_0 \cup \{\infty\} =$

PAM	Description	Billing Units
Subscription	The consumer pays for the time frame during which the product can be used. The subscription fee applies independently from the quantitative consumption of the service within the subscribed time frame.	Day Week Month Quarter Year
Pay Per Use Event	The consumer pays for each event of interaction with the service.	Invocation Notification Transaction Session
Pay Per Use Time	The consumer pays for the time of actual interaction with the service.	Millisecond Second Hour Day Week
Pay Per Use Quantity	The consumer pays for the quantity of resources consumed by interacting with the service.	Kilobyte Megabyte Gigabyte
Licence	The consumer makes a one-time payment, which entitles to consumption of the service without limitation in time.	Licence
Admission	A one-time payment the service consumer makes before using the service for the first time. In contrast to the PAM *licence*, the consumer will have to make further payments over time.	Admission

Table 3.1: PAMs and their associated billing units.

$\{0, 1, 2, ..., \infty\}$ specify the PMC's validity time frame. These integers can be interpreted as points of time. $u_{Min}, u_{Max} \in \mathbb{N} \cup \{\infty\} = \{1, 2, ..., \infty\}$ determine the upper and lower bound of the price fence; u_{Max} and t_{VTo} can take the value ∞ to denote the absence of an upper bound.

3.4. Price Model. A price model contains:

- A set of *price model components*. The example price model mentioned above has 4 price model components: One for the $10 monthly basic fee, one for the $0.1 per call minute, one for the $0.1 per text message for the first 50 text messages, and one for the $0.05 per text message if above 50 text messages. The payment generated by the price model equals the sum of the payments generated by each of its PMCs.
- A *payment limit*. If the *payment* is higher than the *payment limit*, only the amount equal to the *payment limit* is charged to the consumer. The semantic function of the payment limit is to allow the service provider to equip the price model with a cost control function. The example price model mentioned above has the payment limit $30, assuming that the payment calculation (see Sect. 4) and billing is run on a monthly basis.

Formal definition: A price model is a couple $P = (C, paymentLimit)$:

- A set of price model components $C = \{c_1, ..., c_n\}$. $C = \emptyset$ means the service is offered for free.
- A payment limit, referred to as *paymentLimit*.

The cell phone price model presented above is translated into the following structured machine-readable price model in USDL format: $P = (\$30, \{A, B, C, D\})$, with the PMCs A, B, C, D presented in cf. Table 3.2.

4. Payment Calculation. This section demonstrates how to compute the *payment*, i.e. the concrete amount of money charged to the consumer of a service, given the service's price model and the consumed units, i.e. the number of billing units consumed by the service consumer. The payment calculation method is required by eMPs, e.g. for billing and price-based product search.

PMC	A	B	C	D
PAM	Subscription	PPUTime	PPUEvent	PPUEvent
billing unit	month	minute	transaction	transaction
price	$10	$0.1	$0.1	$0.05
t_{VFrom}	0	0	0	0
t_{VTo}	∞	∞	∞	∞
u_{Min}	1	1	1	51
u_{Max}	∞	∞	50	∞

Table 3.2: Cell phone price model translated into structured USDL format.

The *payment* generated by the price model is the sum of the payments generated by each of the price model's PMCs. The payment generated by a PMC is the PMC's *price* multiplied by the PMC's *applying units*. Sect. 4.1 demonstrates how to compute the applying units, given the PMC and the number of consumed units. Additionally, as mentioned in the previous section, the price model contains a *payment limit*. Then, the *payment* is computed as follows:

$$payment = min\left(\sum_{i=1}^{n}(c_i.price * a_i), paymentLimit\right)$$

$c_1, ..., c_n$: PMCs of price model
a_i: applying units of PMC c_i
paymentLimit: payment limit of price model.

4.1. Applying Units. The concept of *applying units* allows the service provider to define fine-grained price models, including discounts and time-based changes in pricing. $t_{UFrom}, t_{UTo} \in \mathbb{N}_0$ denote the time frame during which the service was consumed. $v \in \mathbb{N}$ denotes the number of consumed units. The number of applying units a of a PMC is determined by checking the fulfilment of the PMC's condition for all possible value combinations of the variables t and u. The number of applying units equals the number of consumed units multiplied by the ratio between (see cf. Fig. 4.1):

1. the number of value combinations for which the PMC's condition evaluates to *true*, and
2. the total number of value combinations of $t \in D_t = [t_{UFrom}, t_{UTo}]$ and $u \in D_u = [1, v]$.

This definition yields the following formula:

$$a = v * \frac{|(t,u)|}{|D_t \times D_u|} : \quad condition = true$$

where

$$condition = (t_{VFrom} \leq t \leq t_{VTo}) \wedge (u_{Min} \leq u \leq u_{Max}).$$

4.1.1. PAM *Subscription*. In case of the PAM Subscription, the quantitative consumption of the service is specified by the two dates t_{UFrom} and t_{UTo}. The number of consumed units v is the length of the time span $(max(t_{VFrom}, t_{UFrom}), min(t_{VTo}, t_{UTo}))$ measured in billing units. If v is not an integer, it is rounded to the smallest larger integer. Accordingly, the definition presented above decomposes to[2]:

$$a = v \cdot \frac{|[t_{UFrom}, t_{UTo}] \cap [t_{VFrom}, t_{VTo}]|}{|[t_{UFrom}, t_{UTo}]|} \cdot \frac{|[1,v] \cap [u_{Min}, u_{Max}]|}{|[1,v]|}$$

[2] $[x, y]$ denotes the set of numbers between x and y, so $|[x, y]| = y - x + 1$

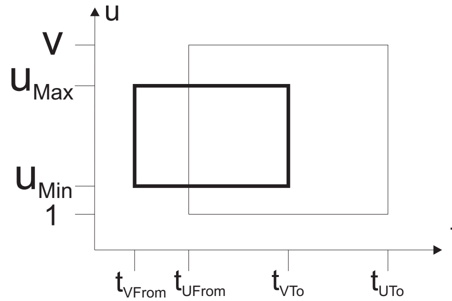


Fig. 4.1: A PMC depicted as a rectangle (with thick lines) in Euclidean space, with the axes being time and quantity of service consumption. The rectangle with thin lines represents the actual consumption by the service consumer. The intersection of the two rectangles equals the set of all combinations for the variables t and u for which the PMC's condition evaluates to *true*.

$$= v \cdot \frac{|[t_{UFrom}, t_{UTo}] \cap [t_{VFrom}, t_{VTo}]|}{(t_{UTo} - t_{UFrom} + 1)} \cdot \frac{|[1, v] \cap [u_{Min}, u_{Max}]|}{v - 1 + 1}$$

$$= \frac{max(0, min(t_{UTo}, t_{VTo}) - max(t_{UFrom}, t_{VFrom}) + 1)}{t_{UTo} - t_{UFrom} + 1} \cdot$$

$$max(0, min(u_{Max} - u_{Min} + 1, v - u_{Min} + 1))$$

4.1.2. PAM other than *Subscription*. In the case of any other PAM than *Subscription*, v is directly specified as a float number which determines the number of billing units consumed within the validity time frame. In this case, $t_{UFrom} = t_{VFrom}$ and $t_{UTo} = t_{VTo}$. Accordingly, the formula for computing a reduces to:

$$a = max(0, min(u_{Max} - u_{Min} + 1, v - u_{Min} + 1))$$

As an example, cf. Fig. 4.2 shows the payment generated by the two PMCs representing the text messages.

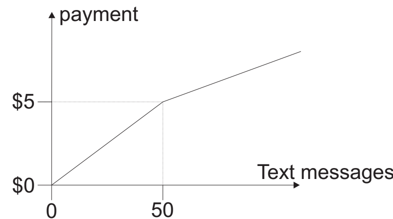


Fig. 4.2: Payment generated by the cell phone price model's PMCs C and D, depending on the number of text messages sent by the service consumer. The quantity discount curve needs to be completed by the customer each time he/she is billed.

4.2. Example. In our cell phone example, the consumer is billed each month. Assume he/she had subscribed to the cell phone service for the entire month 0, and during this time he/she made calls with a total duration of 100 minutes and sent 200 text messages. The *payments* generated by each PMC are presented in cf. Table 4.1[3]. According to the formula presented in Sect. 4, the payment for the entire price model is computed as follows: $payment = min(\$10 + \$10 + \$5 + \$7.5, \$30) = \mathbf{\$30}$.

[3] The numbers v, u_{Min}, u_{Max} and a are measured in billing units. The *price* applies per billing unit (see Sect. 3.3)

PMC	A	B	C	D
PAM	Subscription	PPUTime	PPUEvent	PPUEvent
billing unit	month	minute	transaction	transaction
t_{VFrom}	0	0	0	0
t_{VTo}	∞	∞	∞	∞
t_{UFrom}	0	n.a.	n.a.	n.a.
t_{UTo}	0	n.a.	n.a.	n.a.
v	1	100	200	200
u_{Min}	1	1	1	51
u_{Max}	∞	∞	50	∞
a	1	100	50	150
price	$10	$0.1	$0.1	$0.05
payment	**$10**	**$10**	**$5**	**$7.5**

Table 4.1: Payments generated on behalf of each PMC of the example price model.

5. Price Aggregation. Innovative electronic application marketplaces feature the trading of composite services, which are services yielded in part by using third-party services available on the marketplace. For example, the cell phone service S described in Sect. 1.1 is provided using the 3rd-party services $S_1, ..., S_5$, resulting in the *product graph* shown in cf. Fig. 1.1.

According to the requirements of modern eMPs outlined in Sect. 1, this section presents an approach to aggregate several price models which are defined according to Sect. 3. The straightforward way is to add each PMC of each price model as-is into the resulting aggregate price model. However, a composite product in a real cloud service scenario can consist of hundreds of services, yielding thousands or more PMCs to be aggregated. To accelerate the processes within the marketplace, e.g. cost calculations during product selection, this section presents algorithms minimizing the number of PMCs during price aggregation.

5.1. Terminology and Elementary Aggregation Operations. We refer to the aggregation of a set of PMCs $C_X = \{c_1, ..., c_n\}$ as the computation of a set of PMCs C_Y such that C_X and C_Y are *equivalent*, i.e. C_X and C_Y generate the same *payment*, regardless of the number and temporal occurrence of the consumed units[4]. Aggregation is accomplished by the following procedure:

1. Separate remaining PMCs into groups of same PAM and billing unit.
2. For each group, perform *deinterleaving*, either aggressive or gentle (Sections 5.5 and 5.6).
3. For each group, perform *merging* (Section 5.4).

We refer to *deinterleaving* of a set of PMCs C_X as the process of converting C_X into another set of PMCs C_Y such that C_Y shall contain as few PMCs as possible, and shall contain no overlapping PMCs if possible. We will see that these goals can conflict, and we will present different operations appropriate for different goal priorities.

5.1.1. Merging Horizontally Adjacent PMCs. Two PMCs c_1 and c_2 are *horizontally adjacent* if $c_1.t_{VTo} = c_2.t_{VFrom}$ or vice versa. Merging replaces two horizontally adjacent PMCs by one PMC whose validity time frame is the union of the validity time frames of the two adjacent PMCs (see cf. Fig. 5.1). Horizontally adjacent PMCs can be merged if and only if (1.) they have the same PAM, billing unit and price, and (2.) their price fence has the lower bound zero and no upper bound, i.e. $u_{min} = 0$ and $u_{max} = \infty$.

If the second condition does not hold (i.e. at least one of the PMCs' price fence is not $u_{min} = 0$ and $u_{max} = \infty$), then merging is not possible, even if the first condition holds and all PMCs have the same price fence (see cf. Fig. 5.2). A simple calculation shows that the two sets of PMCs in cf. Fig. 5.2 are not *equivalent*, i.e. they can generate different *payments* for some numbers of consumed units within the time frames (x_1, x_2) and (x_2, x_3).

[4]Thereby, it is ensured the service provider can pay for the 3rd-party services used.

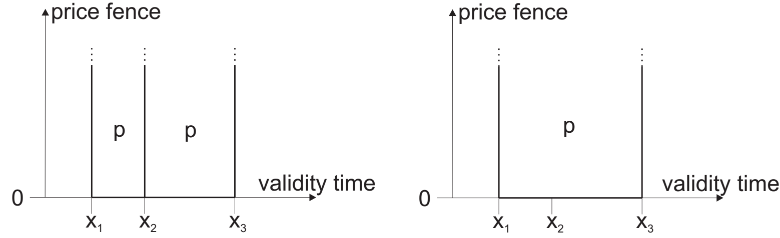


Fig. 5.1: Merging of horizontally adjacent PMCs (before and after).

Fig. 5.2: **Not possible**: Merging of horizontally adjacent PMCs.

5.1.2. Deinterleaving Horizontally Overlapping PMCs. Two PMCs c_1 and c_2 *overlap horizontally* if $c_2.t_{VFrom} < c_1.t_{VTo}$ and $c_2.t_{VTo} > c_1.t_{VFrom}$ or vice versa. Deinterleaving replaces two horizontally overlapping PMCs by three PMCs (see cf. Fig. 5.3). Horizontally overlapping PMCs can be deinterleaved if and only if (1.) they have the same PAM and billing unit, and (2.) their price fence has the lower bound zero and no upper bound, i.e. $u_{min} = 0$ and $u_{max} = \infty$. As in horizontal merging, if the second condition does not hold, then deinterleaving is not possible, even if the first condition holds and all input PMCs have the same price fence.

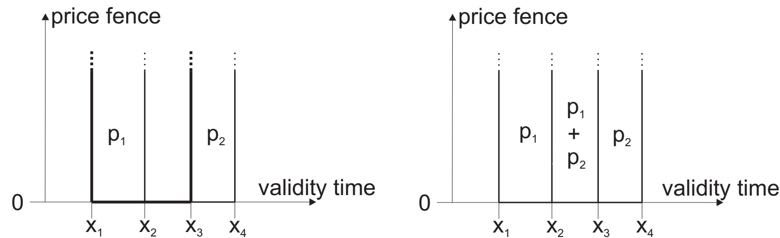


Fig. 5.3: Deinterleaving of horizontally overlapping PMCs (before and after).

5.1.3. Merging Vertically Adjacent PMCs. Two PMCs c_1 and c_2 are *vertically adjacent* if $c_1.u_{min} = c_2.u_{max}$ or vice versa. Vertical merging replaces two vertically adjacent PMCs by one PMC whose price fence is the union of the their price fences (see cf. Fig. 5.4). Adjacent PMCs can be merged if (1.) they have the same PAM, billing unit and price, and (2.) they have the same validity time frame.

5.1.4. Deinterleaving Vertically Adjacent PMCs. Two PMCs c_1 and c_2 *overlap vertically* if $c_2.u_{min} < c_1.u_{max}$ and $c_2.u_{max} > c_1.u_{min}$ or vice versa. Vertically overlapping PMCs can be deinterleaved if and only if (1.) they have the same PAM and billing unit. Due to the analogy to the previous operations, this section does not contain a separate illustrative figure.

5.2. Aggressive Deinterleaving. Aggressive deinterleaving eliminates all overlapping PMCs. The number of output PMCs can be smaller, yet also larger than the number of input PMCs.

5.2.1. Example. To demonstrate the process of aggressive deinterleaving, we take the set of PMCs listed in cf. Table 5.1 as an example. Aggressive deinterleaving transforms the set $C_X = \{A, B, C, D, E, F, G\}$ into the

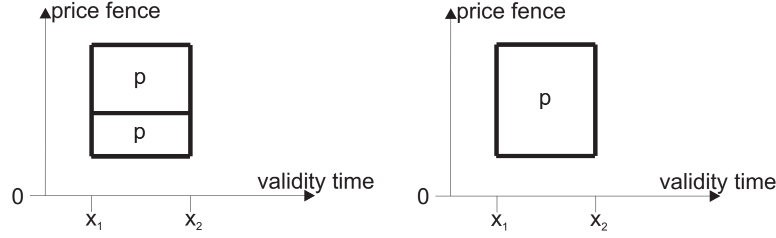


Fig. 5.4: Merging of vertically adjacent PMCs (before and after).

set $C_Y = \{A', B', C', D', E', F', G', H'\}$ are presented in cf. Table 5.2. The PMCs before and after aggressive deinterleaving are visually presented in cf. Fig. 5.5. As outlined in Sect. 5.1, cf. Fig. 5.3, all PMCs in C_X must have (1.) the same PAM and billing unit, and (2.) have the price fence $u_{min} = 0$ and $u_{max} = \infty$.

PMC	t_{VFrom}	t_{VTo}	**price**
A	9	12	1
B	2	9	3
C	7	9	1
D	0	7	1
E	7	11	4
F	6	15	1
G	16	18	2

Table 5.1: Example PMCs before aggressive deinterleaving.

PMC	t_{VFrom}	t_{VTo}	**price**
A'	0	2	1
B'	2	6	4
C'	6	7	5
D'	7	9	9
E'	9	11	6
F'	11	12	2
G'	12	15	1
H'	16	18	2

Table 5.2: Example PMCs after aggressive deinterleaving.

5.2.2. Deinterleaving Algorithm. Algorithm 5.2.1 is the main deinterleaving algorithm. In lines 3 and 4, it sorts the given PMCs by their t_{VFrom} and t_{VTo} dates. The sorted PMCs are stored in two queues, Q_{VFrom} and Q_{VTo}. *lastcut* is initialized with the lowest t_{VFrom} date of any PMC. In the iterations of the **while** loop, algorithm 5.2.1 determines all *cuts* in a sorted order. A PMC c *starts* (respectively *stops*) at a given point $p \in \mathbb{N}$ if $c.t_{VFrom} = p$, respectively $c.t_{VTo} = p$. A *cut* is a point in time where at least one PMC in C_X starts or stops. First, algorithm 5.2.1 computes the price of the next PMC to be added to its output C_Y by (i.) adding the prices of all PMCs in A_{start} to and (ii.) subtracting the prices of all PMCs in A_{end} from the price of the previous PMC added to C_Y. In the first iteration, A_{start} contains all PMCs starting at the smallest t_{VFrom} date of any PMC in C_X and $A_{end} = \emptyset$. In line 17, algorithm 5.2.1 calls algorithm 5.2.2 to determine the next cut, i.e. the smallest t_{VFrom} or t_{VTo} date of all PMCs which have not yet been polled from Q_{VFrom} and Q_{VTo} (see cf. Table 5.5). Algorithm 5.2.2 also stores the PMCs starting and stopping at the next cut in the sets A_{start} and A_{end}. cf. Table 5.3 presents the content of the variables *lastcut*, *cut* and *currentprice* when line 17 of

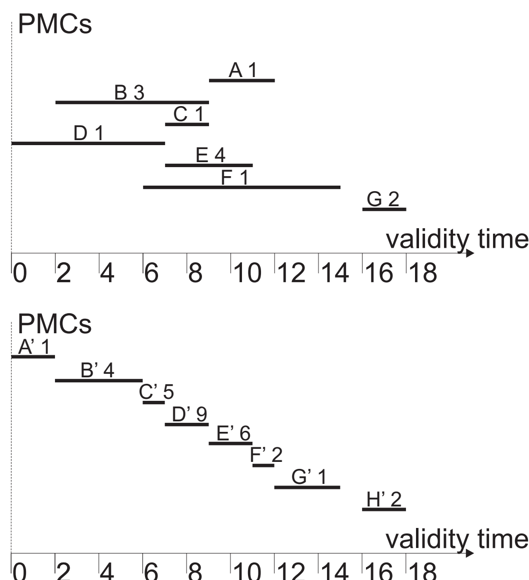


Fig. 5.5: Example PMCs before and after aggressive deinterleaving. The numbers indicate the PMCs' price.

algorithm 5.2.1 has been executed. *Iteration* refers to the iteration of the **while** loop during which the variables take the presented state. In lines 18-21, algorithm 5.2.1 adds a new PMC to C_Y; cf. Table 5.4 shows the PMCs created in each iteration of the *while* loop.

Algorithm 5.2.2 is used by the main deinterleaving algorithm to determine the next *cut*, i.e. the smallest t_{VFrom} or t_{VTo} date where at least one PMC remaining in Q_{VFrom} and Q_{VTo} starts or stops. The queues are sorted, so:

$$cut = min(Q_{VFrom}.peek().t_{VFrom}, Q_{VTo}.peek().t_{VTo})$$

Since several PMCs can start/stop at *cut*, algorithm 5.2.2 polls PMCs from Q_{VFrom} and Q_{VTo} in two *while* loops.

Iteration	1	2	3	4	5	6	7	8	9
lastcut	0	2	6	7	9	11	12	15	16
cut	2	6	7	9	11	12	15	16	18
currentprice	1	4	5	9	6	2	1	0	2

Table 5.3: Example PMCs: Content of variables *lastcut*, *cut* and *currentprice* after execution of line 17 of algorithm 5.2.1.

Iteration	1	2	3	4	5	6	7	8	9
t_{VFrom}	0	2	6	7	9	11	12	n.a.	16
t_{VTo}	2	6	7	9	11	12	15	n.a.	18
price	1	4	5	9	6	2	1	n.a.	2

Table 5.4: Example PMCs: New PMCs created in lines 18-21 of algorithm 5.2.1.

Q_{VFrom}	D	B	F	C	E	A	G
Iteration	0	1	2	3	3	4	8
Q_{VTo}	D	B	C	E	A	F	G
Iteration	3	4	4	5	6	7	9

Table 5.5: Example PMCs: Iterations of *while* loop in algorithm 5.2.1 during which PMCs are polled from the two queues.

Algorithm 5.2.1 Deinterleave

Require: $C_X = \{c_1, ..., c_n\}$: PMCs to be deinterleaved
Require: $C_Y = \emptyset$: holder for algorithm output

```
1:  A_start ← ∅
2:  A_end ← ∅
3:  Q_VFrom ← sortByValidFrom(C_X)
4:  Q_VTo ← sortByValidTo(C_X)
5:  cut ← 0
6:  lastcut ← GetNext(Q_VFrom, Q_VTo, A_start, A_end)
7:  currentprice ← 0
8:  while |Q_VFrom| + |Q_VTo| > 0 do
9:    for all c ∈ A_start do
10:     currentprice ← currentprice + c.price
11:   end for
12:   for all c ∈ A_end do
13:     currentprice ← currentprice − c.price
14:   end for
15:   A_start ← ∅
16:   A_end ← ∅
17:   cut ← GetNext(Q_VFrom, Q_VTo, A_start, A_end)
18:   if currentprice > 0 then
19:     c := new PMC
20:     c.price = currentprice
21:     c.t_VFrom = lastcut
22:     c.t_VTo = cut
23:     C_Y = C_Y ⊕ c
24:   end if
25:   lastcut ← cut
26: end while
```

5.2.3. Upper Bound of Number of PMCs After Deinterleaving. As seen in cf. Fig. 5.5, deinterleaving can increase the number of PMCs. This section proves that the number of PMCs will, however, always be less than doubled: $|C_Y| \leq 2|C_X| - 1$.

Proof.

Induction basis: $|C_X| = 1$. Then: $|C_Y| = 1 \leq 2 * 1 - 1$

Induction hypothesis: $\forall |C_X| < n : \quad |C_Y| \leq 2|C_X| - 1$

Induction step: Prove that $|C_X| = n : \quad |C_Y| \leq 2|C_X| - 1$:

Let $C_{Y,n-1}$ be the resulting set of PMCs of deinterleaving $C_X \setminus \{c\}$ for some $c \in C_X$.

$\Rightarrow |C_{Y,n-1}| \leq 2|C_X \setminus \{c\}| - 1$ according to induction hypothesis.

Prove that $|C_Y| \leq |C_{Y,n-1}| + 2$, i.e. adding c to $C_X \setminus \{c\}$ will increase the number of PMCs after deinterleaving by at most 2.

1. If and only if $\forall c_o \in C_X \setminus \{c\} : \quad c.t_{VFrom} \neq c_o.t_{VFrom} \wedge c.t_{VFrom} \neq c_o.t_{VTo}$, then $|C_Y| \geq |C_{Y,n-1}| + 1$. Illustration: If and only if c starts where no other PMC in C_X starts or stops, then one PMC in $C_{Y,n-1}$ is split in two by adding c to $C_X \setminus \{c\}$, so the number of PMCs after deinterleaving is increased by 1.

Algorithm 5.2.2 GetNext

Require: Q_{VFrom}: Queue containing PMCs sorted by t_{VFrom}
Require: Q_{VTo}: Queue containing PMCs sorted by t_{VTo}
Require: $A_{start} = \emptyset, A_{end} = \emptyset$

1: $min \leftarrow \infty$
2: **if** $|Q_{VFrom}| > 0$ **then**
3: $\quad min \leftarrow Q_{VFrom}.poll().t_{VFrom}$
4: **end if**
5: **if** $|Q_{VTo}| > 0 \wedge min > Q_{VTo}.peek().t_{VTo}$ **then**
6: $\quad min \leftarrow Q_{VTo}.poll().t_{VFrom}$
7: **end if**
8: **while** $|Q_{VFrom}| > 0 \wedge Q_{VFrom}.peek().t_{VFrom} = min$ **do**
9: $\quad A_{start} \leftarrow A_{start} \oplus Q_{VFrom}.poll()$
10: **end while**
11: **while** $|Q_{VTo}| > 0 \wedge Q_{VTo}.peek().t_{VTo} = min$ **do**
12: $\quad A_{end} \leftarrow A_{end} \oplus Q_{VTo}.poll()$
13: **end while**
14: **return** min

Otherwise, the number of PMCs after deinterleaving remains unchanged.

2. Analogously: If and only if $\forall c_o \in C_X \setminus \{c\}: \quad c.t_{VTo} \neq c_o.t_{VTo} \wedge c.t_{VFrom} \neq c_o.t_{VTo}$, then $|C_Y| \geq |C_{Y,n-1}| + 1$.
3. In total, $|C_Y| \leq |C_{Y,n-1}| + 2$.

$\Rightarrow |C_Y| \leq |C_{Y,n-1}| + 2 \leq 2|C_X \setminus c| - 1 + 2 \leq 2|C_X| - 1$. □

5.2.4. Time Complexity of Deinterleaving Algorithm. The deinterleaving algorithm has a time complexity of $O(n \cdot log(n))$, where $n = |C_X|$.

Proof. Lines 1-2, 5 and 7 of Algorithm 5.2.1 take $O(1)$. Lines 3-4 take $O(n \cdot log(n))$ due to sorting. Line 6 of Algorithm 5.2.1 calls Algorithm 5.2.2, one call of which has complexity $O(n)$: Lines 1-7 of Algorithm 5.2.2 have complexity $O(1)$. Lines 8-13 take $O(n)$, since initially the queues Q_{VFrom} and Q_{VTo} are both filled with n PMCs, and at most all of these n PMCs can be polled by lines 9 and 12. Lines 15-16 and 18-25 of Algorithm 5.2.1 take $O(1)$ over one iteration of the **while** loop, and therefore take $O(n)$ over all $O(n)$ iterations of the **while** loop.

We use aggregate analysis to show that line 17 of Algorithm 5.2.1 takes $O(n)$ over all iterations of the **while** loop, despite the call of Algorithm 5.2.2, which can take $\omega(1)$. As mentioned before, lines 1-7 of Algorithm 5.2.2 take O(1). Initially, the queues Q_{VFrom} and Q_{VTo} are both filled with n PMCs, so $|Q_{VFrom}| = |Q_{VFrom}| = n$. Since both algorithms never push any PMCs onto the queues, lines 8-13 Algorithm 5.2.2 can pop at most those n PMCs from the queues over all iterations of the **while** loop in Algorithm 5.2.1. In fact, since line 6 of Algorithm 5.2.1 has already popped one or more PMCs from each queue, the **while** loop actually causes less than n pop operations.

Analogously, we use aggregate analysis to show that lines 9-14 of Algorithm 5.2.1 take $O(n)$. Since A_{start} and A_{end} are filled with the elements popped from the queues by Algorithm 5.2.2 and cleared by every iteration of the **while** loop of Algorithm 5.2.1, each of the $O(n)$ PMCs initially in the queues is exactly once an element of A_{start} and A_{end}. Therefore, both **for** loops in lines 9-14 execute $O(n)$ times over all iterations of the **while** loop. Each execution of any of the **for** loops takes $O(1)$.

In total, the **while** loop of Algorithm 5.2.1 takes $O(n)$, the sorting operations of lines 3-4 take $O(n \cdot log(n))$ and all other operations of Algorithm 5.2.1 take $O(1)$. Therefore, Algorithm 5.2.1 has a complexity of $O(n \cdot log(n))$. □

5.3. Gentle Deinterleaving. Gentle deinterleaving reduces the number of PMCs as far as actually possible, and never increases the number of PMCs. This feature is achieved at the expense that some overlapping PMCs may remain in the deinterleaving output. An example for gentle deinterleaving can be seen in cf. Fig. 5.6. In contrast, aggressive deinterleaving eliminates all overlapping PMCs, at the expense of possibly increasing the number of output PMCs. The advantage of having no overlapping PMCs is that for marketplace operations

such as payment calculation, PMCs are more easily organizable, e.g. in tree structures. The choice between aggressive and gentle deinterleaving needs to be made based on the requirements of the marketplace application.

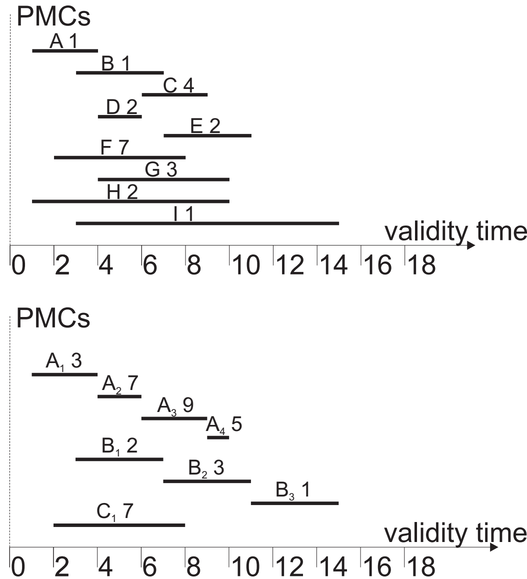


Fig. 5.6: Example PMCs before and after gentle deinterleaving. The numbers indicate the PMCs' price.

5.3.1. Gentle Deinterleaving Algorithm. A PMC is *lonely* if both its t_{VFrom} and t_{VTo} dates are not identical with the t_{VFrom} or t_{VTo} date of any other PMC. The gentle deinterleaving algorithm follows this procedure:

1. Generate PMC graph from the input set of PMCs, each of whose nodes represents one PMC. Two nodes are connected if and only if the represented PMCs start or stop at the same point, or both. This procedure can be performed e.g. by hashing the validFrom/validTo dates of the input PMCs or other duplicate identification approaches. Figure 5.7 shows the PMC graph generated according to the example PMCs shown in cf. Fig 5.6.
2. Identify sets of connected PMCs using standard algorithms for detecting strongly connected components. Our example yields three groups: $G_1 = \{A, D, C, G, H\}, G_2 = \{B, E, I\}, G_3 = \{F\}$.
3. For each set of connected PMCs, perform aggressive deinterleaving (Algorithm 5.2.1), and add the resulting PMCs to output.

The resulting set of PMCs is minimal, i.e. the number of PMCs cannot be reduced any further (except by merging, see Sect. 5.4). Deinterleaving any two PMCs of the resulting set will either leave the number of PMCs unchanged, or add a PMC. The time complexity is $O(n \cdot log(n))$, identically to aggressive deinterleaving. The optimality question is (of course) different, and is left to future work at this point.

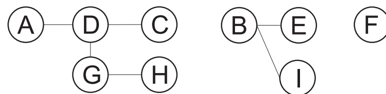


Fig. 5.7: Example PMC graph. PMC F forms a set of its own.

5.4. Merging. Merging is the second part of the aggregation procedure, and is intended to further reduce the number of PMCs based on the operation depicted in cf. Fig. 5.1. The deinterleaving procedure may output adjacent PMCs with the same PAM, billing unit and price. We refer to *merging* as the replacement of adjacent PMCs with the same PAM, billing unit and price by one single PMC. To demonstrate the process of merging,

we take the PMCs listed in cf. Table 5.6 as an example. Merging then transforms these PMCs into the PMCs listed in cf. Table 5.7.

PMC	t_{VFrom}	t_{VTo}	**price**
A	2	5	1
B	5	7	3
C	7	9	3
D	9	11	3
E	12	14	2
F	14	15	2

Table 5.6: Example PMCs before merging.

PMC	t_{VFrom}	t_{VTo}	**price**
A'	2	5	1
B'	5	11	3
E'	12	15	2

Table 5.7: Example PMCs after merging.

Assuming the deinterleaving algorithm has been performed before, the PMCs are sorted by t_{VFrom} (and t_{VTo}). Algorithm 5.4.1 demonstrates merging in time complexity $O(n)$.

Algorithm 5.4.1 Merge

Require: $C_Y = \{c_1, ..., c_n\}$: queue containing PMCs sorted by t_{VFrom} property by Algorithm 5.2.1
Require: $C_Z = \emptyset$: holder for algorithm output
1: $lastPMC = nil$
2: **for all** $c \in C_Y$ **do**
3:   **if** $lastPMC \neq nil \wedge lastPMC.t_{VTo} == c.t_{VFrom}$ **then**
4:     $lastPMC.t_{VTo} = c.t_{VTo}$
5:   **else**
6:     $C_Z = C_Z \oplus c$
7:     $lastPMC = c$
8:   **end if**
9: **end for**

5.5. Deinterleaving and merging based on price fence. The three previous sections demonstrated deinterleaving and merging based on validity time frame. However, as depicted in cf. Fig. 5.4, PMCs can also be deinterleaved and merged based on their price fence, given the conditions mentioned in Sect. 5.1. The procedure is identical, yet the properties t_{UFrom} and t_{UTo} are replaced with u_{min} and u_{max} in the algorithms 5.2.1 and 5.4.1. The operation of adding a PMC to the output set of PMCs in algorithm 5.2.1 can be enhanced with tree structures that produce a set of linked lists of PMCs, where each linked list contains PMCs with identical t_{UFrom} and t_{UTo} dates, which are sorted by price fence. This will increase the time complexity of the aggregation procedure (though not the asymptotic complexity of $O(n \cdot log(n))$), however decreases the effort for payment calculation. This approach is appropriate since aggregation of price models is typically performed only once for one composite product, yet the payment calculation for the resulting aggregate price model is performed multiple times.

5.6. Aggregation Bases. The aggregation procedures presented in the previous sections aggregate PMCs based on their validity time frame, i.e. the part-condition $t_{VFrom} \leq t \leq t_{VTo}$ of a PMC's *condition*. Since the subset of USDL used to define price models will be extended in the future, the *condition* of a PMC may be amended by further part-conditions of the form $a_{low} \leq a \leq a_{high}$. Such part-conditions exhibit the same

properties as the validity time frame: Both have a lower and an upper value referred to as *c.lower*(A) and *c.upper*(A). The applying units are limited to the units consumed between the lower and the upper values. As a result, the aggregation procedures can be adapted, so they can also be applied to these part-conditions.

To facilitate such an adaptation, the this section presents the formal concept of the *aggregation basis*, which generalizes the aggregation procedures presented in previous sections from the validity time frame $t_{VFrom} \leq t \leq t_{VTo}$ to any part-condition of the form $a_{low} \leq a \leq a_{high}$. The *aggregation basis* determines by which of their properties PMCs are aggregated. For each aggregation basis A, the PMC possesses a lower and an upper bound, referred to as *c.lower*(A) and *c.upper*(A). The lower and upper bounds take values from a totally ordered set, referred to as the *aggregation set* S_A. The condition *c.lower*(A) < *c.upper*(A) must hold for the PMC to be properly defined. In this paper, we presented the aggregation basis *validity time frame* A_{VT}. Let $\mathcal{A}$ be the set of all aggregation bases. Accordingly, $\mathcal{A} = \{A_{VT}\}$. $\mathcal{A}$ will be amended once further aggregation bases are added to the structural definition of a PMC. Table 5.8 shows the lower bound, upper bound and aggregation set for the aggregation basis. The price fence is currently not considered an aggregation basis, since cf. Fig. 5.2 shows that aggregation procedures are not necessarily applicable in all cases of price fences.

A	$c.lower(A)$	$c.upper(A)$	S_A
A_{VT}	$c.t_{VFrom}$	$c.t_{VTo}$	$\mathbb{N}_0 \cup \{\infty\}$

Table 5.8: Lower bound, upper bound and aggregation set for a PMC c.

5.7. Multi-Dimensional Deinterleaving and Merging Procedures. Aggregating PMCs by several aggregation variables is significantly more complex than by one single aggregation variable, since PMCs may have different lower/upper values for several aggregation variables. Due to the high complexity of the facts and circumstances, this section merely gives a short sketch of multi-dimensional aggregation of PMCs. Future work shall investigate more deeply on the issue.

5.7.1. Approaches. The trivial way to aggregate PMCs by several aggregation bases is to iterate over all aggregation bases and apply the one-dimensional aggregation presented in previous sections each time. The limitation of this approach is that the number of PMCs is not reduced as far as possible, as one can exemplary see in cf. Fig. 5.8. Aggregation by aggregation basis 1 only merges the PMCs in the upper right corner, and aggregation by aggregation basis 2 introduces no changes at all. However, it is possible to aggregate all PMCs into one 'big' PMC by alternatingly aggregating by each aggregation basis. Multi-dimensional aggregation algorithms can therefore trace which PMCs have been deinterleaved and merged by aggregating by one given aggregation basis, resulting in one or more new PMCs with new lower/upper values to be considered when aggregating over another aggregation basis. As a result, aggregation over one aggregation basis may have to be performed multiple times to minimize the number of PMCs, as seen in cf. Fig. 5.8.

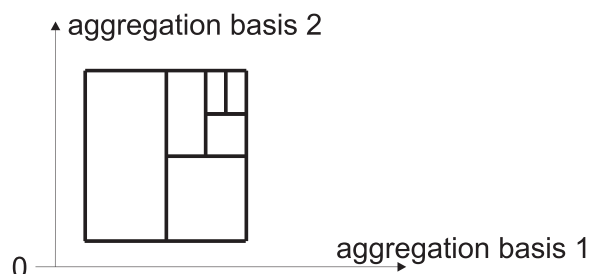


Fig. 5.8: Multi-Dimensional Merging Example: The components need to be merged alternatingly by the two aggregation bases to reduce the number of PMCs as far as possible.

5.7.2. Number of Output PMCs. The problem of multi-dimensional aggregation is that PMCs may have different lower and upper boundaries for several aggregation bases. In the one-dimensional case, deinterleaving two PMCs to eliminate overlapping introduces at most one additional PMC (see Sect. 5.2.3). In the

multi-dimensional case, the number of PMCs may increase much more. One simple solution is to aggregate PMCs only if their lower and upper values are identical in all but at most one aggregation bases. Similarly to the gentle deinterleaving algorithm, this may leave multidimensionally overlapping PMCs in return for keeping the number of PMCs low. Future work shall include detailed analyses on the upper boundary of the number of output PMCs in the multi-dimensional case, analogously to the proof in Sect. 5.2.3.

6. Conclusions and Outlook. This paper presented an approach to define price models using a subset of USDL, as well as an approach to aggregate several price models in order to support composite services in a cloud computing application environment. From a business perspective, future work should investigate how our approach fits into existing approaches towards pricing strategy in electronic services, such as [14] and [15]. Technically, the theory behind aggregating price models yields interesting algorithmic and mathematical questions. While the price model structure presented in this paper is still comparably simple, introducing extensions of the subset of USDL for more fine-grained price models will result in an array of new problems to be solved. These extensions include:

- Different prices for different consumer groups.
- SLAs specifying prices depending on service level chosen by consumer, as well as penalties paid by provider to consumer if service level is not met.

These changes will require the payment calculation and aggregation approaches to be adapted, yet the concept behind the algorithms presented in this paper remains valid.

Future work shall include multi-dimensional aggregation algorithms to handle several aggregation bases. The optimality of the gentle deinterleaving algorithm still requires investigation. Also, we shall present investigations on the effects of adding more aggregation bases on the algorithmic realization of price aggregation. Furthermore, we strive to define PMCs such that price fences can be considered an aggregation variable, avoiding the problem depicted in cf. Fig. 5.2. Finally, the current approach towards price aggregation is that the aggregate price model is simplified as much as possible while keeping the generated payments exactly the same for any number and temporal occurrence of the consumed units. We shall investigate on approaches towards price aggregation that give up this goal to an acceptable extent, in return for being able to offer a simple price model to the service consumer even when the service relies on third-party services with highly heterogeneous price models.

Acknowledgement. The research leading to these results has partially received funding from the 4CaaSt project (http://www.4caast.eu/) from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 258862. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

REFERENCES

[1] F. Noyer, *Characteristics of Electronic Marketplaces for Cloud Computing Services*, Master Thesis, University of Fribourg, Switzerland

[2] J. Cardoso, A. Barros, N. May, U. Kylau, *Towards a Unified Service Description Language for the Internet of Services: Requirements and First Developments*, IEEE International Conference on Services Computing, IEEE Computer Society Press, 2010.

[3] M. Eurich, A. Giessmann, T. Mettler, and K. Stanoevska-Slabeva, *Revenue Streams of Cloud-based Platforms: Current State and Future Directions*, in Proceedings of the Seventeenth Americas Conference on Information Systems (AMCIS), 2011, Paper 302.

[4] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Sheng, *Quality Driven Web Services Composition*, Proceeding WWW '03 Proceedings of the 12th International Conference on World Wide Web.

[5] T. Unger, F. Leymann, S. Mauchart, T. Scheibler, *Aggregation of Service Level Agreements in the Context of Business Processes*, 12th International IEEE Enterprise Distributed Object Computing Conference, 2008. EDOC '08.

[6] T. Kohlborn, A. Korthaus, C. Riedl, H. Krcmar, *Service aggregators in business networks*, 13th Enterprise Distributed Object Computing Conference Workshops, 2009. EDOCW 2009.

[7] H. Elshaafi, J. McGibney, D. BotvichODO, *Business Driven Optimisation of Service Compositions*, 7th International Conference on Next Generation Web Services Practices (NWeSP), 2011

[8] F. Faria, J. M. Nogueira, *Context-Based Application-Aware Pricing for Composite Mobile Services in Wireless Networks*, Wireless Days (WD), 2010 IFIP

[9] V. Agarwal, N. Karnik, A. Kumar, *Metering and accounting for composite e-services*, IEEE International Conference on E-Commerce, 2003. CEC 2003.

[10] T. T. Nagle, R. K. Holden, G. M. Larsen, *Pricing, Praxis der optimalen Preisfindung*, Springer-Verlag, Berlin/Heidelberg, 1998.

[11] C. Homburg, *Marketingmanagement, 4. Auflage*, Gabler-Verlag, Springer Fachmedien, Wiesbaden, 2011.

[12] X. Wang, *Price Heuristics for Highly Efficient Profit Optimization of Service Composition*, 2011 IEEE International Conference on Services Computing (SCC).
[13] V. Kantere, D. Dash, G. Francois, S. Kyriakopoulou, A. Ailamaki, *Optimal service pricing for a cloud cache*, IEEE Transactions on Knowledge and Data Engineering, 2011
[14] A. Dixit, T. W. Whipple, G. M. Zinkhan, E. Gailey, *A taxonomy of information technology-enhanced pricing strategies*, Journal of Business Research, Volume 61, Issue 4, 2008, pp. 275-283
[15] E. Iveroth, A. Westelius, C.-J. Petri, N.-G. Olve, M. Cöster, F. Nilsson, *How to differentiate by price: Proposal for a five-dimensional model*, European Management Journal, Available online 1 August 2012
[16] A. Watanabe, *Web Service Selection Algorithm Using Vickrey Auction*, 2012 IEEE 19th International Conference on Web Services (ICWS)

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

Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 251–267. http://www.scpe.org

ISSN 1895-1767
© 2012 SCPE

DATASTORES SUPPORTING SERVICES LIFECYCLE IN THE FRAMEWORK OF CLOUD GOVERNANCE

ADRIAN COPIE[†,*], TEODOR-FLORIN FORTIŞ[*,†], VICTOR ION MUNTEANU[*†], AND VIOREL NEGRU[*†]

Abstract. While adopting the Cloud, the small and medium sized enterprises (SMEs) could increase their benefit by embracing some Platform-as-a-Service (PaaS) solutions, doubled by a Cloud Governance or Cloud brokerage approach.

However, in order to fully exploit the advantages brought by Cloud environments, and enter in real competition with "big players" from different markets, SMEs must group themselves under the umbrella of a common marketplace, via some Cloud Governance solutions, and expose together complex, tailored and integrated solutions.

The implementation of an effective Cloud Governance solution requires a strong support for storing and manipulating data which is relevant for various aspects of applications, both at business and technical level, support which is closely linked to cloud service lifecycle.

This paper focuses on analyzing the main requirements related to data storage in an effective Cloud Governance system, details the functionality of the most important datastores and presents various use cases involving the entire storage architecture.

Key words: Cloud computing, Cloud governance, Cloud databases

AMS subject classifications. 68M14, 68P15, 68P20

1. Introduction. Built around the five core characteristics, as identified in [21, 25], and over a set of technical characteristics [18], including *service orientation*, *loosely coupling*, *strong fault tolerance*, *link with business models*, *virtualization*, or *ease of use*, Cloud Computing could offer the ideal environment for developing of new models of applications.

In order to fully exploit the new business models offered by the Cloud Computing [33, 34] – by which the Small and Medium-sized Enterprises (SMEs) could ease their migration to the cloud, benefiting from the delegated infrastructure management, high reliability and "pay-as-you-go" economic model – a strong support is required for easy development and deployment of applications (by adopting appropriate Platform-as-a-Service solutions), and (automatically) solve different aspects related with resource management (supported by a cloud management or cloud brokerage solution).

As the *vendor lock-in* problem occurred as a consequence of the large pool of vendor proprietary technologies, and insufficient standardization, different Platform as a Service (PaaS) solutions were considered in order to override the aforementioned difficulties. At the same time, PaaS solutions could enable uniform development, deployment and execution of cloud-enabled applications in various cloud environments.

They are usually built over the Infrastructure-as-a-Service (IaaS) layer, seldom on top of the Hardware-as-a-Service (HaaS) layer, and aims to be the support for the Software-as-a-Service (SaaS) layer, by providing a series of services, as identified in Table 1.1.

One can notice that the majority of the existent PaaS solutions address the resource management and scaling issues (mOSAIC [22, 11, 23, 24], Cloud Foundry [1], OpenShift [2], Morfeo 4CaaSt [3], ActiveState Stackato [4]), offer a development framework agnostic from the cloud providers (mOSAIC, Cloud Foundry, OpenShift, ActiveState Stackate, Morfeo 4CaaSt), address applications lifecycle management (mOSAIC, OpenShift, ActiveState Stackato) or provide a business ecosystem (Morfeo 4CaaSt).

While the different PaaS solutions offer support for applications development and deploying, there are different issues that require special attention through specialized developments, like resource management,service integration, or service orchestration.

The importance of these specialized developments was emphasized in different white papers [12, 8], offering the basic principles for cloud management. Based on these developments, different organizations proposed various *cloud reference models* [20, 30, 27]. However, in order to address the integration and orchestration

[*]West University of Timişoara, Faculty of Mathematics and Informatics, Computer Science Department, Blvd. V.Pârvan nr. 4, Timişoara, Romania (adrian.copie@info.uvt.ro, fortis@info.uvt.ro, vmunteanu@info.uvt.ro, vnegru@info.uvt.ro).

[†]Institute e-Austria, Timişoara (IeAT), Romania

[1]http://cloudfoundry.org

[2]http://openshift.redhat.com/app

[3]http://4casst.morfeo-project.org

[4]http://www.activestate.com/stackato

Service offered	mOSAIC	Cloud Foundry	OpenShift	Morfeo 4CaaSt	Active State's Stackato	WSO2 Stratos
Cloud resource management and scaling	yes	yes	yes	yes	yes	
Framework support		yes	yes		yes	yes
Application services		yes	yes		yes	
Application lifecycle management				yes		
Business ecosystem				yes		

Table 1.1: Services offered by different PaaS solutions

issues, different papers like [12] or [19] recommend a complementary set of services and a mechanism that is able to complement the cloud management limitations called in its entirety *Cloud Governance*.

The present paper is a direct extension of the work performed in [9]. The structure of this paper is as follows: Section 1 offers an introduction in the problematic of the paper. A motivation related to the present work is provided in section 2. In section 3 the architecture of the proposed Cloud Governance framework is detailed, while issues related to the Identity Management in cloud are discussed in section 4. The Services Datastore architecture is discussed in section 5 while the conculsions of the paper are established in section 6.

2. Motivation and related work. Cloud Governance also comes as a solution to the increasing demand of integration of different SaaS offerings [3, 7]. Integration and manipulation of data from various sources is thus an important issue that comes together with Cloud Governance, as a large variety of activities, like service management (publishing, brokering, monitoring, billing), security and others, are required. With a diversity of data types and usage patterns, a unitary approach for data storage cannot offer a feasible solution. With different datastores and catalogues, like the service, template, instance, or policy catalog, as identified in [12, 13], there is a need to identify specific requirements for each of these datastore categories.

This paper focuses on the storage layer in the context of a multi-agent architecture for a Cloud Governance environment aiming to support the service lifecycle, based on the work described in [12], extending previous work performed in [16, 26] for which it details the main components and presents various significant use-cases related to the service lifecycle.

While there is some resembling with the Service Oriented Architecture (SOA) in what concern the services storage paradigm, in service repositories, the present approach treats also the dynamic scaling issues, the services being dynamically distributed over many virtual machines in the cloud. The analyzed architecture pays attention to the issues raised by the privacy management and security as a central pillar in order to offer the SMEs a trustworthy environment in which they can migrate their businesses, in conformity with governmental laws and complying with the business standards.

At the same time, the existent literature and knowledge base related to the current Cloud Governance solutions does not present relevant information related to their persistence layer, so the storage architecture presented in this paper is our original contribution.

2.1. Cloud Management and Cloud Governance. Having a complex environment that allows the orchestration of SME's products in order to offer sophisticated solutions, raises important management challenges. The actual economic context, but also the financial principles lead the cloud providers to look for solutions in order to obtain significant cost savings through standardization of their basic operations. Provisioning the capacity of the servers as they will have an adequate functionality in the same time with an optimal performance is

a difficult task which will influence at the end the economic benefits of the cloud providers. These requirements are achieved through Cloud management solutions. According to [15] there are three core features that a cloud management system must fulfill:

- dynamically provisioning or deprovisioning of infrastructure resources
- ensures that the provisioned resources are securely used by the consumers
- offers a centralized way to plan, monitor, report and bill the infrastructure resources in cloud

Cloud Management solutions exist on an emergent market and in the last years many companies tried to provide dedicated products in order to solve specific issues. For example CA Technologies targets its solution called 3Tera [5] to a single platform, while Cloud.com [5] or Enomaly [6] have multiple target solutions for cloud management.

These solutions, however, are not enough when it comes to the enterprise solutions where different workloads must be processed, facing with issues like various ownerships, compliance, security and so on, requiring a Cloud Governance implementation. This implementation must take into account, according to [14] aspects like: *advanced user authentication*, *tight access control*, *encryption* and *budget management*. EnStratus [7] provides a cloud governance solution focused on important aspects of the enterprise cloud like access control, financial control, key management and encryption, logging and auditing. Also, WSO2 developed a framework able to support SOA Governance, called Governance Registry [35].

2.2. mOSAIC project. The FP7-ICT mOSAIC project (Open Source API and platform for multiple Clouds) aims to offer a friendly environment for writing and hosting applications in cloud, by freeing the developers by the 'vendor lock-in' paradigm. The API abstracts all the cloud providers intricacies, providing a unitary development platform and generic infrastructure resources in terms of computation, storage and network. The users of mOSAIC are able to specify their service requirements in a high level language using cloud ontologies and their resources are negotiated and brokered by a multi-agent system organized in a component called Cloud Agency (CA) [31], [32]. Cloud Agency has also responsibilities in terms of cloud management, being in charge with resource provisioning (buying them from cloud vendors), deploying them in the cloud and monitoring at the infrastructure level. CA is able to work together with the mOSAIC's platform or independent.

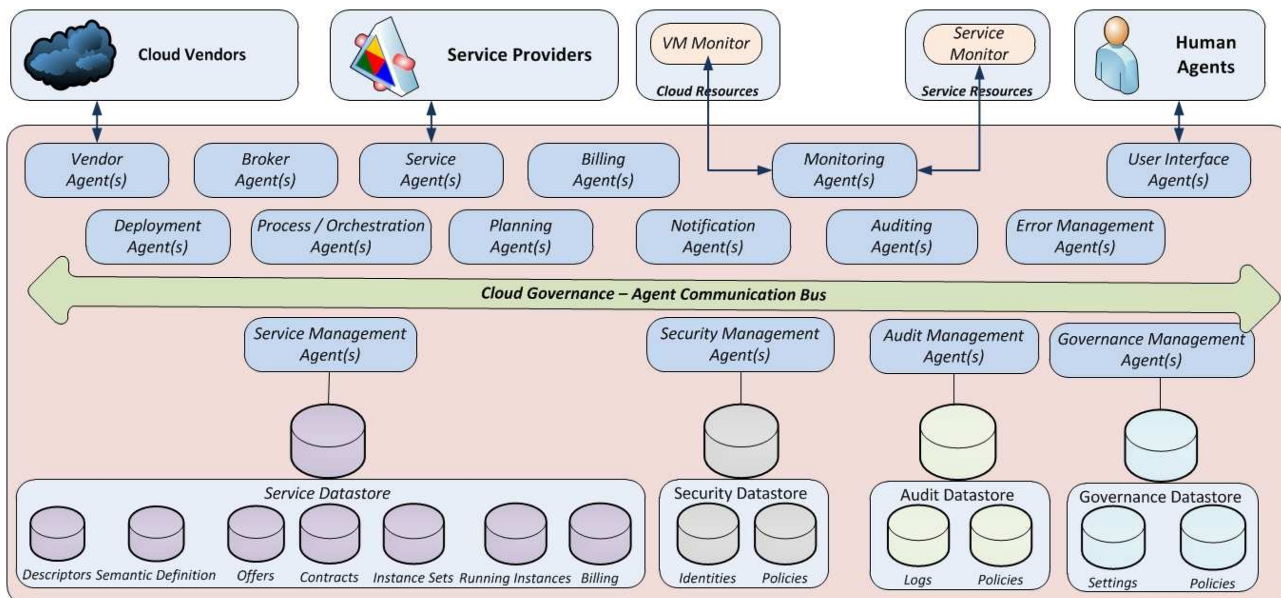


Fig. 3.1: Cloud Governance

[5]http://www.cloudstack.org/
[6]http://www.enomaly.com/
[7]http://www.enstratus.com

3. Architecture of a Cloud Governance system. Using mOSAIC's Cloud Agency component and taking into account the recommendations existing in the Distributed Management Task Force's (DMTF) white papers [12, 13] it is possible to build a Cloud Governance solution able to address the management of services and security, the service monitoring and auditing and offers support for services lifecycle. The proposed solution is a multi-agent system Figure 3.1 that have four specialized interdependent components:

Service Management, that handles all the phases related to the service lifecycle: publish, update, query, remove and instance handling. The requests for publishing a service inside the Cloud Governance Authority environment is directed to the Service Management agency which initiates the appropriate procedures. It is in charge with the service discovery process and takes care of the contracting and instantiation of the services.

Security Management, which deals with all the security issues like identity tracking, generating security tokens and validate them across the system. The Cloud Governance Authority is designed with security in mind, as being a crucial pillar targeted to cloud adoption by the SMEs. This is why a distinct agency dealing with security issues is provided.

The access to resources is performed in a secure way, based on various policies and access control lists, through specialized tokens which are attached to every request. Security Management agency issues the authentication requests to the underneath persistence layer and attaches the generated security tokens to the resource access requests.

Audit Management, for storing and analyzing the audit notifications sent by all the services inside the Cloud Governance environment, being able to raise alerts in case of system malfunctioning

The audit and monitoring inside the Cloud Governance Authority is performed at two distinct levels. The infrastructure level is monitored by the cloud management component which in this case is mOSAIC's Cloud Agency, while the services level is monitored by a specialized agency called Audit Management.

In order to fulfil the SLAs established in the contracting phase, the services are continuously monitored and alarms are raised through Cloud Agency Bus. Audition is also important since, based on the data produced by the audited services, further statistic or functional analyses can be performed.

Governance or *Configuration Management*, which is responsible with the whole system configuration, maintaining information about all the virtual machines running the agents and about the registered services. The Cloud Governance Authority is a complex mechanism composed from many physical or virtual machines and from many software components that are installed and configured to collaborate each other. The information related to the whole ensemble is managed by this dedicated agency which is permanently tied with the cloud management component.

3.1. Cloud Service Lifecycle. In order to have an effective Cloud Governance solution, is essential to offer a service lifecycle functionality which is able to support a service during all its phases in its life. Even that it follows some of the recommendations in [12] and is conceived in terms of Service Oriented Architecture (SOA) the approach of the Cloud Governance system is still different in the approach related to services distribution across many clouds and benefiting of the scaling offered by the cloud environment. When comes to design the architecture for services lifecycle, some important operations must be supported:

- *Design-time* operations: cover the syntactic and semantic service description, allowing the service to be published.
- *Provisioning* operations: cover the offer describing and service publishing, discovery and contracting
- *Deployment* operations: cover the service instantiation and commissioning
- *Execution* operations: cover the services management and billing
- *Retirement* operations: cover the services retirement There is a strong connection between the service lifecycle stages and the Cloud Governance datastores which is revealed in figure 3.2.

Different approaches for service lifecycle management were considered in order to "*manage the dynamic nature of the cloud environment*"[8], or to "*automate the service deployment and the dynamic provisioning of services*"[9] by solutions like BMC Cloud Lifecycle Management, WSO2 Governance Registry, Enstratus DevOps, or Morfeo Claudia. Also, service lifecycle in distributed environments is covered in different papers. In [28] an SCA design model was considered, while in [29] an application-centric lifecycle is offered.

[8] http://www.bmc.com/products/cloud-management

[9] http://claudia.morfeo-project.org/wiki/index.php/Service_Lifecycle_Manager

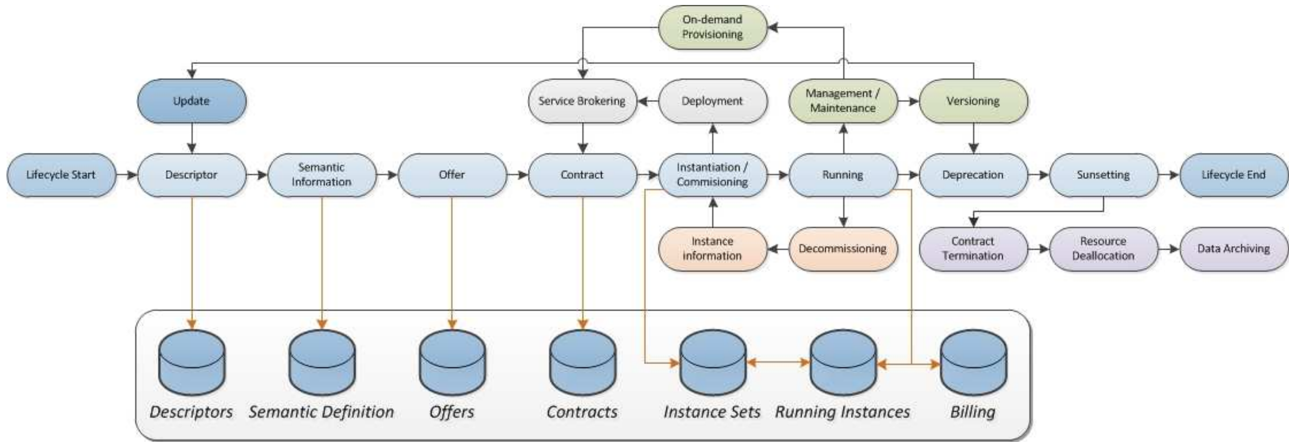


Fig. 3.2: Services Lifecycle

From the Cloud Governance architecture described in figure 3.1 and from the service lifecycle exposed in figure 3.2 four datastores were identified: *Services Datastore*, *Security Datastore*, *Audit and Billing Datastore* and *Governance* or *Configuration Datastore*.

3.2. Databases in Cloud Computing. The large number of services coexisting in a real environment, together with their dependencies, could lead to scalability problems for traditional databases which usually resides on a single server. A relative new scalability technique for relational databases, called sharding, could be used to keep a decent level of performance, but this is not satisfactory because adding or removing one or more server nodes lead to database restructuring, which is time consuming and degrades performance during the operation. In the same time adding more server nodes have as an immediate result the database partitioning, which makes the database system a distributed system.

According to [4], [17] in a distributed system, three essential properties for data cannot be achieved in the same time: *consistency*, *availability* and *partition tolerance*. However, not all the applications need support for all these properties in the same time and based on this assumption, many revolutionary databases have been developed by sacrificing one of the properties for the other two. More than this, some of these new databases, called in opposition to the traditional ACID (Atomicity, Consistency, Isolation, Durability) databases BASE (Basically Available, Soft state, Eventually consistent) are offered in cloud as a service, while some of them are available to be installed as open-source implementations in the private cloud. In this paper, some of the most important database types are analyzed in order to find the best matches for the problems raised by the datastores supporting cloud governance.

3.2.1. Key-Value stores. The key-value stores originate from a paper [10] published by Amazon which proposed a new type of database, highly available and highly reliable that can run on top of commodity hardware and with extremely good scalability while maintaining a high level of performance. Conform to [4] the consistency property is sacrificed for availability and partition tolerance, but this is acceptable for the applications using this kind of database.

The data model is very simple, it consist in a map or a dictionary in which values are assigned or retrieved in correspondence with unique keys, usually represented as strings. The value is perceived by the key-value store as a Binary Large OBject (BLOB) and it is the client's responsibility to interpret the content. The API interface for this datastore is very simple, exposing only few operations: *put*, *get* and *delete*.

The information related to cloud services is very heterogeneous, from structured data describing the services, formatted data resulted from monitoring and auditing processes, to unstructured data and binaries containing the executable code. Storing it in a key-value store involves complex keys management and also requires implementing a transactional mechanism that acts in an all-or-nothing manner for write operations.

Searching a key-value store is not supported by the database itself and it can be performed in a two-step process on the client side: first, all the keys have to be fetched and then their content must be interpreted which is degrades considerably the performance, especially for large data.

3.2.2. Document databases. The document databases represent a key-value stores evolution in terms of the complexity of the information they can store, multiple key-value pairs are allowed to be persisted in a document. The data types supported are the same as for the key-value stores, but the document allows a better information management.

Some of the existing document databases are even ACID compliant [2] fact that assures the data consistency during the Create/Read/Update/Delete (CRUD) operations over the services components. In the same time these database types allow to group more than one service related information inside a document or split it over many documents, offering a primitive linking mechanism between document fields.

3.2.3. Graph databases. The data in a graph database is represented by nodes, links and properties. This representation maps very well to the object oriented applications. The nodes are the object correspondences, where the links signify the relations between objects. Properties are additional information about the objects. The horizontal scalability is extremely good, being able to accommodate with billions of records. The graph databases are suitable for dynamical data since and they not enforce any rigid schema. Many of them are also ACID compliant.

3.2.4. Relational databases in the cloud. The continuous technological advances in cloud computing have shifted the focus to the non relational databases, many 'flavors' being developed, in order to solve specific problems. In the same time, the relational databases have passed through a rethinking process, emphasizing the scalability problems and the opportunity to offer them as a service in the cloud, paradigm called *NewSQL*.

The scalability provided by this paradigm is focused on read operations and is implemented in an elastic manner. From the services inside the cloud governance system, this database imposes constraints related to the structure of the data, which cannot be fulfilled all the time, but on the other side, the searching capabilities are very good supported.

4. Identity management. Cloud computing is an amalgam of technologies which makes possible on-demand resource provisioning with an unprecedented degree of availability and reliability while at the same time allow dynamic scalability. These internal technologies are stacked in various software layers that communicate and provide different services for the adjacent stack level. These layers are heterogeneous, usually they are produced by different vendors. In order to be orchestrated together and offer safe functionality, they have to obey various security rules and protocols implying the use of multiple ways of handling credentials.

This is not a trivial task since the services' environment is the cloud which can scale the infrastructure on multiple servers depending the workload facing the service. These workloads usually have different ownership, needs and compliances which make even harder the management of the entire system. It is the responsibility of a cloud governance system to solve these problems and to provide a unitary framework for the service creators which offer general solutions and for the service consumers which benefit from them.

The services inside a cloud governance system have multiple creators and use various security patterns for user authentication and authorization. More than this, in the most of the times, a customer has multiple accounts and implicitly multiple identities over a multitude of cloud services (e.g. e-bay, Amazon, Gmail, etc). It is an extremely difficult task from the consumer side to use multiple credentials for the different services inside the cloud. Some of them use credentials based on user name and password, some use various secret keys and some are using credit card numbers. A mechanism that can manage all the credentials in a unitary way along all the cloud computing environment is called *identity management* and it is a must for the governance framework.

Identity management is an integral part of cloud governance. Figure 4.1 introduces the identity management architecture used in our proposed cloud governance solution. The presented architecture is shown as high-level components:

The *Client Framework* is a component on the Client side. It facilitates authentication and authorization through a range of methods, tokens and protocols that are used when connecting to one of the *Communication Agents*. It enables communication using standard protocols using various authentication token technologies.

The *Communication Agents* are in charge of providing external communication with the Cloud Governance solution. This component is made out of three agents:

- *Vendor Agent* – is in charge of communicating with all cloud providers. Its main role is provisioning of cloud resources;
- *Service Agent* – is in charge of communicating with service providers as well as service instances. Its main role is to act as a proxy between them and the available cloud governance services;

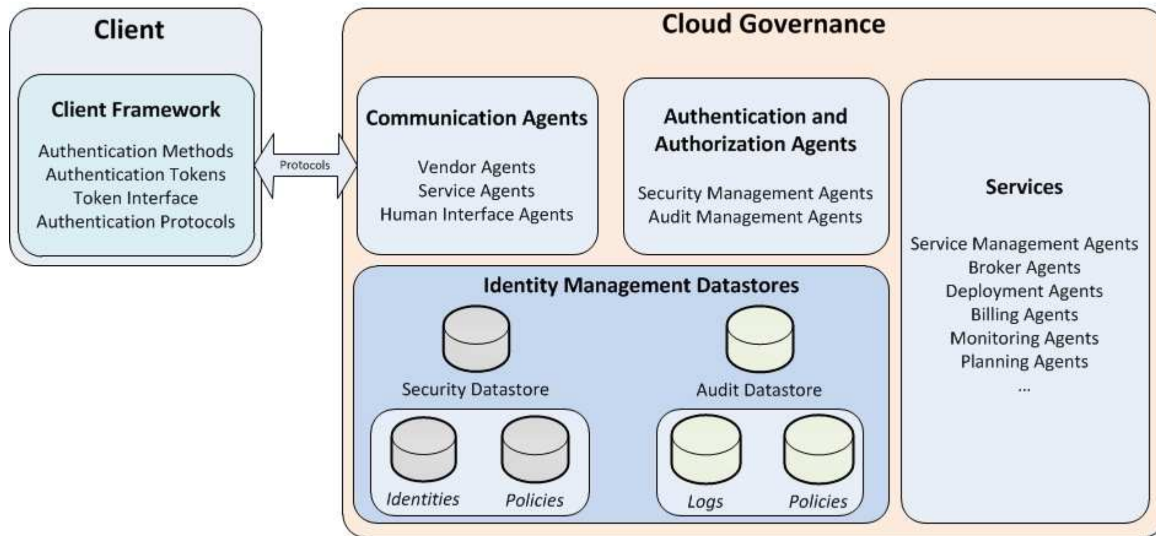


Fig. 4.1: Identity management architecture

- *Human Interface Agent* – is in charge of communicating with human clients. Its role is to enable human interaction with the system, both administrators an users needed for decision/expertise in certain tasks.

The *Authentication and Authorization Agents* handle identity related information like retrieving, generating, logging. They use the *Identity Management Datastores* through two agents:

- *Security Management Agent* – is in charge of validating different types of authentication credentials using information found in the *Security Datastores*. *Communication Agents* forward authentication credentials for validation and receive back authentication tokens that delegate the external party's authority to the agent to which it is connected. *Services* that are offered by cloud governance communicate with this agent in order to validate tokens and receive authorization for the requested actions.
- *Audit Management Agent* – receives and stores all user actions in the system like logins, actions etc. It also monitors them, and, based on policies, can trigger notifications for certain situations.

Services represent the agents performing different actions within the cloud governance solution. They require strong identity authentication and authorization in order to restrict unsanctioned actions by unknown entities.

Identity Management Datastores are distributed databases containing sensitive information about user identities, policies and logs. There are two main datastores:

- *Security Datastore* – stores sensitive user information like identity, credentials (passwords, tokens) as well as policies (permissions). Information related to service instances, cloud providers are also held here.
- *Audit Datastore* – stores information related to user/entity events within the solution like login/logout, service publishing etc., as well as policies which act as rules that, when met, trigger notifications within the system.

4.1. The Security Datastore. Security is a major concern for the users in a cloud environment, being the most important barrier to widespread adoption of cloud computing by the Small and Medium Enterprises. The presented Cloud Governance system meets these security requirements by implementing a complex security mechanism. The access to the services inside the Cloud Governance system is made full or in a granular way and must be performed in a secured manner, only authorized entities having the right to use them. The access rights are better implemented through various security policies that describe the interactions between the services and consumers.

Security Datastore is responsible with holding the credentials for all the participants in the Cloud Governance system, being them service providers or customers, together with their corresponding security policies. In order to access a certain service, the user must be first authenticated by comparing the provided credentials

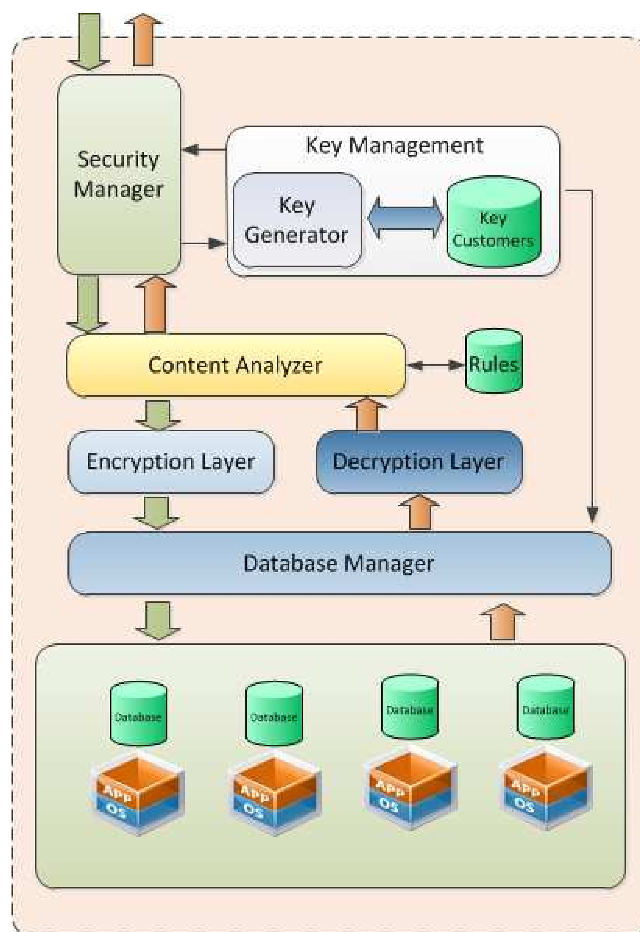


Fig. 4.2: Security Datastore

with the ones stored in the Security Datastore and then, a security token is generated to be used in further services access operations.

The Security Datastore, due its critical importance and extremely sensitive character of the stored data, must be hosted locally inside the Cloud Governance provider's location and not in the Cloud. All the information is encrypted using the AES-256 encryption protocol, and every entity has its own encryption key. More than this, sensitive information related to credit cards or other certificates is split and replicated across many distinct databases hosted on separate machines and recomposed at request, in order to protect the data from possible attacks or accidents. In case of such an incident, not all the records are stolen or lost.

4.2. The Audit Datastore. The mOSAIC's Cloud Agency component performs monitoring tasks for the infrastructure resources negotiated with the cloud providers, but this is not enough in the Cloud Governance context because the registered services need to be also the subject of monitoring in order to raise alerts when the Service Level Agreements (SLAs) are broken or other unpredictable events occurs. The Cloud Governance system offers this upper level monitoring through the Audit and Monitoring Management agent. The source of audit and monitoring data generates a big amount of data, in chunks of various sizes, depending the format exposed by the particular services. This data, once collected, is then refined and aggregated by specialized background processes, and moved in other databases in order to offer better visibility.

It is very important that this information won't be lost due the limited bandwidth to the database so the datastore must be highly writable, which is achieved by a key-value store. On the other hand, the key-value stores are not offering a search mechanism, which is important when data analysis must be performed, so an additional relational database is added in the datastore, to keep the searchable meta-information together

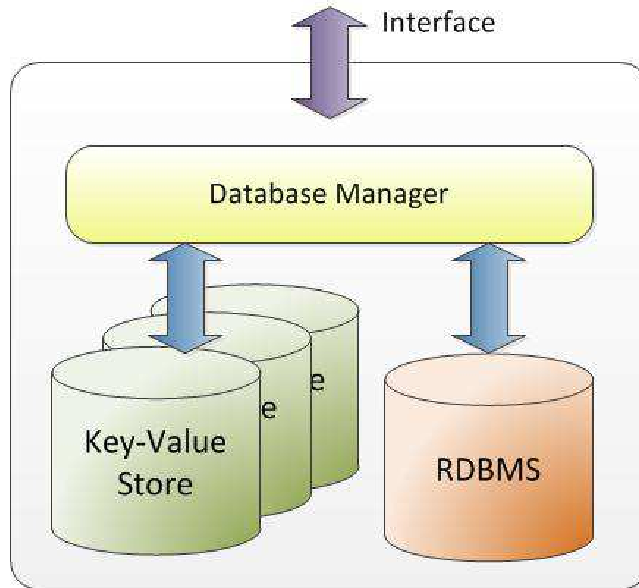


Fig. 4.3: Audit Datastore

with pointers to the keys in the key-value store. The correlation between these databases is made through a specialized software layer called Database Manager, which also implements a transactional mechanism to prevent the de-synchronizations.

5. The Services Datastore. This is the most complex datastore related to the Cloud Governance storage system and its purpose is to persist all the services existent inside the framework, offering in the same time the possibilities for the service providers to publish their final works and for the consumers to discover the services

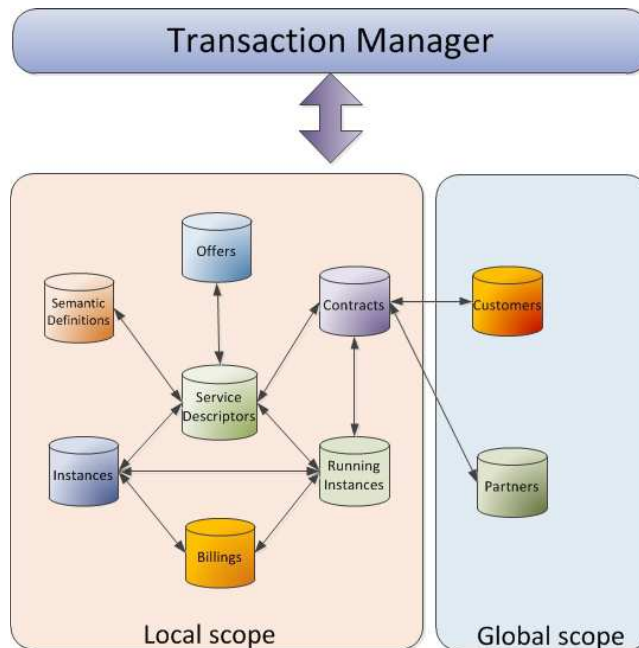


Fig. 5.1: Services Datastore

that best fit their needs and expectations.

The importance of a Service Datastore was emphasized in different white papers, including DMTF's Architecture for Cloud Management document [13], or the ITIL v3 specification, where the Service Catalogue was defined as an entity which "*provides a central source of information on the IT services delivered to the business by the service provider organization, ensuring that business areas can view an accurate, consistent picture of the IT services available, their details and status.*" [6].

5.1. Services Datastore components. The information related to the services is very heterogeneous, part of the data having a well defined structure, while other data being unstructured. All this information must be put in place and coordinated in order to offer a whole vision about the entire system, making possible the smooth functionality of the cloud governance process. In order to simplify the storage system but in the same time to offer a better performance, every kind of information is put in relation with a specialized database. These storage components have a local scope, being visible and accessible only from inside the Services Datastore.

Service Descriptors component contains information related to the services installed in the system. This is the core of the Services Datastore since it contains the description of the services in terms of functional (e.g. QoS, SLA) and non-functional parameters (e.g. Name, Creator, Version, Publisher, etc).

Semantic Definitions component contains information about the existing services, described from a semantic point of view. This information is used in the service discovery process and put in relation with the Service Descriptors which offers a syntactical characterization of the service.

Offers database is responsible with the service offers persistence. Every service must have an offer which better describes its capabilities together with the corresponding price.

Contracts database will hold the contracts resulted from a negotiation or contracting phase between the service provider and service consumer.

Instance Sets and Running Instances databases hold information about the existent services in the system. While Instance Sets contain information about all the service instances in the Cloud Governance, the Running Instances will keep only the services that are executing at a given time.

Billing database is crucial for the service invoicing process, since it keeps all the information generated as a result of service usage.

Complementary to these specialized databases, there are another two important databases called Customers and Partners which maintain relationships with the entities that are using the services and have a global scope visibility, being accessible from all the Cloud Governance components.

Because the Service Datastore is a mesh built from different databases of different types, it is necessary to have a component that is responsible for assuring the atomicity of the operations inside the service publishing use case, the consistency of the data over all the databases, the isolation of the operations and also the durability of the data over the entire datastore. Basically there is a need for a Transaction Manager (TM) that will assure the ACID properties of the Services Datastore as a whole. Due to the distributed character of the Services Datasore, the TM component must deal with distributed transactions which have to assure that all the databases are correctly changed after the operations involving the database modifications. In case of a failure occurred to one or more individual databases, all the changes are rolled back and the Service Datastore is brought to its initial state, in the pre-transactional state. The different types of the databases composing Services Datastore lead to the use of asynchronous transactions, the necessary time to perform the operations over the databases can vary based on the location of the storage, if it is installed locally or remotely, into the cloud and also depending of the amount of data to be written. The TM component is notified by the individual databases about the result of the operations and the transaction is committed only if all the individual database components have successfully performed their changes. If one or more operations fail, the changes are reverted to the initial state. The TM component is based on the X/Open Distributed Transaction Processing [1] specification and implements the two-phase commit protocol (2PC), which ensures that all the individual databases either committed the writes, either rolled back the changes to the pre-transactional state.

5.2. Services Datastore Use Cases. From the very beginning to the end of its existence during the lifecycle a service goes through multiple phases, having many states and interacting with many databases in order to achieve well established goals. In close relation with the Services Datastore, several relevant use cases, associated to lifecycle activities can be described in order to emphasize the interactions that could be considered for this datastore.

These use cases, rather simpler, where selected in order to emphasize the existent and necessary interactions

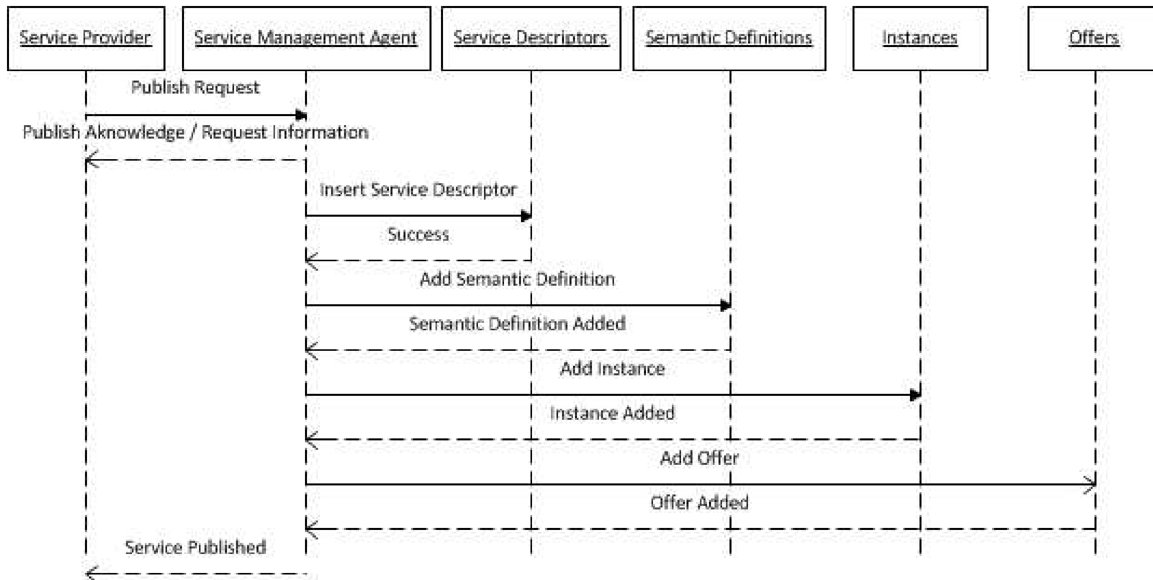


Fig. 5.2: Publishing a service

between the specialized databases composing the Services Datastore. The accent was placed on simplicity, just to keep the discussion at the datastores level and not at the highest level concerning cloud management and cloud governance operations.

5.2.1. Publishing a Service. This is a high level use case and it was selected to emphasize the interactions among some of the Services Datastore components and to reveal the write-intensive character of the operations, being in the same time the first contact between a service provider and the Cloud Governance environment.

In order to be part of the Cloud Governance Authority, a service must conform with a publishing interface which establishes the format and the content of the data that must be provided at the design time by the service creators, like the description of its characteristics, what does it offer, dependencies and price. It is essential that this data exists at the beginning of the publish operation, otherwise the service publishing will fail.

Figure 5.2 reveals the simplified steps followed by the framework to publish a service. All this information is divided, in order to be persisted, in various components of the Services Datastore. The service providers make a request for publishing a service which is received by the Service Management Agent, the entity in charge with the interface to the Services Datastore. The details of this use case together with the interacting actors are presented in Table 5.1 where also information related to the fail conditions are presented.

All the operations over the Services Datastore are performed through the Transaction Manager (TM) component which assures the consistency and integrity of all the composing databases.

When a publish request is received by the Service Management, it will be forwarded to the TM which will assign a transaction identifier for all the operation inside that service publishing. Every operation that changes the content of one of the composing databases must be communicated to the TM which will decide to rollback the entire transaction in case of an operation failure.

5.2.2. Discovering and Contracting a Service. This is a high level use case that shows various interactions between the Services Datastore components with the accent on read-write operations.

This is the first interaction of a service consumer with the Cloud Governance framework. Usually a consumer is interested to find a specific service that meets some characteristics and involving certain constraints related to running costs. Figure 5.3 presents the simplified steps followed by the system to allow a consumer to discover and contract a service. Table 5.2 details all the steps followed in order to discover a service and contract it by a service consumer.

5.2.3. Instantiation and Commissioning a Service. Another consumer interaction with the Cloud Governance system is the instantiation and commissioning phase, as identified in Figure 5.4, in which a service

Use Case: Publishing a service	
Description	
Description and Goal	This use case refers to the service publishing operation, having as result the service available for the consumers over the Cloud Governance environment
Dependencies	None
Actors	Service Provider, Service Management Agent, Service Descriptor, Semantic Descriptor, Offers, Instances
Assumptions and Triggering Events	
Preconditions	All the required information related to the service being published must exist
End conditions	
Success end condition	All the service related information is in the specialized databases, the service is ready for discovery phase
Failure condition	One or more databases signals errors during the change process, the entire distributed transaction is rolled back
Main success scenario	
Normal flow	1. The Service Provider issues a Publish request to the Service Management component, signalling that it wants to publish a service 2. The Service Management responds with a Publish Acknowledge which is a request for the information necessary for a service to be published 3. The Service Management Agent passes the service functional and non functional information to a Broker Agent which is in charge to insert it in the Service Descriptors database. 4. The Service Descriptor reports the result of the operation to the Service Management 5. The semantic definition is passed to the Broker Agent stored in the Semantic Definitions database 6. The Semantic Definitions reports the result of the operation to the Service Management 7. The Service Management passes the information related to the service instance and its dependencies to a Broker Agent which will add them in the Instances database 8. The Instances database reports the result to the Service Management component 9. The information related to the service capabilities and prices is passed to the Broker Agent which adds them in the Offers database. 10. The Offers component reports the result of the operation to the Service Management 11. The Service Management centralizes all the partial results from the Service Datastore components and if all of them are successful, the transaction is committed and the Service Provider is informed that its service was published.

Table 5.1: Publishing a Service Use Case

is launched in execution. This is a high level use case, relevant for the interaction between the components of the Services Datastore, focusing on read-write operations. The details of the use case are revealed in Table 5.3 which presents also all the actors involved in these operation together with the pessimistic scenarios in case of failure.

Use Case: Discovering and Contracting of a Service	
Description	
Description/Goal	This use case refers to the service discovery and contracting operation, having as result the service available to be run in the Cloud Governance environment
Dependencies	Service Publishing completed
Actors	Service Consumer, Service Management Agent, Semantic Definitions, Offers, Contracts
Assumptions and Triggering Events	
Preconditions	The service must be successfully published in a previous step
Postconditions	
Success end condition	The service is successfully discovered, contracted and its instance is ready to be lunched in execution
Failure condition	One or more databases signals errors during the change process, the entire distributed transaction is rolled back
Main success scenario	
Normal flow	1. The Service Consumer makes a request for a specific service to the Services Management with the information about the type of the requested service. 2. The request is transformed by the Service Management into a SPARQL query and passed to a Semantic Definitions to be interrogated 3. The Semantic Definition returns the result of the query to the Service Management 4. The Service Management component passes the result back to the Service Consumer entity which will decide if the service meets its expectations and if further actions will follow. 5. If the Service Consumer entity decides that the discovered service meets its expectation, it informs the Service Management that it is willing for the next step 6. The Service Management queries the Service Descriptor database following the criteria established in the previous step to retrieve the syntactic components of the service 7. The syntactic information is returned to the Service Management which start to compose a local image of the service 8. The Service Management issues a query to the Offers database to retrieve the price of the service 9. The result of the interrogation is returned to the Service Management 10. Service Management passes the result to the Service Consumer, informing it about the price of the service 11. Service Consumer decides if agrees with the service's price and if yes, it sends to the Service Management an Offer Accepted message 12. Service Managements issues an action, which is in fact a modification query, to the Contracts database having a value of contract signature 13. The Contracts database sends back to the Services Management a Service Contracted message 14. The Service Management then informs the Service Consumer about the signed contract.

Table 5.2: Discovering and Contracting a Service Use Case

Use Case: Instantiation and Commissioning a Service	
Description	
Description and Goal	This use case refers to the instantiation and commissioning of a service, having as result the service and executed inside the Cloud Governance environment
Dependencies	None
Actors	Service Consumer, Service Management Agent, Security Management Agent, Security Datastore, Instances, Running Instances, Audit and Billing Management, Billings
Assumptions and Triggering Events	
Preconditions	The service being instantiated must be already published
End conditions	
Success end condition	The service is instantiated and executed, billing information is generated
Failure condition	One or more databases signals errors during the change process, the entire distributed transaction is rolled back
Main success scenario	
Normal flow	1. The Service Consumer issues an Instantiation request to the Service Management component, signalling that it wants to instantiate a service 2. The Service Management passes the request to the Security Management Agent which is in charge with the authorization and authentication operations 3. The Security Management interrogates Security Datastore about the identity of the requester and if successful, it generates the security token that will be attached to all the resource requests 4. The security token is returned by the Security Datastore to the Security Management 5. The Security Management returns the identity and security token back to the Services Management 6. The Services Management issues a query in the Instances database, looking for the specified service. 7. If the specified service is found, it is returned back to the Service Management 8. Having the identity confirmed, a valid security token and an existent service, Service Management is now able to instantiate the service and to issue a query that adds the new running service in the Running Instances database. 9. The result about the new entry in the Running Instances is returned to the Services Management 10. Services Management notifies the Audit and Billing Management component that a new service was started, to produce the information necessary for invoicing the customer. 11. The Audit and Billing Management start to generate billing information and sends it to Billings database. 12. The Billing component responds to the Audit and Billing Management about the result of the operations 13. The Audit and Billing Management informs Services Management that the billing started successfully 14. The Service Management notifies the Service Consumer about teh success of the Instantiation and Commissioning operation

Table 5.3: Instantiation and Commissioning a Service Use Case

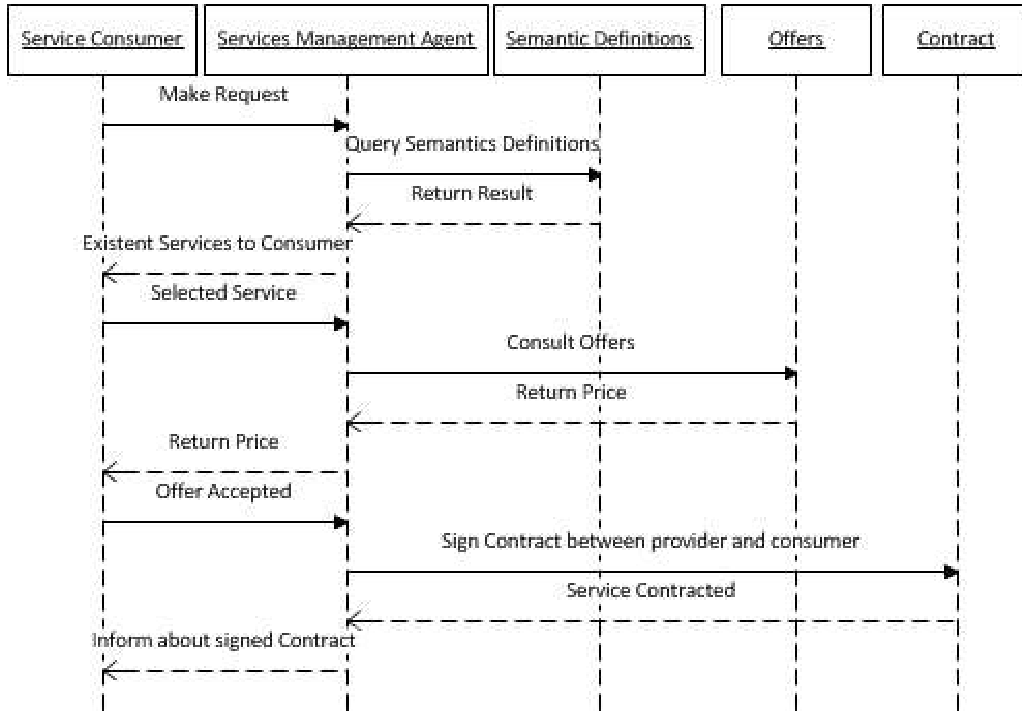


Fig. 5.3: Discovery and Contracting a Service

6. Conclusions and future work. Cloud computing is an evolutionary step in the IT domain, that promises new business and economic models which can be embraced by the SMEs in order perform cost savings and in the same to collaborate in offering complex solutions on an emergent and dynamic market. However, to sustain this virtual collaboration, sophisticated Cloud Governance frameworks that solve the cloud management issues, together with authentication, security, billing and more must be developed. Some companies already integrated cloud governance functionalities in their products, but they do not offer a complete cloud governance solution.

This paper addressed one fundamental aspect of cloud governance: the storage system with its specialized datastores and a series of use cases that emphasize their interaction. The principal existent solutions related to cloud management and cloud governance were identified and also the main database types used in cloud. The architecture of the proposed Cloud Governance system was revealed and the subsystem composing the datastores were detailed. This research conducted in this paper will have fundamental importance in building a working prototype.

Acknowledgments. This work was partially supported by the grant of the European Commission FP7-ICT-2009-5-256910 (mOSAIC), and Romanian national grant PN-II-ID-PCE-2011-3-0260 (AMICAS). The views expressed in this paper do not necessarily reflect those of the corresponding projects' consortium members.

This article is also partially a result of the project "Creşterea calităţii şi a competitivităţii cercetării doctorale prin acordarea de burse"[10]. This project is co-funded by the European Social Fund through The Sectorial Operational Programme for Human Resources Development 2007-2013, coordinated by the West University of Timisoara in partnership with the University of Craiova and Fraunhofer Institute for Integrated Systems and Device Technology - Fraunhofer IISB.

REFERENCES

[10]POSDRU/6/1.5/S/14 Increase the attractiveness, quality and efficiency of university doctoral studies by doctoral scholarships;

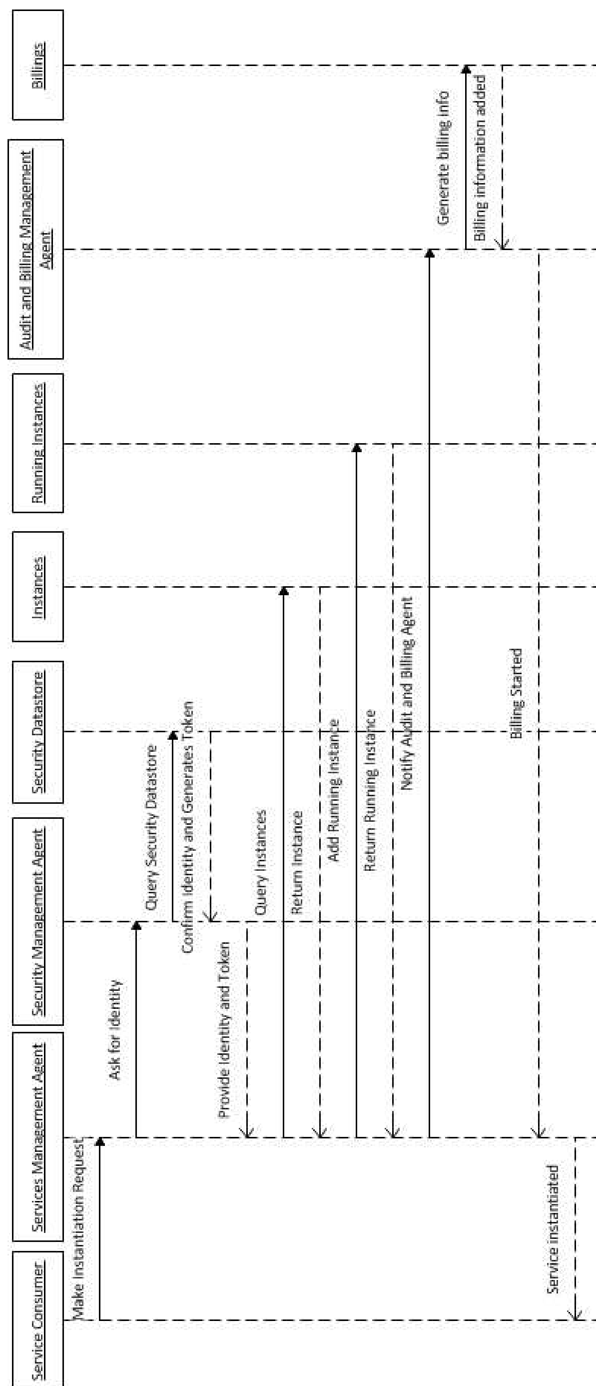


Fig. 5.4: Instantiation and Commissioning a service

[1] *Distributed transactions processing: the xa specifications* http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf, November 1998.

[2] *Apache CouchDB*. http://http://couchdb.apache.org/, October 2012.

[3] S. Bennett, T. Erl, C. Gee, R. Laird, A. T. Manes, R. Schneider, L. Shuster, A. Tost, and C. Venable, *SOA Governance: Governing Shared Services On-Premise & in the Cloud*, Prentice Hall/PearsonPTR, 2011.

[4] E. A. Brewer, *Towards robust distributed systems*. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, July 2000.

[5] CA Technolofies, *Cloud Management*. http://www.ca.com/us/cloud-platform.aspx, August 2010.
[6] A. Cartlidge, A. Hanna, C. Rudd, I. Macfarlane, J. Windebank, and S. Rance, *An introductory overview of ITIL® v3*. http://www.itilbookshop.org/Documents/an_introductory_overview_of_itil_v3[0].pdf, 2007.
[7] T. Cecere, *Five steps to creating a governance framework for cloud security*. Cloud Computing Journal http://cloudcomputing.sys-con.com/node/2073041, November 2011.
[8] Cloud Computing Use Cases Group, *Cloud computing use cases white paper*. http://opencloudmanifesto.org/Cloud_Computing_Use_Cases_Whitepaper-4_0.pdf, July 2010.
[9] A. Copie, T.-F. Fortis, V. I. Munteanu, and V. Negru, *Service datastores in cloud governance*, in ISPA, IEEE, 2012, pp. 473–478.
[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's highly available key-value store*, SIGOPS Oper. Syst. Rev., 41 (2007), pp. 205–220.
[11] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, and M. Loichate, *Building a mosaic of clouds*, in Euro-Par 2010 Parallel Processing Workshops, M. Guarracino, F. d. r. Vivien, J. Traff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knapfer, B. Di Martino, and M. Alexander, eds., vol. 6586 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 571–578.
[12] DMTF, *Architecture for Managing Clouds*. http://dmtf.org/sites/default/files/standards/documents/DSP-IS0102_1.0.0.pdf, June 2010.
[13] DMTF, *Use Cases and Interactions for Managing Clouds*. http://www.dmtf.org/sites/default/files/standards/documents/DSP-IS0103_1.0.0.pdf, June 2010.
[14] enStratus, *Enterprise Cloud Governance White Paper*. http://www.enstratus.com/page/1/ecg-wp-track.jsp, August 2010.
[15] C. Escapa, *Cloud Management Guide*. White paper http://www.abiquo.com/files/white_paper_cloud_management_guide.pdf, August 2010.
[16] T.-F. Fortis, V. I. Munteanu, and V. Negru, *Steps towards cloud governance. a survey*, in ITI, 2012, pp. 29–34.
[17] S. Gilbert and N. Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, SIGACT News, 33 (2002), pp. 51–59.
[18] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, *The characteristics of cloud computing*, in ICPP Workshops, W.-C. Lee and X. Yuan, eds., IEEE Computer Society, 2010, pp. 275–279.
[19] G. Gruman, *Integration issues may hinder saas adoption*. CIO Update http://www.cioupdate.com/trends/article.php/3695096/Integration-Issues-May-Hinder-SaaS-Adoption.htm, August 2007.
[20] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, *NIST cloud computing reference architecture*. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/ReferenceArchitectureTaxonomy/NIST_SP_500-292_-_090611.pdf, September 2011.
[21] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*. White paper http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, September 2011.
[22] mOSAIC Consortium, *The mOSAIC project*, October 2012.
[23] F. Moscato, R. Aversa, B. D. Martino, T.-F. Fortiş, and V. Munteanu, *An analysis of mosaic ontology for cloud resources annotation.*, in FedCSIS, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, eds., 2011, pp. 973–980.
[24] F. Moscato, R. Aversa, B. D. Martino, D. Petcu, M. Rak, and S. Venticinque, *Cloud computing: methodology, system, and applications*, CRC, Taylor & Francis group, 2011, ch. An Ontology for the Cloud in mOSAIC.
[25] A. Mulholland, J. Pyke, and P. Fingar, *Enterprise Cloud Computing: A Strategy Guide for Business and Technology Leaders*, Meghan-Kiffer Press, Tampa, FL, USA, 2010.
[26] V. I. Munteanu, T.-F. Fortis, and A. Copie, *Building a cloud governance bus*, International Journal of Computers, Communications and Control, 7 (2012), pp. 888–894.
[27] NIST, *Cloud architecture reference models: A survey*. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/Meeting4AReferenceArchtecture013111/NIST_CCRATWG_004v2_ExistentReferenceModels_01182011.pdf, January 2011.
[28] C. Ruz, F. Baude, B. Sauvan, A. Mos, and A. Boulze, *Flexible SOA lifecycle on the cloud using SCA*, in Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International, 29 2011-sept. 2 2011, pp. 275 –282.
[29] K. Tang, J. M. Zhang, and C. H. Feng, *Application centric lifecycle framework in cloud*, E-Business Engineering, IEEE International Conference on, 0 (2011), pp. 329–334.
[30] T. Tung, *Defining a cloud reference model*, in Cluster Computing and the Grid, IEEE International Symposium on, IEEE Computer Society, 2011, pp. 598–603.
[31] S. Venticinque, R. Aversa, B. Di Martino, M. Rak, and D. Petcu, *A cloud agency for SLA negotiation and management*, in Proceedings of the 2010 conference on Parallel processing, Euro-Par 2010, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 587–594.
[32] S. Venticinque, R. Aversa, B. D. Martino, and D. Petcu, *Agent based Cloud Provisioning and Management - Design and Prototypal Implementation*, in CLOSER 2010, 2011, pp. 184–191.
[33] C. Weinhardt, A. Anandasivam, B. Blau, N. Borissov, T. Meinl, W. Michalk, and J. Stösser, *Cloud Computing–A Classification, Business Models, and Research Directions*, Business and Information Systems Engineering (BISE), 1 (2009), pp. 391–399. ISSN: 1867-0202.
[34] C. Weinhardt, A. Anandasivam, B. Blau, and J. Stösser, *Business Models in the Service World*, IEEE IT Professional, Special Issue on Cloud Computing, 11 (2009), pp. 28–33. ISSN: 1520-9202.
[35] WSO2, *Governance Registry*. White paper http://wso2.com/products/governance-registry/, August 2010.

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

Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 269–284. http://www.scpe.org
ISSN 1895-1767
© 2012 SCPE

A DISTRIBUTED PROGRAM GLOBAL EXECUTION CONTROL ENVIRONMENT APPLIED TO LOAD BALANCING

JANUSZ BORKOWSKI*, DAMIAN KOPAŃSKI*, ERYK LASKOWSKI†, RICHARD OLEJNIK‡, AND MAREK TUDRUJ†*

Abstract. The paper is concerned with a new distributed program design environment based on the global application states monitoring. The environment called PEGASUS (from Program Execution Governed by Asynchronous SUpervision of States) supplies to a programmer a ready to use control primitives to design distributed program execution control in which decisions for synchronous and asynchronous control actions are based on predicates evaluated on global application states. Such strongly consistent global application states are automatically constructed by the run-time system which additionally provides mechanisms for their analysis and organizing the respective program execution control in processes and threads of user programs executed in multicore processors. The PEGASUS control mechanisms are graphically supported in the respective program design framework. The paper first presents main general features of the PEGASUS environment. Next, it presents a method for load balancing inside distributed programs based on a set of parameters which are dynamically measured during program execution. Then, the paper presents how the described load balancing method can be implemented inside the PEGASUS environment taking as an example distributed programs for solving the Traveling Salesman Problem (TSP).

Key words: distributed program design tools, global application states monitoring, load balancing;

1. Introduction. Distributed programs execution control based on the monitoring of global application states is an important problem in designing many distributed application programs. In recent years, many modern distributed program frameworks were published, which propose new paradigms in the design of distributed programs and systems. As their number is too big to enumerate them all in this paper, we will point out only some of them: active objects as in the ProActive framework [1], message-driven objects as in Charm++ framework [2], new virtualization approach to provide software for chipset services in systems of multicore processors as in vSMP framework of ScaleMP [3], a task based data flow programming model as in StarSS [4], which hides much of internal parallelism from programmers. The proposed solutions put also emphasis on different aspects of the distributed program and system design like FELI support for CMP-based systems [5], which automatically allocate application data to on-chip memories without user programming. All the cited solutions contribute to the development of the design methods for systems based on multicore processors, however, they do not contain any support for the distributed program execution control based on the global application states monitoring.

In the history of programs execution management there were initial tries to include global states monitoring in the execution control but not developed into programming environments meeting common standards. Linda environment [6] was based on a common global tuple space for the exchange of globally available control information. The user processes could write and read in the tuple space. Initial primitives for the design of global control for interactive software components were included in coordination languages. In Manifold and Reo frameworks [7] primitives for inclusion of communicating software components into coordinated structures were embedded. This was done without any notion of a global state introduced in these systems. In the Meta system [8] distributed programs could be designed based on communicating components with the use of a notion of a global application state. In this system, application processes could send messages to a global monitor on their local states. Consistent global states were constructed on these states and some forms of global predicates could be evaluated. In Meta, a complicated formal framework based on guards could construct program control based on global states. Some simplification of the Meta formalism was attempted in the Lomita language [8]. However, it was not successfully efficient due to very costly message broadcasts of ordered states. Global control constructs for the OCCAM language were proposed in [9] with an implementation based on replication of global state variables. The first legacy usable framework for the asynchronous global execution control in distributed programs based on monitoring of global application states was implemented in a graphical parallel program design system PS–GRADE [10] for construction of distributed programs in the C language and standard

*POLISH-JAPANESE INSTITUTE OF INFORMATION TECHNOLOGY, UL. KOSZYKOWA 86, 02-008 WARSAW, POLAND ({JANB,DAMIAN.KOPANSKI,TUDRUJ}@PJWSTK.EDU.PL)
†INSTITUTE OF COMPUTER SCIENCE POLISH ACADEMY OF SCIENCES, WARSAW, POLAND ({LASKOWSK, TUDRUJ}@IPIPAN.WAW.PL)
‡COMPUTER SCIENCE LABORATORY OF LILLE (UMR CNRS 8022), UNIVERSITY OF SCIENCES AND TECHNOLOGIES OF LILLE, FRANCE ({RICHARD.OLEJNIK}@LIFL.FR)

communication libraries: PVM and MPI.

The features of the PS–GRADE and other distributed program design frameworks based on global application states monitoring have been conceptually and practically extended in PEGASUS (from Program Execution Governed by Asynchronous SUpervision of States) which is a basis for this paper. The main extension is the graphically supported distributed program control flow design framework in which the flow of control depends on the predicates computed on distributed application global states.

This paper discusses the use of the facilities of global state monitoring and program execution global control available in the PEGASUS environment to organize load balancing in distributed applications [11]. The features of the PEGASUS framework are original in this sense since, to our knowledge, no operational contemporary environment provides similar ones.

The PEGASUS environment provides global program execution control constructs for distributed programs. They assume a modular structure of parallel programs based on the notions of processes and threads. The PEGASUS global control constructs enable global control of the synchronous and asynchronous character. For synchronous program execution control the constructs logically bind program modules and define the involved global control flow which depends on the application global states. For the asynchronous program execution control, the PEGASUS control constructs enable asynchronous change of process and threads behaviour based on global application states. The monitoring and the management of the global application states in PEGASUS is implemented by the use of special control processes called the synchronizers which automatize the implementation of the program execution control based on global application states. The synchronizers collect local state information from processes and threads, automatically construct global strongly consistent application states, evaluate relevant control predicates on global states and provide a distributed support for sending control Unix-type signals to distributed processes and threads to stimulate the desired control reactions. The repertoire of considered local and global states, the control predicates and the reactions to them are user programmed using a special API provided in the system. It is the run-time system, which provides the necessary implementation of this control model. The design of the global control flow is graphically supported and the provided GUI is decoupled from the API for data processing inside modules. The proposed global control constructs enable better verification and are less error prone.

The PEGASUS framework aims at both hardware and software features to prevent global control overheads. The hosting system for PEGASUS has a triple communication network, built of three separate parallel networks which are serviced by three separate communication libraries whose implementation is not conflicting. One network is used for interprocessor computational data communication (Ethernet). Another network (Infiniband) is used for control data communication including transfers of state information and program execution control signals. The third network (Fast Ethernet) is used for communication involved in processor clock synchronization necessary for detection of strongly consistent application global states.

The contribution of this paper is to show how the necessary control for dynamic load balancing can be implemented under the PEGASUS framework. The PEGASUS framework seems to be especially convenient to design dynamic load balancing algorithms in user programs. For the load balancing under PEGASUS for programs in the C/C++ language, we have adapted the iterative method which has been earlier designed by us for load balancing of object-oriented programs in the Java language [12]. However, the global control infrastructure of PEGASUS enables designing load balancing support for practically any of load balancing methods. A good overview of the known load balancing methods and principal load balancing systems is given in [13].

Under PEGASUS, all load balancing control (except for sending load states reports and receiving work control signals by worker threads) is done by synchronizers in the background of the primary computations (in parallel with them). The synchronizers (organized in a hierarchical distributed structure) receive load reports, judge the load imbalance, take decisions upon balancing operations and instruct the worker processes and threads how to continue computations, including the migration of the computational load. The direct contacts of worker threads with the infrastructure of synchronizers is by shared memory inside processors, so it is fast and scarcely disturbing the thread's computational work since the load balancing control is done asynchronously not causing any busy waiting. Only transfers of control data at higher levels of the synchronizer tree is done by message passing. But this is done after strong reduction of control data at lower levels of the synchronizer tree and it is done only when the load balancing decisions can not be taken at these lower levels. In fact, such transfers for global decisions are unavoidable in any load balancing distributed infrastructure, so, in this respect, the performance of load balancing under PEGASUS is not worse than in other systems. However,

in PEGASUS, a programmer has a ready to use convenient global control infrastructure based on automatic construction of global consistent load state to design and master load balancing for distributed computations. The presented approach partially leverages also our earlier works, reported in [14, 15].

The paper consists of three parts. In the first part the features of the assumed PEGASUS program design and execution environment are described. In the second part, the principles of the proposed load balancing strategy, implemented using global application states monitoring are presented. The third part describes the design of an exemplary application which is the algorithm for load balancing in solving the Traveling Salesman Problem by the Branch & Bound method.

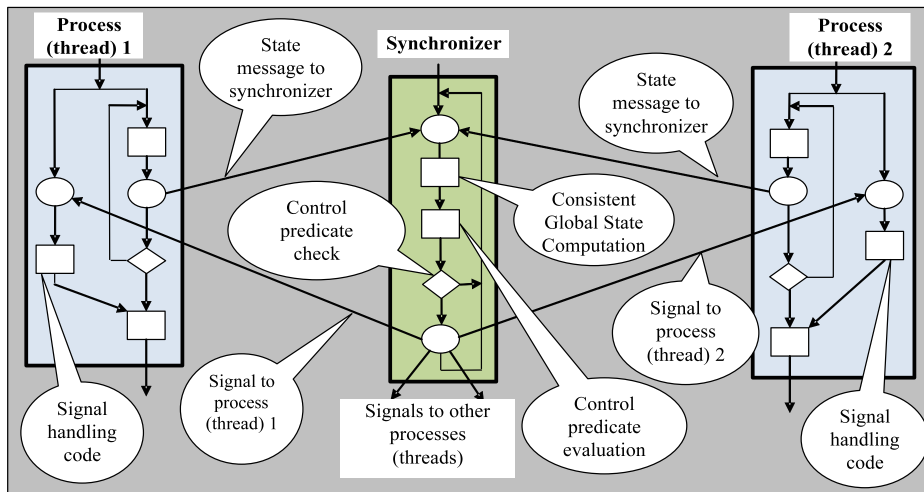


Fig. 2.1: Processes (threads) co-operating with a synchronizer.

2. Distributed program execution control based on global application states . The PEGASUS environment enables designing distributed programs written in the C/C++ languages built of processes and threads in which program execution control is governed by the global states of the entire distributed application. The environment works under control of the Linux operating system and provides the run-time framework for the designed programs. The PEGASUS environment includes a built-in programming infrastructure which supports designing program global execution control based on predicates computed on global application states. Program execution control under PEGASUS framework has features of the asynchronous and synchronous control. The asynchronous control enables constructing internal process and threads behavior which depends on the global application states. It corresponds to a local process/thread control by global states. It is organized by process/thread reporting reporting of local states to special monitors of global states which reconstruct global application states and organize reactions to control signals issued after an analysis of global states.

The synchronous control enables constructing program execution control in which the flow of control between program modules depends on the global application states, discovered by the mentioned above global state monitors. The synchronous control paradigm includes gathering local states responsible for decisions on global control flow inside programs, computing control predicates on global states and issuing signals to the flow of control switching blocks in programs. in response to the signals the switches direct the flow of control in the program in desired way.

The design of the program execution control in PEGASUS is graphically supported. The system enables Graphical User Interface for designing distributed program control flow graphs using global high level control flow primitives which are global application states sensitive. The graphical phase of design is further supported by the Application Program Interface to define the attributes of the global control flow. The system provides also the API to design asynchronous control of design the process and thread programs whose behavior is depending in the asynchronous way on the global application states.

The executive distributed systems are assumed to be built of multicore processors interconnected by a message passing network. The network is used for interprocessor computational data communication at the

process/thread levels with the use of standard message passing MPI library. For computational data communication between threads inside multicore processors communication by shared memory is used, supported by the OpenMP and Pthreads libraries. The implementation of global program execution control requires control data communication between processors and processor cores. This communication is decoupled from the computational data communication and is implemented using separate software and hardware means. The programmer specifies the variables which are to be used as attributes of this control using special API. It is the system which assures the necessary software/hardware communication environment for respective control data transfers and automatically generates the necessary fragments of program code. His code uses additional dedicated networks for control data transfers necessary for implementation of global control mechanisms. A programmer is able to control assignments of processes to processor nodes and threads to processor cores.

2.1. Asynchronous model of program execution control based on global states.. The asynchronous program execution control is based on special control processes called the synchronizers introduced to the API and GUI of distributed programs. The general scheme for organizing the asynchronous program execution control is shown in Fig. 2.1.

Application program processes (threads) send messages on their *states* to special globally accessible processes (threads) called *synchronizers*. A synchronizer collects local state messages, determines the application's strongly consistent global state, evaluates control predicates and stimulates desired reactions to the signals in application components.

A strongly consistent global state (SCGS) means a set of fully concurrent local states detected unambiguously by a synchronizer [16]. Processor node clocks are synchronized with a known accuracy to enable the construction of strongly consistent global states by projecting the local states of all processes or threads on a common time axis and finding time intervals which are covered by recognized local states in all participating processes or threads [17].

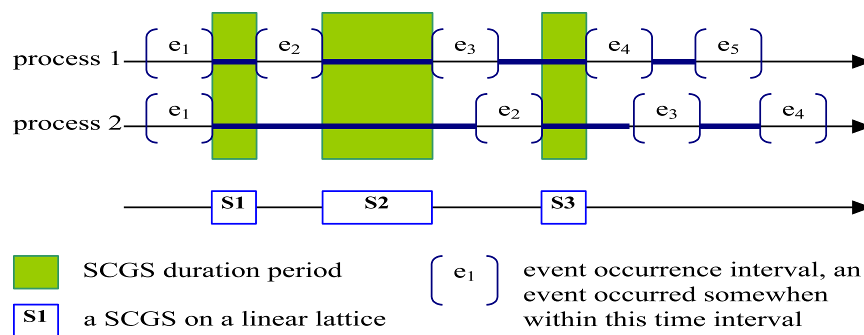


Fig. 2.2: Detection of strongly consistent global states.

The way of reconstructing strongly consistent global states from messages on local process/thread states is shown in Fig. 2.2. The synchronizers receive messages on local states by means of messages on starting and ending events of the states accompanied by time stamps which are given with the accuracy of the local clocks synchronization in processors. The time stamps are treated by the SCGS detection algorithms as time intervals with the event appearance time points expressed with the accuracy equal to the clock synchronization skew. For multicore processors, the threads issue state messages with the time stamps based on the processor clock shared by all existing threads. The strongly consistent states are those which are not covered by the uncertainty interval of events. A strongly consistent state for a group of processes/threads appears between time points not covered by any uncertainty intervals of the appearance of constituent local state events.

The way a synchronizer operates is shown in Fig. 2.3. The SCGS detecting program is present in every synchronizer. The algorithm works permanently and on each received local state message automatically it tries to detect a next SCGS. For each detected SCGS, one or more control conditions (control predicates) are computed. Predicates are specified as blocks of code in C. If a predicate value is true, then a number of control signals are sent by the synchronizer to selected application processes (threads). Inside programs allocated to the same processor signals are sent as Unix signals, which work asynchronously in the way which resembles

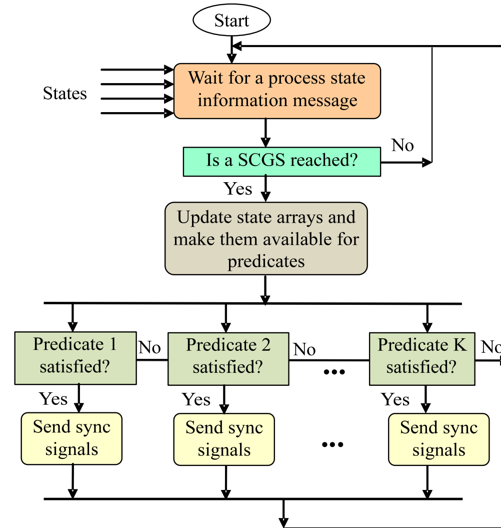


Fig. 2.3: Synchronizer control flow diagram.

interrupts. For signals sent to a remote processor, Unix signals are converted into messages sent by message passing with the use of the MPI communication library. When a message reaches a distant processor it is again converted — this time into a Unix signal, which is asynchronously received by the code of a target process or thread. Together with a signal some data can be transferred.

Two types of reaction to control signals in target processes or threads are possible. The first type of the reaction is a signal-driven activation (interrupt), which breaks current computation and activates a reaction code associated with the region. After completion of the reaction code the broken computing resumes. The second type of reaction to a signal is a signal-driven cancellation. It stops current computation and activates a cancellation handling procedure associated with the region. Program execution resumes just after the abandoned region. The described mechanism of distant Unix signals resembles the paradigm of distributed interrupts.

Since the messages representing Unix signals have to be transmitted over an external message passing network. The transfer time can be here very long. An important problem is the control of the validity of reactions for the signals which travel with a long delay. Sometimes the signaled program, which is not suspended but continues execution after sending the local state message, is advancing execution too far in respect to the acceptable time distance between the local signal and the reaction cased by it. In the code of a process (thread), regions sensitive to incoming control signals can be marked by special delimiters. If the process (thread) control is inside a region sensitive to a signal and the signal arrives, then a reaction is triggered. Otherwise, the reaction is neglected.

2.2. Synchronous model of program execution control based on global states. The second distributed program execution control mechanism in the PEGASUS environment involving the global state monitoring concerns defining the flow of control in distributed programs based on the global application state monitoring. PEGASUS framework provides global parallel control structures which are on PARALLEL DO (PAR) and JOIN constructs, embedded (if needed) into standard control statements of high level languages (IF, WHILE...DO, DO...UNTIL, CASE) but governed by predicates on application global states, for some of them see Fig. 2.4. Such global control constructs are graphically supported in PEGASUS. The programmer design its flow of control in the distributed program by drawing a flow of control graph using specially prepared GUI.

The predicate *GP3* in the synchronizer *S* assigned to the *N0* logical processor is evaluated based on a global state generated from local state messages received from the program blocks *P1, ..., Pn*, assigned to logical processors *N1,...,Nn* (for the simplicity of the graph representation we do not draw the respective state message transfer edges). Logical processors are further assigned to physical processors by the program mapping facility provided in the PEGASUS run-time. Based on the value of *GP3* a binary control signal is sent to the switch *SW*, which governs the flow of execution control in the PARALLEL DO–UNTIL construct. Usually the program blocks which participate in global control constructs are additionally asynchronously controlled based

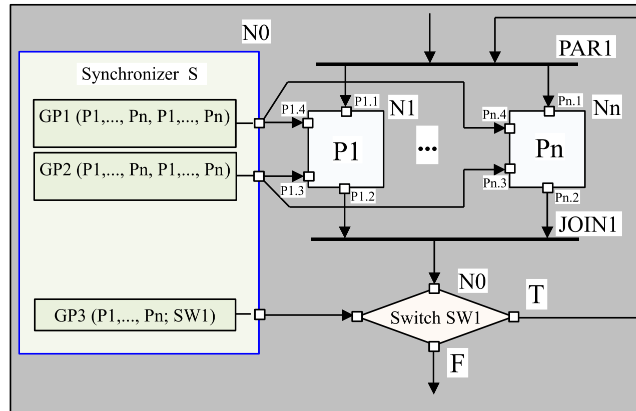


Fig. 2.4: PARALLEL DO-UNTIL construct with an asynchronous control predicates GP1, GP2 and a control flow predicate GP2.

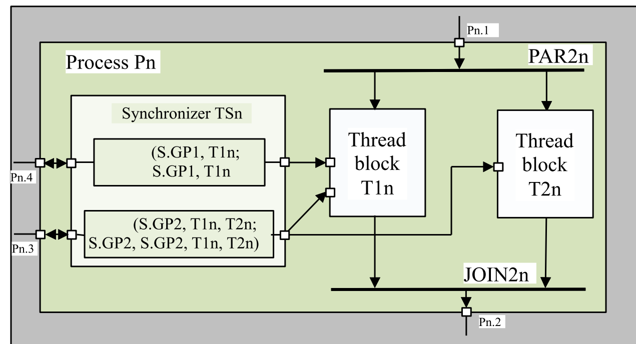


Fig. 2.5: Structure of a terminal program block

on global application states. It is done by the predicates *GP1* and *GP2*. They receive local state messages from *P1, ..., Pn*, program blocks evaluate a predicate condition and send control signals to *P1, ..., Pn*.

The control construct shown in Fig. 2.4 is a replicated control construct which means that the program blocks *P1, ..., Pn* have the same internal configuration and the same program code. The program blocks *P1, ..., Pn* can contain nested other high level control constructs on program blocks. The nesting can be repeated by special clicking on the block with the highest index. If the nesting is not to be introduced since the block does not have any high level conditional control construct nested inside, another special click on the block opens a terminal block design window. The internal structure of a program terminal block is shown in Fig. 2.5. Since the PARALLEL-DO-UNTIL construct from Fig. 2.4 was replicated, Fig. 2.5 represents the control flow of the block *Pn* with the highest index in the replication. The terminal block structure is designed of a number of thread blocks (*T1n, T2n*) placed under a PAR construct and a thread level synchronizer (*TSn*). The thread blocks composed of a number of threads with the same code but working on separate data. The thread level synchronizer implements an asynchronous control model in respect to the thread blocks *T1n, T2n*. Threads in the thread blocks can send local state messages to *TSn*. *TSn* reconstructs the strongly consistent global states of threads belonging to the process *Pn*. Upon reaching an SCGS, *TSn* activates control predicates evaluation on the global states. As a result *TSn* can send control signals to threads in *T1n, T2n* thread blocks to modify their behaviour asychronously. The exit from the thread blocks is synchronized by the JOIN2 block which works as a barrier for all threads in the blocks *T1n, T2n*. *TSn* can also send some discovered global states together with some data to the higher level synchronizer *S* beyond the *Pn* process. The synchronizer *S* can collect such partial states of the synchronizers *TSi* to construct some more global program states to evaluate some global predicate on the reconstructed more global states. Depending on the values of the predicates the synchronizer *S*

can send control signals back to the thread level synchronizers *TSi*. *TSi* can send control signals to the threads working inside their processes *Pi*.

Fig. 2.6 shows a graphical representation of the global PARALLEL IF control construct. In this construct the Switch *SW* control block supervises parallel execution (PAR construct) of a set of replicated program blocks *P1, ..., Pn*. The blocks are assigned to *P1, ..., Pn* parallel processors. The Switch is controlled by the control signal generated by the global predicate *GP1* embedded in the synchronizer. The predicate is evaluated based on the global state composed of local states provided by program blocks *B1, ..., Bn* existing somewhere in the program. For the simplicity of the graph representation we do not draw the respective state message transfer edges. Program blocks *P1, ..., Pn* are additionally asynchronously controlled by the predicates *GP2* and *GP3*. The predicates are evaluated based on the local states provided by the blocks *P1, ..., Pn*. The exit from the PAR construct is synchronized by the JOIN lock working as a barrier. More details on the PEGASUS framework can be found in [11] and [18].

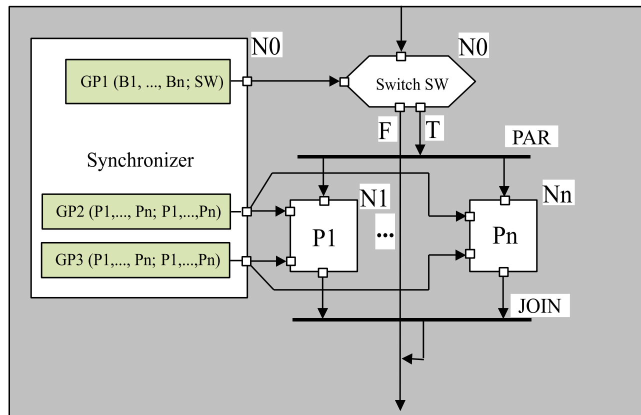


Fig. 2.6: PARALLEL IF construct with a control flow predicate GP1 and an asynchronous control predicate GP2, GP3.

3. Load balancing algorithm. The efficient execution of parallel programs requires balancing of computational loads among available nodes of parallel system. The problem of load balancing is NP–hard [19], thus it is necessary to apply different kinds of heuristics from various fields, such as prefix sum, recursive bisection, space filling curves, work stealing and graph partitioning.

In the static case of the load balancing problem, i.e. when the computational tasks co-exist for the entire duration of the parallel program and their workload is known, it is possible to solve it using the graph partitioning algorithm. That's why METIS [20], a well known graph partitioning framework, is widely used for mapping and static load balancing of parallel applications.

When we encounter changes of a workload and the varying availability of resources, the problem of load balancing becomes a much more hard to solve. In such a case, the only feasible approach is to apply dynamic, on–line load balancing. Also for irregular problems, i.e. when the workload depends on the processed data, the only solution is the dynamic approach.

There exist several dynamic load balancing strategies [21]. The simplest ones are based on the greedy heuristics, where the largest workloads are moved to the least loaded processors until the load of all processors is close to the average load. The more sophisticated algorithms use some refinement strategies, where the number of objects migrated is reduced or the communication between different objects is also considered.

Depending on the execution model of parallel application, dynamic load balancing is implemented by a migration of the application components (processes or threads) or by a data redistribution among computing nodes. In both cases, the goal is to achieve such a distribution of workloads that guarantees the highest possible efficiency of the overall application execution. In the strategy presented in the paper we focus at data migration as a basic load balancing mechanism.

3.1. Load balancing based on global states. In the approach presented in the paper, we use the global state monitoring infrastructure provided by the PEGASUS environment, as a tool to achieve dynamic

load balancing of parallel applications. The motivation for this approach is as follows: global state monitoring provides very efficient, low-latency way to detect and distribute load balancing informations, while asynchronous global control enables for instant response and load balancing accomplishment.

In the proposed implementation of the load balancing control, the complete load balancing algorithm is placed in sychronizers which are fully programmable. This way we are able to use many existing load balancing methods like those based on graph partitioning (METIS [20], Zoltan [22], etc.). However, it would require an extensive load redistribution by many time-consuming distant thread-to-thread load transfers, to follow the global optimal work partition. Thus, we propose a new strategy to avoid such flaws and to allow for real load migration only if unavoidable by other methods.

The overall scheme of load balancing, presented here, is an extended version of our former algorithm for Java-based distributed applications, see [12] for more details. Besides improvements inside balancing heuristics, the main difference, compared to the algorithm from [12], consits in the adoption of general parallel application model, applicable for wider range of computational problems.

The load balancing approach, proposed in the paper, consists of two main steps: **detection** of imbalance and its **correction**. The first step uses some measurement infrastructure to detect the functional state of the computing system and executed application. In the second step, we migrate some load from overloaded computing nodes to underloaded computing nodes to balance the workloads.

The goal of the algorithms is to balance the load of computing nodes of the parallel system. As it will be explained later, during detection of imbalance and its correction, the algorithm takes into account both computational demands of the application and its inter-process communication pattern.

3.2. System and application observations. The on–line detection of the state of the system is necessary since the computing nodes (workstations) can be heterogeneous, moreover they can have different and variable computing capabilities over time. A load imbalance occurs when the differences of loads between the computing nodes become too big.

Applications' observation is necessary since they can be irregular in general, thus their workloads can change in an unpredictable way. The application observation mechanism provides knowledge of the application behavior during its execution. This knowledge is necessary to undertake adequate and optimal load balancing decisions.

There are two types of measurements in the proposed load balancing method for the PEGASUS environment:

System level observations – they provide general functional indicators, e.g. CPU load, which are universal for all kinds of applications. The parameters are measured using software sensors, for that the load balancing mechanism of the PEGASUS environment installs observation kernels on computing nodes.

Application specific observations – they include measurements which have to be done in each application since they furnish data on application–dependent behavior. An example of this kind of data is the workload of a process (or thread). It must be bound to the application since such data can depend for example on the amount of data to be processed in the future which is known only to the application logic.

The PEGASUS global states monitoring infrastructure is a convenient tool to organize both the system and the application level observations. In our implementation, application program processes and system observation agents send messages on local state changes to *load balancing synchronizer*, where they are processed and appropriate reactions are computed using the method described in next sections. Similarly, reactions are organized as asynchronous program execution control. Load balancing logic is implemented as control predicates inside *a load balancing synchronizer.* Figure Alg. 3.1 presents a general scheme of the proposed algorithm. The rest of this section describes functions and symbols used in the flowchart in Alg. 3.1.

3.3. Detection of load imbalance. Computing nodes composing the parallel system can be heterogeneous, therefore to detect the load imbalance, we need to known both the processors availability and their computing power. The heterogeneity of nodes disallows us to directly compare measurements taken on computing nodes whose computing powers are different. Thus, to compare the computing nodes' load, we propose to use the availability index of a CPU computing power on the node n:

$$\mathrm{Ind}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{power}}(n) * \mathrm{Time}^{\%}_{\mathrm{CPU}}(n)$$

where:

$\mathrm{Ind}_{\mathrm{power}}(n)$ — computing power of a node n, which is the sum of computing powers of all cores on the node,

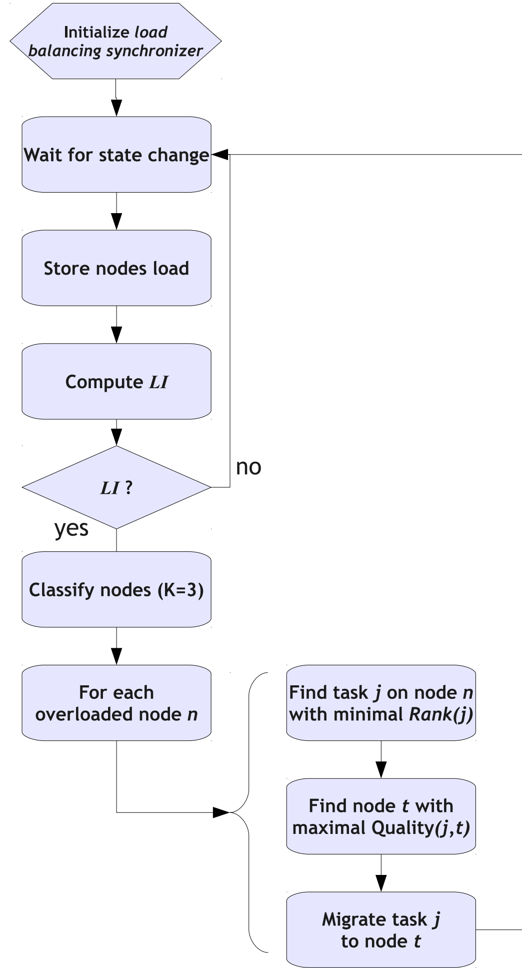


Fig. 3.1: General scheme of load balancing algorithm

$\text{Time}^{\%}_{\text{CPU}}(n)$ — the percentage of the CPU power available for computing threads on the node n, periodically estimated by observation agents on computing nodes.

The way the availability index of a CPU computing power is computed is not obvious, so some explanation follows. The computing power of the node is determined by the *calibration process* [23]. For each node, the calibration should be performed in a consistent way to enable comparisons of calibration results (they can be expressed in MIPS, MFLOPS or similar). The calibration needs to be done only once when the nodes join the system. The percentage of the CPU power available for a single computing thread is computed as a quotient of the time during which the CPU was allocated to the probe thread against the time span of the measurement (see [23] for more details and the description of the implementation technique). $\text{Time}^{\%}_{\text{CPU}}(n)$ value is the sum of the percentage of CPU power available for the number of probe threads equal to the number of CPU cores on the node.

A load imbalance LI is defined as the difference of the availability indexes between the most heavily and the least heavily loaded computing nodes composing the cluster, which can be determined as:

$$LI = \max_{n \in N}(\text{Ind}_{\text{avail}}(n)) \leq \alpha * \min_{n \in N}(\text{Ind}_{\text{avail}}(n))$$

where:
N — the set of all computing nodes, α — a positive constant number.

Power indications $\text{Ind}_{\text{power}}$ and CPU time use rate Time_{CPU} are collected and sent to *load balance synchronizer* by local system agents as state messages.

The proper value of the α coefficient can be determined using both statistical and experimental approaches. Following our previous research [12] on load balancing algorithms for Java-based distributed computing environment, we can restrict the value to the interval $[1.5\ldots 2.5]$. These experimental values give good results because they are neither too restrictive nor too tolerant for the load imbalance.

3.4. Correction of load imbalance. At the beginning of this step, we know the state of computing nodes (i.e. their availability indexes), which is an outcome of the detection of load imbalance. Now, we detect overloaded computing nodes and then we'll re-balance them (i.e. transform into the normally loaded by the migration of the load).

3.4.1. Classification of computing nodes. We classify n computing nodes into three categories, based on the computed availability indexes: overloaded, normally loaded and underloaded. To build categories, we use the K-Means algorithm [24] with $K = 3$. The three centers that we choose are the minimum, average and maximum availability indexes, where the average index is simply the average of indexes measured during the last series of measures over the whole cluster.

3.4.2. Selection of candidates for migration. To correct load imbalance, we migrate the load from overloaded computing nodes to underloaded ones. The load for migration has to be selected carefully, so that the migration does not cause the deterioration of overall computing conditions.

The loads are represented by the data processing activities of the threads which are running on computing nodes. Two parameters are used to characterize the load that we want to migrate:
a) the *attraction* of a load to a computing node,
b) the *weight* of the load.

The attraction of a load to a computing node is expressed in terms of communication, i.e. it indicates how much a particular thread communicates with others allocated to the same node. A strong attraction means frequent communication, so, the less the load is attracted by the current computing node, the more interesting it is to be selected as a migration candidate. The computational weight of the load gives the quantity of load which could be removed from the current node and placed on another.

Both the *attraction* and *weight* are application-specific metrics, which should be provided by an application programmer in the form of state messages sent to *load balance synchronizer*:
1. $\mathrm{COM}(t_s, t_d)$ is the communication metrics between threads t_s and t_d,
2. $\mathrm{WP}(t)$ is the load weight metrics of a thread t.

In our load balancing algorithm we prefer to move an entity whose quantity of work is neither too big, nor too small. Thus, the smaller the distance is to the average thread loads, the more the load is interesting for migration.

The attraction of the load j to the actual computing node:

$$\mathrm{attr}(j) = \sum_{o \in L^*(j)} (\mathrm{COM}(j,o) + \mathrm{COM}(o,j))$$

where:
$L^*(j)$ — the set of threads, placed on the same node as a thread j (excluding j).

The load deviation compared to the average quantity of work of the node j:

$$\mathrm{ldev}(j) = |\mathrm{WP}(j) - m_{WP}| \tag{3.1}$$

where:
$m_{WP} = \frac{\sum_{o \in L(j)} \mathrm{WP}(o)}{|L(j)|}$,
$L(j)$ — the set of threads, placed on the same node as the thread j (including j).

The formula above allow to compute the attraction of an entity to the local node in order to compare it with the attractions of other loads of this node. The comparison formula are:
(1) global attraction measure:

$$\mathrm{attr}^{\%}(j) = \frac{\mathrm{attr}(j)}{\max_{o \in L(j)}(\mathrm{attr}(o))}$$

(2) load deviation compared to the average quantity of work:

$$\mathrm{ldev}^{\%}(j) = \frac{\mathrm{ldev}(j)}{\max_{o \in L(j)}(\mathrm{ldev}(o))}$$

The element to migrate is the one for which a weighted sum of last two parameters has the minimal value:

$$\mathrm{Rank}(j) = \beta * \mathrm{attr}^{\%}(j) + (1 - \beta) * \mathrm{ldev}^{\%}(j) \tag{3.2}$$

where:
β — a real between 0 and 1. Its choice remains experimental. Let us notice however that the bigger β is, the bigger is the weight of the object attraction.

3.4.3. Selection of the target computing node for migration. The target computing node for migration should be both underloaded and strongly attracted by migrated load. In order to select the target, we use the formula which is based on weighted sum of two selection criteria.

The first criterion to qualify a computing node as a migration target is the computing node power availability indexes. We prefer the one whose availability index is the highest, because it is actually the least loaded. We also take into account the number of *waiting* threads in the potential targets ($T_{\mathrm{wait}}(n)$ – the set of waiting threads on a node n). We consider them, however, only as potential load, which must be taken under consideration together with the related load currently processed on the computing node.

The second criterion is based on the attraction of a selected load entity to this node. A relationship of attraction of the load j to node n is defined as follows:

$$\mathrm{attrext}(j,n) = \sum_{e \in T(n)} (\mathrm{COM}(e,j) + \mathrm{COM}(j,e))$$

where:
$T(n)$ — the set of threads, placed on a node n.

The formula to select the target for migration is as follows (we normalize all the values related in the interval $[0 \ldots 1]$):

$$\mathrm{Quality}(j,n) = \gamma * \mathrm{attrext}^{\%}(j,n) + (1 - \gamma) * \mathrm{Ind}^{\%}_{\mathrm{avail}}(n) \tag{3.3}$$

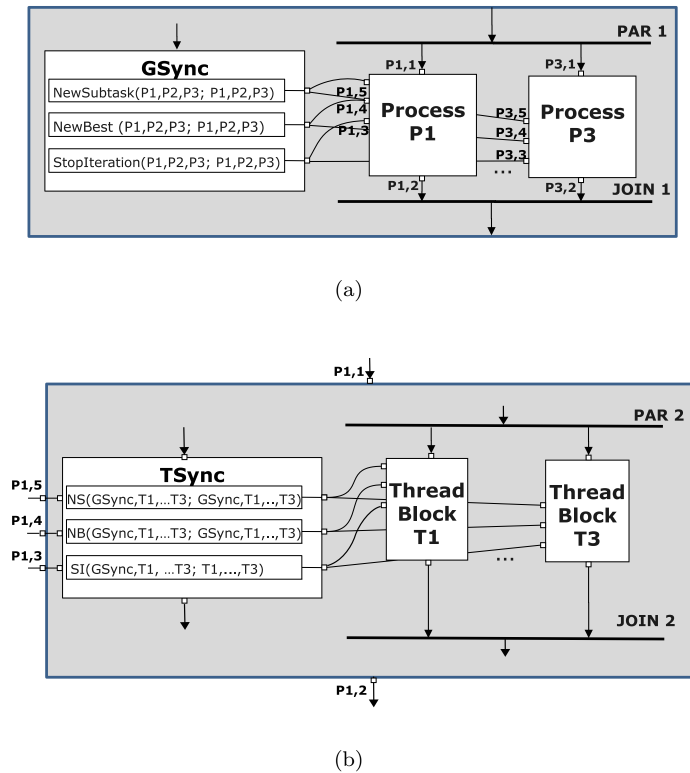
with $\gamma \in [0 \ldots 1]$ and

$$\mathrm{attrext}^{\%}(j,n) = \frac{\mathrm{attrext}(j,n)}{\max_{e \in N}(\mathrm{attrext}(j,e))}$$

$$\mathrm{Ind}^{\%}_{\mathrm{avail}}(n) = \frac{\mathrm{Ind}^{*}_{\mathrm{avail}}(n)}{\max_{e \in N}(\mathrm{Ind}^{*}_{\mathrm{avail}}(e))}$$

$$\mathrm{Ind}^{*}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{avail}}(n) - \mathrm{Ind}_{\mathrm{avail}}(n) * \frac{|T_{\mathrm{wait}}(n)|}{|T(n)|} \tag{3.4}$$

We evaluate the above equations for all potential computing node targets for a load which is a candidate for migration. As the new location for the load we choose the computing node which maximizes equation 3.3. The value of the coefficient γ has to be determined using experimental verification.



(a)

(b)

Fig. 3.2: (a) Control flow graph for the TSP program. (b) Control flow graph for worker process of TSP.

4. Example of load balancing based on global state monitoring.

4.1. General assumptions on the solution search. We will present now how the described load balancing method can be applied to a Traveling Salesman Problem (TSP) solved by a parallel Branch and Bound (B&B) method. The global control in the algorithm supports both organizing a reduced search of the TSP problem solution space due to the B&B method and load balancing of the search computations in distributed processes. A parallel solution space search is done following a "master-slave" method implemented using a number of worker search processes assigned to multicore processors. Each worker process is composed of a number of worker threads and a thread level synchronizer. The solution space is organized as a tree of salesman trajectories which are gradually being developed and searched by worker threads under control of synchronizers. The solution space is divided into search subproblems (subtasks) which correspond to subtrees of the total solution tree. To reduce the search in the solution space, the global state monitoring with a bounding function is organized in all executive processes and threads.

Synchronizers are organized in a hierarchical structure and play the role of masters in the solution search algorithm. A central global synchronizer at the highest hierarchy level divides the set of all search subproblems into subtask pools meant for execution by worker processes. The subtasks are directed for execution to worker threads by the thread level synchronizer placed in each worker process. The thread synchronizer evaluates an estimate of the search work to be done (the current process load), based on state messages sent to it by worker threads. Then, the thread synchronizer sends the estimate to a global load balancing synchronizer. Based on it, the global synchronizer computes the deviation of the load of worker processes. This deviation is used to work out the load balancing decisions for the processes (i.e. the rate and amount of subtask delivery).

The global synchronizer maintains also the value of the best known solution found so far in worker processes. The global synchronizer updates this value if a thread synchronizer in a worker process informs it about a still better solution it has found. The global synchronizer distributes the updated globally best *min_dist* solution to all thread synchronizers of all worker processes. The thread synchronizers distribute this value to all their

threads. The threads use this *min_dist* value to bound further development of every partial trajectory solution (containing a partial set of towns) they consider. They also compare to *min_dist* every full trajectory solution (containing the full set of towns) found in the search process and inform the thread synchronizer on each better new solution (the trajectory and its distance value). The thread synchronizers compare the new received *min_dist* values to those they know and send them to the global synchronizer and back to worker threads if it is smaller than the currently known *min_dist*.

4.2. General load balancing strategy and its global implementation. The control flow graph of the TSP by the B&B method in PEGASUS is shown in Fig. 3.2(a). We have a replicated PAR construct in this graph with 3 parallel worker processes P_i, controlled by the predicates of the global synchronizer *GSync*. Worker processes are composed of worker thread blocks and the thread synchronizers *TSynch*, Fig. 3.2(b). Each worker thread block contains replicated worker threads which are assigned to the same processor core. *TSyncs* periodically report their current loads to the global synchronizer *GSync*. Based on these reports *GSync* works out load balancing decisions which direct pools of search subtasks to *TSyncs* in the way to balance loads in worker processes. *TSyncs* also report to *GSync* the best solutions with the trajectory lengths smaller than the *min_dist* values known to their threads. *TSyncs* send to *GSync* asynchronous requests for new subtasks if the pool of the subtasks they have for worker threads is close to exhaustion.

There are 3 predicates evaluated in the *GSync* synchronizer. The *NewSubtask* predicate computes the global mean load in the system and compares it to the loads reported by *TSyncs*. Inside this predicate the $\text{ldev}(j)$ parameter (see equation 3.1 in the previous section) is computed for each process. Based on $\text{ldev}(j)$ load balancing decisions are taken. The decisions are different for different search stages for the entire TSP problem. In the non-terminal search stages in which not all search subtasks have yet been distributed by *GSync* to search worker processes (more exactly the *TSync* synchronizers), the decision determines the number of new search subtasks which are to be sent to the processes which are soon to be idle. In the terminal stage of the solution search when all subtasks have already been distributed, the decision determines the subtask migration from the most loaded worker process to the least loaded one. In this case, the *NewSubtask* predicate of *GSync* selects the process which is too heavily loaded and which should migrate part of its subtasks to another less loaded process. We avoid direct communication of subtasks between threads of different worker processes located on different processors, so the migration is executed at the level of the TSync synchronizers. The selection for migration decision is done using the equation 3.2 given in the previous section. However, in our TSP problem, communication between worker threads does not appear since worker threads are independent, so the migration decision is taken only based on the $\text{ldev}(j)$ parameter. The decision upon the target process to receive load migration is taken in general based on the $\text{Quality}(j, n)$ parameter (see equation 3.3 in the previous section). In our example, due to the independent tasks, this parameter is reduced to $\text{Ind}^*_{\text{avail}}(n)$ (equation 3.4). The selection of the target for migration is done by *GSync* based on the information on partial processor power available in particular computing nodes ($\text{Ind}_{\text{avail}}(n)$ parameter), supplied periodically by *TSyncs*.

The *NewBest* predicate in *GSync* maintains the best known solution *min_dist* based on the local best solutions found so far. This value is determined based on the *min_dist* state messages sent from the *TSync* synchronizers in worker processes. The globally best solution is distributed back to all *TSyncs* by sending to them the appropriate control signal with the new best solution sent in the MPI message. *TSyncs* distribute the signals to all the threads they cooperate with, together with the placement of the new best solution in the memory shared by *TSyncs* and their threads, available on the processor node. The *StopIteration* predicate in *GSync* takes care of the quality of the TSP solution found so far. The TSP solution search can be stopped if a sufficiently good solution has been found in worker processes even if not entire search tree has been scanned. In this case, the *StopIteration* predicate of *GSync* distributes the *StopIteration* signal to all *TSyncs*.

4.3. Load balancing implementation at the level of threads. The TSP program execution control inside a worker thread is shown in Fig. 4.1. When a thread is activated and all its programs are initiated the thread sends a "thread started" report to the *TSync* synchronizer. Then the tread waits for the first subtask pool to be received from TSync. When the new subtask pool arrives, the thread starts working on a subtask from it. A subtask consists of a vector of towns with a length smaller than the total number of towns which forms a trajectory with a known length. The thread gradually develops the trajectory by adding new towns and computing the length of each formed trajectory. The length of each new trajectory is compared to the current *min_dist* known to the thread. If the length of a new partial trajectory is bigger than or equal to *min_dist*, the trajectory is considered non perspective and it is not further developed (the B&B bounding). A new full

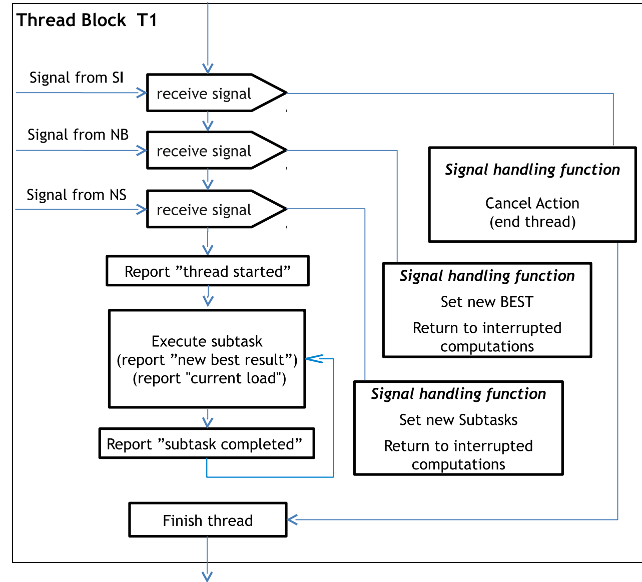


Fig. 4.1: TSP worker thread internal control.

trajectory with the length not smaller than *min_dist* is rejected. Only partial or full trajectories which have the lengths smaller than *min_dist* are maintained and registered. A full trajectory shorter than *min_dist* is reported to the *TSync* synchronizer as a "new best result". Its length is set as a new *min_dist* for a thread. Then the search is continued for the subtask by backtracking in its search tree to the highest partial trajectory which can be further developed. If the entire solution tree of the current subtask for a thread has been inspected, the thread sends a state message "subtask completed" to *TSync* and starts executing a next subtask from its pool. When a new subtask arrives to a thread, the number of search steps to be done $S = (N - t)!$ is computed as the current load of the subtask for the thread, where N is the total number of towns and t is the number of towns in the trajectory given in the subtask specification. During a subtask execution S is decreased by 1 for each step which creates a partial perspective trajectory or a full trajectory. S is decreased by $p!$ after each step which creates a partial non perspective trajectory where p is the difference between N and the number of towns in the created trajectory. The sums of S values for all subtasks present in a thread are periodically sent by threads to *TSyncs* as "current load" reports. On detection of each load SCGS a *TSync* computes the current load *PS* of its worker process as a sum of load values concurrently sent by all threads it controls. *TSyncs* periodically send *PS* to *GSync* which, on detection of respective SCGS computes the mean load of the system and the load deviations $\mathrm{ldev}(j)$ for particular processes. Based on the load deviations, load balancing decisions are taken followed by control signals sent to *TSyncs*.

In Fig. 4.1 we can see three control signals which can come to the thread from the *TSync* synchronizer. The first signal (Stop Iteration) is generated by *SI* predicate of *TSync*. It corresponds to the situation when the search activity in a thread is to be canceled. Next signal (New Best) is sent by the *NB* predicate of *TSync*. It informs the thread that the new best solution has been found and makes it available to the thread code. The last signal (New Subtask) comes from the *NS* predicate of the *TSync*. It brings new subtasks to be executed by the thread. The arrival of each of these signals breaks the basic search loop of the thread and activates a special signal handling function. For signal from *NS* and *NB* the handling function sets a new subtask or the new best solution in the thread. Then, the execution control returns to the interrupted activity. In the case of the signal from *NS* which comes while the thread is idle, the control is passed to the execution of a new subtask. The signals from *SI* cancel the interrupted basic search activity and the execution of the thread is finished.

5. Conclusions. Monitoring of global application states creates an efficient and flexible basis for distributed program execution control including the load balancing support. Global control constructs for the control flow design and the design of asynchronous control based on global application states monitoring with control signal dispatching provide flexible infrastructure for distributed applications implementation and system-

level optimizations. This was the motivation for the research on a new distributed program design framework PEGASUS which has been used in this paper as a basic program design and execution infrastructure. In the PEGASUS framework the semantics of control constructs at the process and thread levels takes into account global application states involved in synchronous and asynchronous program execution control. The proposed methods include new program execution paradigms and the corresponding software architectural solutions. The infrastructure enables an easy and convenient design of the load balancing logic in applications.

Dynamic load balancing based on distributed application global states monitoring in the PEGASUS environment has been illustrated by an example of a distributed program for solving the Traveling Salesman Problem by a Branch and Bound method. For this problem, we have designed a load balancing method in which the troublesome load migration between distributed threads mapped to different processors (by transfers of tasks between distributed threads) can be avoided in the initial program execution stages and replaced by controlling the initial computational tasks distribution by synchronizers. Only when all task pools have been distributed, the real task migration is used when it is unavoidable at this stage.

The described PEGASUS framework is in final implementation stage based on a cluster of contemporary multicore processors. The processors in the cluster are interconnected by three dedicated different networks used for control and computational data communication. Interprocess control communication including Unix signal propagation between processors is organized by the use of message passing over Infiniband network. Computational data communication and data exchange for processor clock synchronizatin purposes is performed by dedicated Ethernet networks. C/C++ language with the MPI2, OpenMP and Pthreads libraries are used for writing application programs and the framework control code.

This paper has been partially sponsored by the MNiSW research grant No. NN 516 367 536.

REFERENCES

[1] Baude et al., Programming, Composing, Deploying for the Grid, In GRID COMPUTING: Software Environments and Tools, J. C. Cunha, O. F. Rana (Eds), Springer, January 2006.

[2] L.V. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, Proc. of OOPSLA'93, ACM Press, Sept. 1993, pp. 91–108.

[3] `http://www.scalemp.com/architecture`

[4] Y. Etsion, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, M. Valero, Task Superscalar: Using Processors as Functional Units, Usenix Workshop on Hot Topics In Parallelism (HotPar), January 2010.

[5] C. Villavieja, Y. Etsion, A. Ramirez, N. Navarro, FELI: HW/SW Support for On-Chip Distributed Shared Memory in Multicores, Euro-Par 2011, LNCS 6852, Part I, pp. 280-292, Springer 2011.

[6] N. Carriero, D. Gelernter, *Linda in Context*, in Communications of the ACM 32 (4), April 1989, pp. 444–459

[7] `http://www.cs.ucy.ac.cy/courses/EPL441/manifold/tut.pdf`, `http://reo.project.cwi.nl/`

[8] K. Marzullo, D, Wood, *Tools for constructing distributed reactive systems*, Technical Report 14853, Cornell University, Department of Computer Science, Feb. 1991.

[9] M. Tudruj, *Fine–Grained Global Control Constructs for Parallel Programming Environments*, in Parallel Programming and Java: WoTUG–20, IOS, 1997, pp. 229–243.

[10] M. Tudruj, J. Borkowski, D. Kopański, *Graphical Design of Parallel Programs with Control Based on Global Application States Using an Extended P–GRADE System*, in Distributed and Parallel Systems, Cluster and GRID Comp., Kluver, Vol. 777, 2004.

[11] M. Tudruj, J. Borkowski, Ł. Maśko, A. Smyk, D. Kopański, E. Laskowski, Program Design Environment for Multicore Processor Systems with Program Execution Controlled by Global States Monitoring. 10th International Symposium on Parallel and Distributed Computing, Cluj-Napoca, July, 2011, ISPDC 2011, IEEE CS, pp. 102-109.

[12] R. Olejnik, I. Alshabani, B. Toursel, E. Laskowski, M. Tudruj, Load Balancing Metrics for the SOAJA Framework, Scalable Computing: Practice and Experience, 2009, Vol. 10, No. 4.

[13] K. Barker, N. Chrisochoides, An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications, Supercomputing 2003, Phoenix, ACM, 2003.

[14] J. Borkowski, M. Tudruj, D. Kopański, Global predicate monitoring applied for control of parallel irregular computations, 15th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP'07, Naples, Italy, IEEE CS, 2007, pp. 105-111.

[15] J. Borkowski, M. Tudruj, Tuning the Efficiency of Parallel Adaptive Integration with Synchronizers, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 2008, pp. 351-357.

[16] O. Babaoglu, K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms, Distributed Systems, Addison-Wesley, 1995.

[17] S. D. Stoller, Detecting Global Predicates in Distributed Systems with Clocks, Distributed Computing, Vol. 13, N. 2, 2000, pp. 85-98.

[18] J. Borkowski, M. Tudruj, Dynamic Distributed Programs Control Based on Global Program States Monitoring, Scalable Computing: Practice and Experience, Journal, Vol. 13, N. 2, June 2012, pp. 173–186

[19] H. El-Rewini, T. G. Lewis, H. H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice Hall 1994.

[20] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, Proc. 24th Intern. Conf. Par. Proc., III. CRC Press, 1995, pp. 113–122.

[21] A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, F. Pellegrini, Applying graph partitioning methods in measurement-based dynamic load balancing, PPL Technical Report 2012, University of Illinois, Dept. of Computer Science, URL: papers/201107_ScotchCharm.

[22] K.D. Devine, E.G. Boman, L.A. Riesen, U.V. Catalyurek, C. Chevalier, Getting Started with Zoltan: A Short Tutorial, Sandia National Labs Tech Report SAND2009-0578C, 2009.

[23] G. Paroux, B. Toursel, R. Olejnik, V. Felea, A Java CPU calibration tool for load balancing in distributed applications, ISPDC/HeteroPar'04, IEEE CS 2004, pp. 155–159.

[24] J.A. Hartigan, M.A. Wong, A K-Means clustering algorithm, Applied statistics, Vol. 28, pp. 100-108, 1979.

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

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:
- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:
- programming environments,
- debugging tools,
- software libraries.

Performance:
- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:
- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:
- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in $\LaTeX 2_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.