Toward an autonomic engine for scientific workflows and elastic Cloud infrastructure

Hadrien Croubois

Numéro National de Thèse : 2018LYSEN061

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

operée par **l'École Normale Supérieure de Lyon**

École Doctorale N°512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Présentée et soutenue publiquement le 16/10/2018, par :

Hadrien CROUBOIS

Toward an autonomic engine for scientific workflows and elastic Cloud infrastructure

Étude et conception d'un système de gestion de workflow autonomique

Devant un jury composé de :

Noël De Palma	Professeur des universités, Université Joseph Fourier	Rapporteur
Johan Montagnat	Directeur de recherche, CNRS	Rapporteur
Luciana Bezerra Arantes	Maître de conférences, Université Sorbonne	Examinatrice
Pushpinder Kaur Chouhan	Chercheure, Ulster University Jordansstown	Examinatrice
Frédéric Desprez	Directeur de recherche, INRIA	Examinateur
Eddy Caron	Maître de conférences, ENS de Lyon	Directeur de thèse

Toward an autonomic engine for scientific workflows and elastic Cloud infrastructure

Hadrien Croubois
Supervisor: Eddy Caron

Abstract

The constant development of scientific and industrial computation infrastructures requires the concurrent development of scheduling and deployment mechanisms to manage such infrastructures. Throughout the last decade, the emergence of the Cloud paradigm raised many hopes, but achieving full platform autonomicity is still an ongoing challenge.
Work undertaken during this PhD aimed at building a workflow engine that integrated the logic needed to manage workflow execution and Cloud deployment on its own. More precisely, we focus on Cloud solutions with a dedicated Data as a Service (DaaS) data management component. Our objective was to automate the execution of workflows submitted by many users on elastic Cloud resources.
This contribution proposes a modular middleware infrastructure and details the implementation of the underlying modules:

- A workflow clustering algorithm that optimises data locality in the context of DaaS-centered communications;
- A dynamic scheduler that executes clustered workflows on Cloud resources;
- A deployment manager that handles the allocation and deallocation of Cloud resources according to the workload characteristics and users' requirements.

All these modules have been implemented in a simulator to analyse their behaviour and measure their effectiveness when running both synthetic and real scientific workflows. We also implemented these modules in the Diet middleware to give it new features and prove the versatility of this approach. Simulation running the WASABI workflow (waves analysis based inference, a framework for the reconstruction of gene regulatory networks) showed that our approach can decrease the deployment cost by up to 44% while meeting the required deadlines.

Keywords: Middleware, Workflow, Cloud, DaaS, Autonomic, Scheduling, Provisioning, Distributed systems.

Étude et conception d'un système de gestion de workflow autonomique

Hadrien Croubois
Directeur de thèse: Eddy Caron

Résumé

Les infrastructures de calcul scientifique sont en constante évolution, et l'émergence de nouvelles technologies nécessite l'évolution des mécanismes d'ordonnancement qui leur sont associé. Durant la dernière décennie, l'apparition du modèle Cloud a suscité de nombreux espoirs, mais l'idée d'un déploiement et d'une gestion entièrement automatique des plates-formes de calcul est jusque la resté un vœu pieu.
Les travaux entrepris dans le cadre de ce doctorat visent a concevoir un moteur de gestion de workflow qui intègre les logiques d'ordonnancement ainsi que le déploiement automatique d'une infrastructure Cloud. Plus particulièrement, nous nous intéressons aux plates-formes Clouds disposant de système de gestion de données de type DaaS (Data as a Service). L'objectif est d'automatiser l'exécution de workflows arbitrairement complexe, soumis de manière indépendante par de nombreux utilisateurs, sur une plate-forme Cloud entièrement élastique.
Ces travaux proposent une infrastructure globale, et décrivent en détail les différents composants nécessaires à la réalisation de cette infrastructure :

- Un mécanisme de clustering des tâches qui prend en compte les spécificités des communications via un DaaS ;
- Un moteur décentralisé permettant l'exécution des workflows découpés en clusters de tâches ;
- Un système permettant l'analyse des besoins et le déploiement automatique.

Ces différents composants ont fait l'objet d'un simulateur qui a permis de tester leur comportement sur des workflows synthétiques ainsi que sur des workflows scientifiques réels issues du LBMC (Laboratoire de Biologie et Modélisation de la Cellule). Ils ont ensuite été implémentés dans l'intergiciel DIET.
Les travaux théoriques décrivant la conception des composants, et les résultats de simulations qui les valident, ont été publié dans des workshops et conférences de portée internationale.

Mots-clés : Intergiciel, Workflow, Cloud, DaaS, Autonomic, Ordonnancement, Allocation, Systèmes Distribués.

Table of contents

List of figures

List of tables

List of algorithms

Chapter I

Introduction

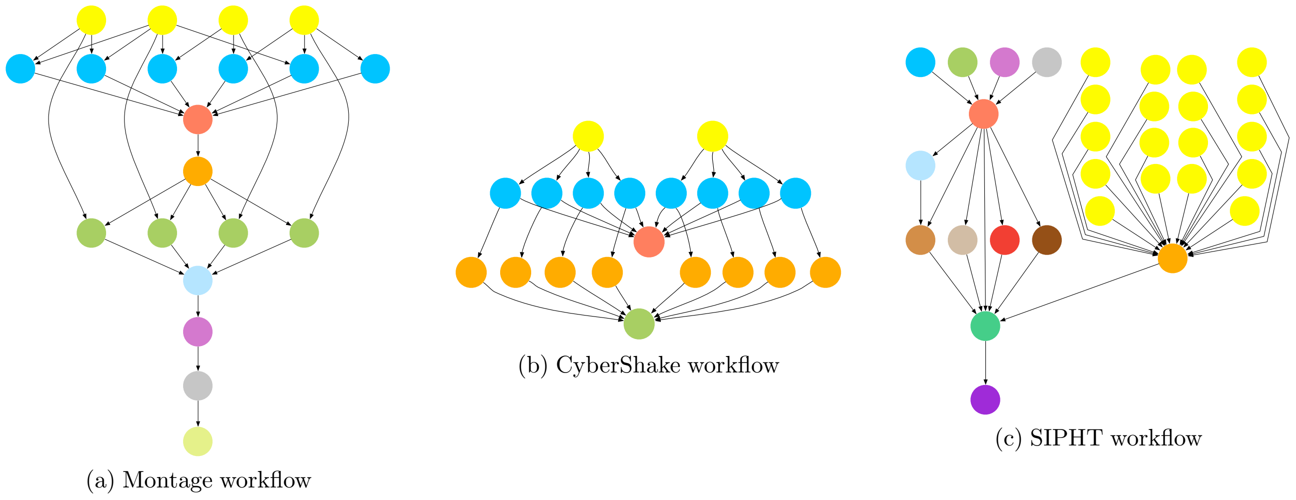
Section I.1

Workflows: A tool for parallel computing

Workflows are used to describe a series of computations in a structured manner. This vision helps with the parallel execution of large applications. Operating systems can already run multiple processes at the same time, share the available resources (CPU time, but also memory and IO), and handle sleeping processes. If we stop seeing the application as an atomic object but rather as a succession of operations that exchange pieces of data, then we can use this representation to achieve parallelism. Modern CPU can perform this kind of operation at the instruction level (exploring branches and reordering independent operations[1]). However, having a proper subdivision of the job into tasks can help achieve good parallelism at the multi-core and multi-node levels.

In the world of large-scale computation, which includes scientific and industrial applications, workflows are compelling tools. Not only do they help with the description of large applications by subdividing them into simpler elements (that can be reused), but it also enables application designer to benefit from the work computer scientists have put in the design of workflow engines and schedulers.

Although there are other models [Pautasso and Alonso, 2006], scientific workflows are most commonly modelled as Directed Acyclic Graphs (DAGs). This model can be represented using various syntaxes and is supported by many workflow management systems [Deelman et al., 2005, Fahringer et al., 2005, Couvares et al., 2007]. Examples of scientific workflows are given in [Bharathi et al., 2008] (see Figure 1).


(a) Montage workflow

(b) CyberShake workflow

(c) SIPHT workflow

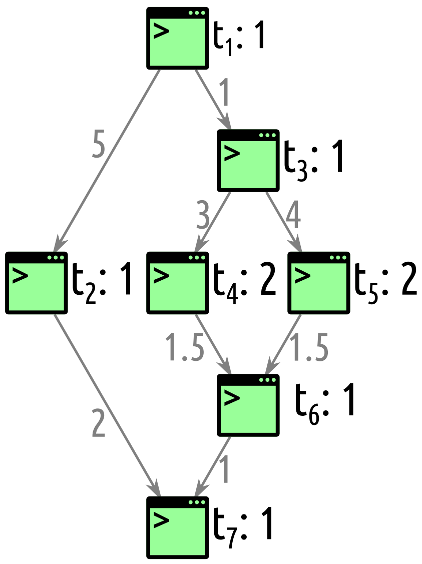
Figure 1: Examples of scientific workflows from [Bharathi et al., 2008]

Scheduling is key to leveraging the benefits offered by workflows and achieving an efficient and autonomous execution in distributed environments. Figure 2 illustrates the importance of scheduling and how it can affect the effective execution of workflows.

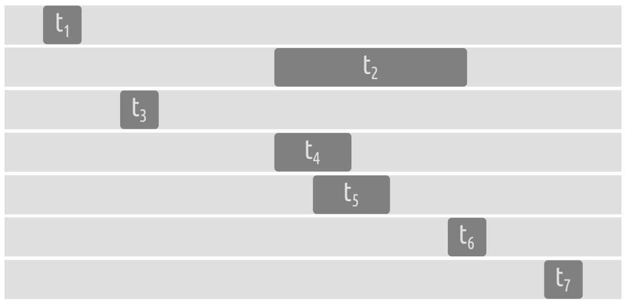
Workflows and the relative workflow engines can be found at many levels. For example, the OpenMP task model [Ayguadé et al., 2009] provides mechanisms for developers to describe workflows inside the application code using pragmas in their C code. The OpenMP runtime will later play the role of the workflow engine, scheduling tasks and discussing with the operating system to achieve an efficient execution. This parallelisation is transparent to the user. With the NUMA[2] model, this vision can be

[1]This can sometimes lead to critical vulnerabilities that affect the whole system.

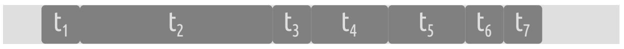
[2]Non-uniform memory access



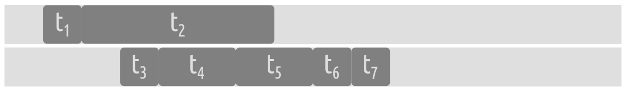
(a) Workflow represented as a DAG



(b) Bad scheduling: one-to-one (makespan: 14)



(c) Bad scheduling: all-to-one (makespan: 13)



(d) Good scheduling (makespan: 9)

Figure 2: Example of workflow and possible schedules. Figure 2a is a workflow with communications. Durations of the tasks and inter-node communications are shown next to the nodes/edges of the graph. Figure 2b, 2c and 2d are possible schedules. Background lines (light-grey) represent nodes, and dark-grey blocks are tasks running on these nodes. Figure 2b shows that the cost of communication makes a fully parallel execution inefficient. On the other hand, Figure 2d shows that a good schedule can reduce the execution time (makespan) of the workflow. However, computing such as schedule is not an easy task as we discuss in Chapter II.

scaled up to multiple CPUs, at which point memory isolation becomes a serious concern.

At a higher level, we find engines that perform workflow scheduling on clusters of nodes. Each one has its own, independent, operating system and process scheduler. Such a workflow engine is an orchestrator that can either be centralised (one node is dedicated to this role) or distributed between the different nodes. Using services (daemons) running on the different nodes the engine can use the resources provided by the nodes operating systems. As such, a distributed workflow engine can be seen as a minimalistic distributed operating system. With each level of scheduling comes a different representation of the workflows that focuses on the critical aspects of the targeted platform.

Scheduling of workflow should always be guided by a set of constraints that formulate a required quality of service (QoS). Workflow engines have to use the available resources effectively if they hope to fulfil these requirements. An appropriate model of these resources is therefore essential to meet the challenges that derive from the features of the targeted platform.

Section I.2

Cloud: On-demand computing resources

Thanks to virtualisation technologies it is possible to access and seamlessly reconfigure computing resources, including hardware specific features. This concept dates back to the 60's but we had to wait for the 21's century to have a technology mature enough for the development of Cloud solutions.

Cloud computing is the commercial evolution of grid computing. Using virtualisation to configure computing nodes, the Cloud paradigm provides on-demand access to virtual resources ranging from a single node to entire clusters. While the computing power of the nodes does not significantly suffer from virtualisation, having these nodes spread out geographically means that the network performances are far from what is achieved in supercomputers. Therefore, supercomputers and distributed Cloud infrastructures target different classes of applications.

The Cloud ecosystem inherits from previous work relative to grid infrastructure. Projects like SETI@home[3] paved the way for the execution of extensive computation on off-site resources. Starting from 1999, this project (hosted by the University of California, Berkeley), asked volunteer users to donate computing time to the analysis of signals that might originate from intelligent life outside Earth. This led to the BOINC[4] project (2002), a generic middleware for volunteer and grid computing. Moving away from volunteer computing, scientific and industrial workflows are today eager to follow the same logic and off-load their computation onto more versatile platforms built on top of Cloud solutions.

The first usage of Cloud was the deployment of web services and databases. For these services, virtualisation offers many advantages, particularly in term of QoS.

More recently, companies (like Amazon) with computing resources scaled for handling the increased load they might face during the Christmas holidays realised they could rent these resources when they are not in use. AWS (Amazon Web Service), the Amazon commercial Cloud, was launched in 2006. Microsoft launched a similar service (Microsoft Azure[5]) in 2010. Since then commercial Cloud diversified to include many providers (Figure 3) and service models to access and use these resources. These services models (Figure 4) are often referred to as "... as a Service".

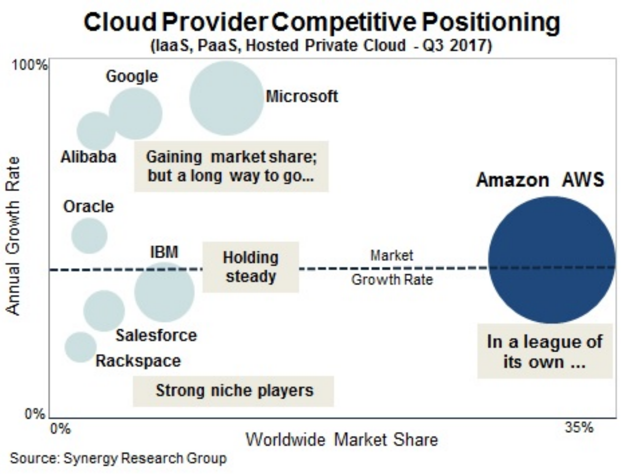


Figure 3: Market share and annual growth of the leading Cloud providers (Q3 2017)

IaaS: Infrastructure as a Service (IaaS) refers to Cloud services that provide access to low-level resources through full stack virtualisation. Resources available in an IaaS Cloud run a hypervisor which allows the end user to deploy their own operating system. This approach provides high versatility but is also the most complex to manage as the user is in charge of more layers. Using IaaS Cloud, institutions can move their entire infrastructure to Cloud while keeping their complete stack.

PaaS: Platform as a Service (PaaS) are Cloud services that provide direct access to a configured platform. Unlike IaaS, the user is here not required to deploy an operating system and can run their

[3]https://setiathome.berkeley.edu/

[4]Berkeley Open Infrastructure for Network Computing

[5]formerly Windows Azure

software on a preconfigures OS.

SaaS: Software as a Service (Saas) is a higher level model where the Cloud provider gives access to a specific software on an infrastructure it manages. When IaaS requires the user to deploy their own stack from the operating system all the way to the actual software, SaaS can deploy services like versioning tools, email servers, databases, or video conference service in just one click.

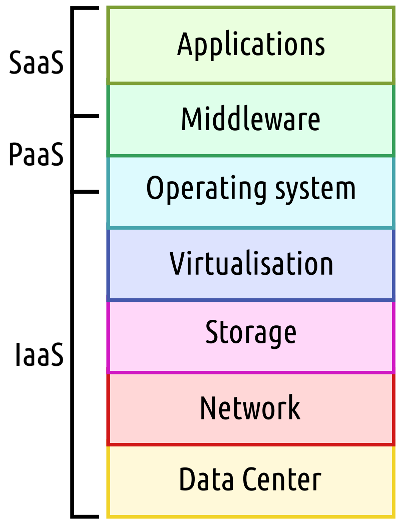


Figure 4: Layering of Cloud service models

Among the many other denominations that derive from these Cloud layers, Data as a Service (DaaS) is a specific type of SaaS that is focused on data storage. Similarly to SaaS, the Cloud providers offering this service manages all the system stack and only exposes an API to a piece of software. Solutions like Dropbox or Owncloud instances[6] fall into this category. Services with lower level interfaces such as NFS, FTP, or IPFS could also be considered as DaaS solutions.

Section I.3

Toward dynamic workflow on Cloud

During this PhD, we focused on the use of Cloud infrastructures that rely on IaaS or PaaS for the deployment of the computing resources and DaaS for the management of the data exchanged between these computing resources. Such a deployment could use nodes provided by Amazon EC2 for the computation while inputs and results are stored on the Amazon S3 storage solution. We thereafter designate such infrastructures as DaaS-based (or DaaS-centered) Cloud computing platforms.

The combination of these two paradigms (IaaS and DaaS) provides the possibility of elastic computing platforms with resilient, low maintenance storage. Such a model is very different from the existing usages of Cloud resources that focused on mimicking cluster infrastructure rather than building solutions tailored to the features of Cloud deployment.

Our objective is to design a middleware that would provide Workflow as a Service (WaaS) on top of an IaaS computing layer and a DaaS storage. This middleware should manage the computing layer so that resource deployment is transparent to the end user.

[6]or any other WebDAV – Web-based Distributed Authoring and Versioning – server

I.3.1 Related Work

As discussed hereabove, workflows can be found in many contexts and each one of these requires specific features from the engine in charge of their execution. In this section, we will discuss existing workflow engines and the related Cloud solutions.

An interesting point to notice is that this field deals with issues both on the theoretical side and on the implementation side. The optimisation of placement has been modelled extensively, and many solutions have been proposed to solve different variations of the problem. On the other hand, implementing a workflow engine is a difficult undertaking that requires significant engineering work. We, therefore, end up with many contributions that show a good understanding of the theoretical issues, but fail to discuss implementation details as well as many technical reports that detail workflow engines architecture but fail to model and comment the optimisation problem they try to solve.

Tasks vs Services

There are many aspects that differentiate workflow engines. The first distinction we can formulate is between the task-based and service-based approaches. This distinction focuses on the way users can express the computation they require. While choosing between these two strategies is mostly an implementation issue, it may have consequences on the scheduling policy when considering the batch processing of large datasets.

- In the task-based approach, users submit computing tasks that can use any executable, with any set of parameters. This is implemented in middleware such as GLOBUS [Foster, 2006]. This is more flexible for the user but reduces the middleware insight.
- The service-based approach relies on pieces of code (services) wrapped around standardised interfaces. Rather than describing the complete parameters for a task call, users can very easily make a call to the service. Reusability is increased and the middleware can learn on the past runs of this service to predict future executions. On the other hand, this interfacing mechanism requires an extra effort when implementing the service and can constrain the interactions with the inner piece of code. This approach is a the heart of the GridRPC paradigm and is implemented by middleware such as DIET [Caron et al., 2002], GridSolve [Yarkhan et al., 2007] and Ninf [Tanaka et al., 2003].

In a service-based approach, once the services have been written and made available, any user can use them to build their workflows, but deploying the workflows then requires nodes providing the services. The workflow engine has to account for the availability of the services on the different nodes.

Still, when a task is executed many times through multiple executions of the same workflow, having an already deployed service, ready to process any input data, can save time and increase the throughput of the platform.

Data composition

Workflows, like any other application, are designed to process pieces of data. It is the datasets that are processed by the workflows which constrain the execution of the underlying tasks. When a single execution of a workflow is required, the tasks or services this workflow depends on will have to be deployed for this specific execution. On the other hand, if a large dataset is to be processed by many executions of the same workflow, these executions can be pipelines and the underlying services reused.

When using workflows to process a combination of different datasets, the structure of the datasets and the way they are combined have a direct impact on what would an effective deployment be. Parallelisation of pipelined computation is discussed in [Subhlok and Vondran, 2000]. The issue of data composition and its effects on the effective placement of services-based workflows is extensively discussed in the MOTEUR workflow engine [Glatard et al., 2006].

DAG placement

Far from pipelined workflows, the issue of single execution of workflows on dedicated (not shared) platforms has been extensively modelled and studied.

Clustering algorithms are typical answers to this issue. Their goal is to pack tasks into clusters. Given a global vision of the platform and the tasks to execute, a task-clustering is a groupement of the tasks that are to be executed on the same nodes. The underlying problem is very complex[7] and it is expensive to achieve an efficient task-clustering. Still, any change concerning the platform or the tasks to execute would lead to the whole clustering becoming invalid.

In this category, we found algorithms dealing with both homogeneous platforms [Wu and Gajski, 1990, Yang and Gerasoulis, 1993, Yang and Gerasoulis, 1994, Kwok and Ahmad, 1994] and heterogeneous platforms [Topcuouglu et al., 2002]. While their use is relevant for the execution of workflows on computer clusters with a fixed number of nodes, they were designed at a time when IaaS Clouds were not even an idea and are not designed to work with elastic platforms nor with the specificities of DaaS-based communications.

Communication models

Among the workflow models used to build clustering algorithms, most consider communication cost between tasks. However, the communication models have not evolved with technology, and we argue in Chapter II that they do not match the reality of DaaS solutions.

Actual workflow engines are not that talkative about communication models. Data locality is known to be an issue, but surveys show that it is rarely considered as part of the optimisation of workflow deployment [Wieczorek et al., 2009, Bala and Chana, 2011]. Moreover, when this issue was considered, it was on simplified DAGs which failed to model the whole extent of real applications accurately.

Workflows on Cloud

Execution of workflows on Cloud platforms combines the intrinsic complexity of both approaches into an optimisation nightmare. Workflow scheduling is a graph matching problem where the workflow is a graph of tasks, the platform is a graph of resources, and the matching has to consider QoS constraints and optimise communication induced latency. Building heuristics to solve this problem is already difficult, but in a Cloud environment we are dealing with an elastic graph of resources (nodes can be added on-demand), we have to consider this elasticity as part of the optimisation objectives and,

[7]Optimal workflow clustering is generally NP-Complete (see Chapter II)

last but not least, we want to share the platform (which means mapping workflows on top of a graph of resources that is already constrained by previous jobs).

Lin *et al.* [Lin and Wu, 2013, Wu et al., 2015] discuss the complexity (NP-completeness) of workflow placement and discuss heuristics for the placement of workflows on Cloud infrastructure. Their approach considers a global vision of the deployment, from tasks to VMs to physical infrastructure. Their communication model is based on point-to-point messages. Moreover, their solution deals with the placement of a fixed workload which means it does not allow for real-time scaling of a shared platform in a multi-tenant context.

Mao *et al.* [Mao and Humphrey, 2013] dealt with a single workflow in which the tasks have different resources requirements. Their algorithm not only packs tasks but also determines which type of nodes to deploy. However, this result does not account for multiple workflows. It also does not handle contexts in which communications go through a DaaS. We might be interested in investigating the underlying communication model with respect to the targeted platform's behaviour.

Malawski *et al.* [Malawski et al., 2012] handled multiple dynamically submitted workflows composed of single-threaded tasks. In this work, the Cloud platform used for execution is dynamic but has an upper bound. Therefore, some workflows – with a lower priority – can be dropped if the platform cannot be extended as required. Similarly to Mao *et al.* [Mao and Humphrey, 2013], the communication model used in this paper does not match our observations of DaaS-based platforms.

Many workflow engines are capable of handling workflows in Cloud environments. However, the optimisation mechanisms are rarely discussed. When looking in detail, the proposed solutions often deal with resource deployment and configuration (a complex engineering issue) but fail to optimise the use of Cloud specific features. As an example, the Cloud-Batch of DIET can deploy Cloud instances that contain a particular service. Before running any operation, the engine will deploy a dedicated Cloud instance for this operation to be executed on. Once the computation has been performed, the instance is released and a new one is allocated for the next operation. This is particularly inefficient. Data locality cannot be achieved as all operations are running on different instances. Besides, deployment and configuration time slows down the execution of the workflows. Using instances for the execution of multiple tasks is at the heart of our solution. Chapter II discusses the construction of blocks of tasks to be executed by a single instance and Chapter III discusses these instances life cycle.

Cloud deployment

Solutions to dynamically scale Cloud computing instances exist. For example in [Mao et al., 2010] the solution is based on deadline and budget information. In [Kailasam et al., 2010] the solution deals with Cloud Bursting. Another autonomic auto-scaling controller, which maintains the optimal number of resources and responds efficiently to workload variations based on the stream of measurements from the system, is introduced in [Londoño-Peláez and Florez-Samur, 2013]. This paper shows the benefit of auto-scaling solution for Cloud. Unfortunately, all these solutions do not handle workflow applications. Likewise in [Nikravesh et al., 2015], authors gave a suitable prediction technique based on the performance pattern, which led to more accurate prediction results. Unfortunately, they do not consider the delays resulting from communications between the different tasks of a workflow.

In [Mao and Humphrey, 2013], the authors show the benefit of auto-scaling to deal with workflows. They show that the scheduling-first and scaling-first algorithms have different advantages over each other within different budget ranges. The auto-scaling mechanism is introduced as an exciting research direction for future work.

In [Heinis et al., 2005] an autonomic controller respond to workload variations to reconfigure a cluster's configuration. Authors introduce a mechanism of self-healing in case of configuration update of the cluster. This solution shows some benefits from cluster architecture but does not work well for Cloud architecture communication mechanism, particularly in the context of DaaS-based Cloud platforms. Details of a workflow engine for Cloud environment dealing with dynamically scalable runtime are given in [Pandey et al., 2012]. This work is efficient for an ECG[8] analysis application but it is inadequate to re-use for other applications.

Section I.4

Positionning and structure of our work

We saw that many interesting results have been achieved that relate to workflow execution on DaaS-based Clouds. Yet, none of these provides all the features that are essential to achieving full autonomy. This comforts our quest toward an efficient and fully autonomous Workflow as a Service engine. The structure of this engine and the different modules that compose it focus on solving the issues that have emerged hereabove:

- Providing mechanisms for the optimisation of workflow execution on DaaS-based Cloud computing platform through data locality (see Chapter II);
- Efficiently deploying and managing an elastic DaaS-based Cloud computing platform (see Chapter III);
- Using this platform to execute the tasks or services required by many workflows in a way that satisfies the users (see Chapter III).

I.4.1

Objectives and constraints

The workflow and Cloud environments are feature-rich, and building a single solution to all issues is unrealistic. To build a solution we must first formalise the problem. This means specifying the execution environment, user constraints, and the metrics we want to optimise.

Workflow format

We focus on unconditional workflows represented as DAG. We do not impose any constraints on the structure of the DAG, but the static analysis step requires this structure to be known before the execution. Details of the data representation will be discussed in Chapter II. We also need a forecast of the tasks runtime (or at least we expect a reasonable prediction). Similarly, we assume knowledge of the size of all pieces of data which are involved in the workflow execution.

This structure and the assumption that comes with it are compatible with the workflow syntax used by Pegasus [Deelman et al., 2005].

[8]ECG: ElectroCardioGram

Computing environment

The execution of workflows is to be performed on Cloud computing resources. The computing resources themselves would be IaaS or PaaS instances. As we target scientific workflows that are usually executed on homogeneous clusters (an example is discussed in Chapter V), our work is focused on the usage of homogeneous Cloud instances. If all computing nodes have identical specifications they can be viewed as fungible assets by the deployment manager.

Data communication relies on a DaaS solution. The users would upload input data to the DaaS, and output data produced by the workflows would be placed on the DaaS for the users to retrieve them. Intermediary pieces of data could be stored locally on the computing instances if the task that requires them runs on the same instance as the one that produced it. If it is not the case, then the data transfer between the instances hosting both tasks would have to go through the DaaS.

While the DaaS could be implemented in many ways (and could very well rely on a distributed infrastructure), we consider that once an object has been uploaded to it, anyone, anywhere, can download it. From a synchronisation point of view, the DaaS is, therefore, a central point in the computing infrastructure. Workflow clustering requires a model of the computing platform which is discussed in Chapter II. The model we designed as part of this work assumes we have metrics on the bandwidth between the IaaS/PaaS computing instances and the DaaS storage.

Our objective is to achieve an efficient deployment by sharing computing instances between tasks and even between workflows. In particular, we want to avoid the "one instance per task" logic that wastes a lot of time configuring VMs. Rather, we want to benefit from the elasticity of Cloud solutions and not just allocate a fixed number of resources regardless of the actual workload.

Constraints and optimisation

Users submit their workflows along with some requirements. These requirements will be the constraints of the optimisation problem. In our approach, the users provide a deadline for the execution of its workflow. If the deadline cannot be achieved the workflow engine will take actions to obtain the result as fast as possible. On the other hand, if there is some margin, the workflow engine will have some wiggle room to optimise the Cloud deployment.

The goal of this optimisation is simple: reduce the deployment cost of the Cloud computing platform. This cost is the consequence of the deployment required by the middleware and the Cloud provider policy.

Our model considers that the cost of a platform is the sum of the deployment cost of all instances. Similarly to Amazon EC2[9], we consider instance-hours as the elementary billing item. This policy matches the commercial practices and is also a useful metric of the deployment efficiency as it favours busy instances. It penalises the fact of having idle instances as well as the rapid allocation and deallocation of under-used instances.

I.4.2 Proposed architecture

To answer this scheduling and deployment problem, and to address the issues listed hereabove, we propose a modular structure for a fully autonomous workflow manager. This structure is shown in

[9]https://aws.amazon.com/premiumsupport/knowledge-center/ec2-instance-hour-billing/

Figure 5 and was presented in [Croubois, 2016]. The modules it contains are dedicated to meeting the different challenges we identified earlier.

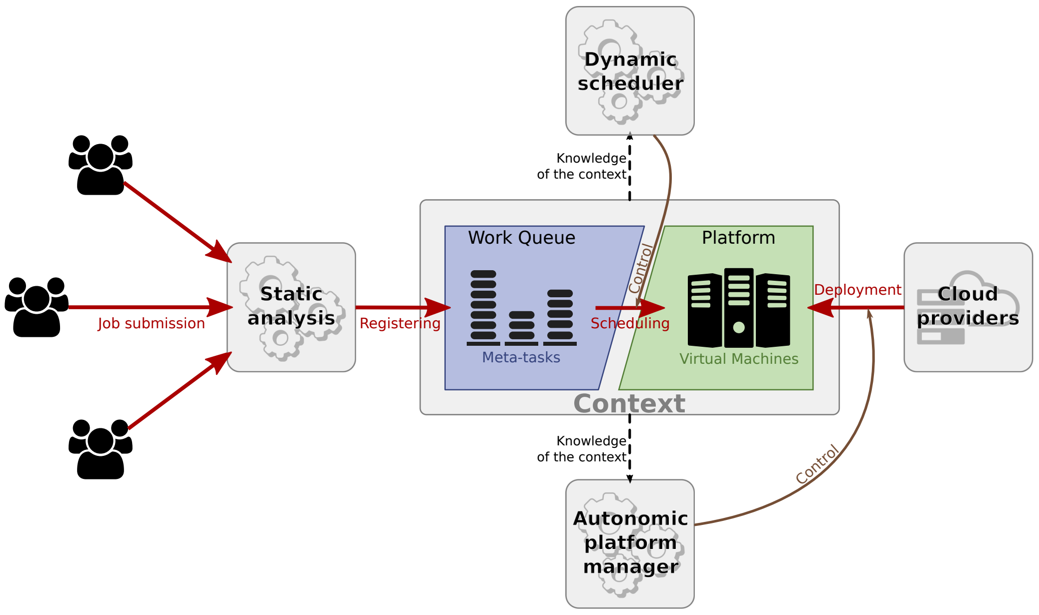


Figure 5: Overview of the envisioned modular structure for a fully autonomous workflow engine.

Static analysis

The first module in our framework is the static analysis step. This module is in charge of the offline optimisation of workflows. The objective is to precompute meta-data that will later help in the placement and execution of the relative workflows on the shared platform.

Computations performed in this module are oblivious to any specific deployment of the platform. This implies that this step can be parallelised and performed by the users requesting the execution, which is favorable to the scalability of the solution. On the other hand, choices made during this process are not aware of other similar decisions that could have been taken for optimising other workflows. This limits the range of possible optimisations to the construction of abstract representations.

Theoretical models used for the optimisation of workflows as well as the construction of abstract workflow representations are discussed in Chapter II.

Once offline optimisation is achieved, the analysed workflows are registered in queues. These queues, along with the deployment of the platform at a given time, constitute the execution context.

Dynamic scheduling

The second module contains the scheduling logic. This logic is similar to that of existing workflow engines but needs to work with the abstract workflow representation computed by the static analysis step. Besides, the scheduling must account for the elasticity of the platform when assigning (or not assigning) tasks.

This module is one of two controllers that require the knowledge of the execution context. While we consider this module as a single logical entity, its logic (detailed in Chapter III) is distributed between the different agents of our middleware.

Autonomic platform management

The last issue we have to tackle is platform management. A key feature we want to include is that our workflow engine should handle the deployment of Cloud instances automatically. This is the mission of the last module.

The autonomic platform manager requires periodic updates on the execution context (task queues and platform status) to adjust the Cloud deployment. This is essential to meeting the QoS requested by the users (having enough resources) while limiting the deployment cost to a minimum (not having too many resources).

Similarly to the dynamic scheduling module, the autonomic platform manager can be seen as a single entity but is in fact distributed (see Chapter III) to achieve good scalability.

I.4.3 Plan

Chapter II discusses the static analysis module. This includes discussion about the workflow representation, the construction of a communication model for DaaS-based Clouds and the use of this model for the clustering of workflows. These are contributions of our work that have been published in [Croubois and Caron, 2017].

Chapter III discusses the distributed infrastructure in charge of online workflow scheduling and platform management. We discuss the structure of the middleware as well as details of the algorithms that implement global behaviours in a decentralised way. Finally, we benchmark the effectiveness of the platform through simulation. The contributions of this chapter (distributed infrastructure and autonomic deployment loop) have been published in [Bonnaffoux et al., 2018]

Chapter IV discusses the implementation of our autonomous workflow engine logic in the DIET middleware. The contribution of this chapter lies in the adaptation work that was required to turn our distributed mechanisms into modules within the constraints of a real middleware.

Chapter V discusses an application use-case, WASABI, that illustrates the effectiveness of our method for the execution of scientific workflows.

Chapter II

Static workflow analysis

The first step in our journey toward an autonomic workflow engine is the pre-processing of workflows. We will see in this chapter that workflow scheduling is a complex issue and that solutions are based on heuristics. We will discuss previous models and heuristics and look at how we can improve them.

Our objective in this chapter is to design meta-data and structures around the workflows. They are computed before the execution even starts (static analysis), and will prove usefull at runtime. We want to achieve the maximum here while keeping in mind that the static analysis step is performed off-line, and has therefore no knowledge of the specificity of the workload or the platform deployment at the time of the execution.

Section II.1 A brief history of graph scheduling

II.1.1 Notations

Task Graph

A task graph is a directed weighted graph $G = (V, E, \omega)$ where:

- The set V represents the tasks
- The set E of edges represents the dependencies between tasks: $e = (u \to v) \in E$ if and only if $u \prec v$.
- A weight function $\omega : V \to \mathbb{R}^*$ gives the weight or execution time of a task.

The precedence postulate $u \prec v$, with u and v tasks of the graph designates the fact that v depends on u and can only be executed after it. $\forall v \in V$, we note:

- $Pred(v)$ the set of immediate predecessors of task v.
- $Succ(v)$ the set of immediate successors of task v.

We say that v is an entry task if $Pred(v) = \varnothing$ and that v is an exit task if $Succ(v) = \varnothing$

Schedule and allocation

A schedule of a task graph $G(V, E, \omega)$ is a function $\sigma : V \to \mathbb{R}$ such that:

$$\forall (u \to v) \in E, \quad \sigma(v) \geq \sigma(u) + \omega(u) \tag{Eq. 1}$$

σ can be seen as the starting time of the tasks. A simple result states that there exists a valid schedule for G if and only if G does not contain any cycle. Graphs of tasks with cycles are ill-formed, and we will only consider acyclic ones. Therefore, we will indifferently use the term DAG of tasks (Directed Acyclic Graph) to designate the graph of tasks.

If we have a limited number p of processors[10], we consider the placement function $alloc : v \in N \rightarrow [\![1, p]\!]$ that describes on which processor each task is executed. This adds constraints to the schedule σ, which are often expressed as implicit dependencies between independent tasks allocated on the same processor.

$$\forall u, v \in N, \quad alloc(u) = alloc(v) \Rightarrow \sigma(v) \geq \sigma(u) + \omega(u) \vee \sigma(u) \geq \sigma(v) + \omega(v) \qquad \text{(Eq. 2)}$$

Makespan

Let $G(V, E, \omega)$ be a graph of tasks and σ be a schedule for G using p processors. The makespan of G is the total execution time:

$$MS(\sigma, p) = \max_{v \in V} (\sigma(v) + \omega(v)) - \min_{v \in V} (\sigma(v)) \qquad \text{(Eq. 3)}$$

Usually we assume that:

$$\min_{v \in V} (\sigma(v)) = 0$$

The makespan represents the computation time of the whole DAG of tasks. $PB(p)$ is the name given to the problem of finding an optimal schedule (i.e. a schedule of minimum makespan) using p processors. $MS_{opt}(p)$ be the makespan of an optimal schedule using p processors.

Top- and Bottom- levels

A useful tool to build efficient schedules for the PB problem is the computation of the top- and bottom- levels. The top-level $tl(v)$ (see Equation 4a) is the maximum weight of a path from an entry task to the task v, excluding the weight of v.

$$tl(v) = \begin{cases} 0 & \text{if } Pred(v) = \varnothing \\ \max_{u \in Pred(v)} (tl(u) + \omega(u)) & \text{otherwise} \end{cases} \qquad \text{(Eq. 4a)}$$

The bottom level $bl(v)$ (see Equation 4b) is the maximum weight of a path from the task v (included) to an exit task.

$$bl(v) = \begin{cases} \omega(v) & \text{if } Succ(v) = \varnothing \\ \omega(v) + \max_{u \in Succ(v)} (bl(u)) & \text{otherwise} \end{cases} \qquad \text{(Eq. 4b)}$$

Both can be computed through a traversal of G in $\mathcal{O}(|V| + |E|)$

II.1.2 Scheduling without communications

So far we are disregarding the communications between tasks. Still, even this case can be challenging.

[10]if $p < |V|$

$PB(\infty)$

The problem of DAG scheduling on an infinite number of processors without communication is simple. Thanks to the *tl* and *bl* computation we can build an optimal schedule using one processor per task:

$$\sigma_{ASAP} : v \to tl(v)$$

This schedule, called ASAP[11], is optimal by construction. Similarly, we can build another optimal schedule, called ALAP[12]:

$$\sigma_{ALAP} : v \to MS_{OPT}(\infty) - bl(v)$$

$PB(p)$

Unlike $PB(\infty)$, the problem of DAG scheduling on a limited number of processors, even without communication, is known to be NP-complete[13] (see [Garey and Johnson, 1990]). Many other schedule optimisation problems, with more complex assumptions than just "a single graph of task on p processors" are also NP-complete. At this point, we are surrendering the idea of computing optimal solutions, and we are focusing instead on building heuristics that provide good-enough solutions.

List scheduling heuristics

The idea behind list scheduling heuristics is simple: Assign priorities to tasks, sort them by priority and greedily schedule them. The objective is to avoid idle processes. That is achieved by feeding them the most urgent (ready) tasks. The implementation of this mechanism is simple and guarantees decent results. Graham showed [Coffman and Graham, 1972] that for any list scheduling heuristic:

$$MS(\sigma, p) \leq \left(2 - \frac{1}{p}\right) MS_{OPT}(p) \qquad \text{(Eq. 5)}$$

A priority commonly used for list scheduling is the bottom-level of the task. The bottom-level is the lower bound on the remaining execution time of the graph from the state of the execution of the considered tasks. It is a good indicator of the urgency of the task as it measures the importance of its execution to unlock further tasks.

II.1.3 Scheduling with communications

The dependencies between tasks in a workflow $G(V, E, \omega)$ are justified by data shared between the tasks. If task u produces a piece of data d, which is one of the inputs of task v, then task v cannot start before the end of task u. This is denoted by the edge from u to v in E. On the other hand, if two tasks u and v do not share any data, we can execute them in an arbitrary order or in parallel.

[11] As Soon As Possible
[12] As Late As Possible
[13] reduction from 3-partition

The model commonly used to express the latency induced by the communication between tasks is as follows. For a graph of task $G(V, E, \omega)$ with $\omega : V \cup E \to \mathbb{R}^*$ extended to measure communication duration between two dependent tasks, and an allocation $alloc$, we can extract an edge cost c_{alloc}:

$$\underset{(u \to v) \in E}{c_{alloc}(u \to v)} = \begin{cases} 0 & \text{if } alloc(u) = alloc(v) \\ \omega(u \to v) & \text{otherwise} \end{cases} \tag{Eq. 6}$$

In this context, a schedule σ must satisfy the updated dependencies constraints:

$$\forall (u \to v) \in E, \quad \sigma(v) \geq \sigma(u) + \omega(u) + c_{alloc}(u \to v) \tag{Eq. 7}$$

PB with communications

With communication added to the model, the placement problem, expressed through the *alloc* function, becomes even more important. In the previous model, the workflow structure was fixed and independent of the placement decisions. However, when taking communication into account, placement decisions affect the top- and bottom- level measurement. This considerably increases the complexity of the scheduling problem.

A consequence is that the $PB(\infty)$ problem, which is easily solved without communications becomes NP-complete when adding communications[14]. Same goes for $PB(p)$, which imposes restrictions on $PB(\infty)$ and is, therefore, more difficult to solve. As discussed previously, we should not await optimal solutions to interesting scheduling problems, but rather rely on heuristics to produce good approximations.

Naive critical path

The critical path method is a list scheduling approach which uses the bottom-level for priority. In this new context, top- and bottom- level computations can be updated to account for communications:

$$tl_{alloc}(v) = \begin{cases} 0 & \text{if } Pred(v) = \varnothing \\ \max\limits_{u \in Pred(v)} \left(tl_{alloc}(u) + \omega(u) + c_{alloc}(u \to v)\right) & \text{otherwise} \end{cases} \tag{Eq. 8a}$$

$$bl_{alloc}(v) = \begin{cases} \omega(v) & \text{if } Succ(v) = \varnothing \\ \omega(v) + \max\limits_{u \in Succ(v)} \left(c_{alloc}(v \to u) + bl_{alloc}(u)\right) & \text{otherwise} \end{cases} \tag{Eq. 8b}$$

Considering the worst case scenario, where $\forall u, v \in V, alloc(u) \neq alloc(v)$, the bottom level with communication (see Equation 8b) gives an idea of the urgency of the tasks. This measurement can be used to adapt the list scheduling approach to graphs of tasks with communications.

[14] reduction from 2-partition

Modified critical path

The modified critical path approach is similar to the critical path approach in that it considers ready tasks by order of priority (using the bottom-level). However, contrary to traditional list scheduling algorithms which place the tasks on the earliest available processor, the tasks are here placed on the processor where they would start the earliest, considering communication costs and previous task placement.

DCP, an Edge-Zeroing based clustering algorithm

With the update of the top- and bottom- level formulas to account for communication cost (see Equation 8) DAG scheduling is now focused on the issue of data locality. Rather than deciding when to schedule the task, we first go through the process of deciding which tasks should be grouped together to avoid communication costs. This is called clustering. Clusters of tasks are structures which contain tasks that should run on the same processor, regardless of the number of processors available. It is the dual representation of the *alloc* function discussed earlier.

$$\forall v \in V, \forall t \in [[1,p]], \quad v \in \mathcal{C}(t) \Leftrightarrow alloc(v) = t \qquad \text{(Eq. 9)}$$

Edge-zeroing based merging algorithms constitute an important subclass of clustering algorithms. Their role is to build a sequence of clusterings that converge toward a good solution of the clustering problem, which is part of the scheduling problem. Starting from an initial clustering $\mathcal{C}_0$, where each task is alone in its cluster[15], algorithms in this class use an iterative process to upgrade the clustering. In order to transition from a clustering $\mathcal{C}_{i-1}$ to the next one $\mathcal{C}_i$, we consider an edge (or a set of edges) to zero. Zeroing an edge corresponds to merging the clusters containing both sides of the edge to remove the communication cost associated with the edge. Before applying this transition, we first verify that the change in clustering translates into a diminution of the makespan associated. Multiple examples of such algorithms are described in [Gerasoulis and Yang, 1992].

DCP [Kwok and Ahmad, 1994] has been a reference for the scheduling of workflows with communications. It is an edge-zeroing algorithm that recomputes the critical path at each step and tries to remove the largest edge in it (see Algorithm 1).

Toward heterogenous platforms with HEFT

Heterogeneous Earliest Finish Time [Topcuouglu et al., 2002], or HEFT, is a scheduling heuristic that deals with fixed size heterogeneous platforms. In this context, the targeted platform contains p processors with different characteristics such that the runtime of the tasks $\omega(v)$ can differ arbitrarily between processors. Similarly, the communication cost $\omega(u \to v)$ between task u and task v can be different for each pair of processors running tasks u and v.

HEFT relies on a list-scheduling approach similar to the modified critical path. It places the tasks in the node where they would finish the earliest, considering the current usage of the nodes and the runtime on each one of them.

This list scheduling approach requires a priority ranking of the tasks. The priority used by HEFT is an adaptation of the bottom level that considers average values:

[15] equivalent to having a bijection from $N \to [[1, |N|]]$ as *alloc*

Algorithm 1 DCP static scheduling algorithm

```
 1: $\mathcal{C} \leftarrow$ empty clustering (one virtual processor per task)
 2: compute $TL$ and $BL$ for each task using $\mathcal{C}$
 3: while $\exists$ unmarked dependency between tasks do
 4:     $e = (u \to v) \leftarrow$ unmarked edge with the largest path length and size.
 5:     $\mathcal{C}' \leftarrow \mathcal{C}.mergeClusters\,(\mathcal{C}(u), \mathcal{C}(v))$
 6:     compute $TL'$ and $BL'$ for each task using $\mathcal{C}'$
 7:     if $DCPL(TL', BL') \leq DCPL(TL, BL)$ then
 8:         $(\mathcal{C}, TL, BL) \leftarrow (\mathcal{C}', TL', BL')$
 9:     end if
10:     mark $e$
11: end while
12: return $\mathcal{C}, TL, BL$
```

Line 4: use a lexicographical order. First select the largest path length, then among these select the largest edge.

$$rank_{HEFT}(v) = \bar{\omega}(v) + \max_{u \in Succ(v)} \left(\bar{\omega}(v \to u) + rank_{HEFT}(u)\right) \qquad \text{(Eq. 10)}$$

with $\bar{\omega}(v)$ the average value of $\omega(v)$ accross all nodes, and $\bar{\omega}(v \to u)$ the average value of $\omega(v \to u)$ accross all pairs of distinct nodes.

Section II.2 Models for the DaaS-based Clouds and workflows

II.2.1 Network contention on DaaS-based Clouds

DaaS-based Clouds rely on a different communication pattern than previous grid-like approaches. Rather than performing point-to-point communications between the nodes, data transfers are a two-step process. The pieces of data are first uploaded to the DaaS and then, subsequently, downloaded from the DaaS. This communication pattern has major consequences on the execution of workflows with data dependencies as we will now see.

II.2.1.A *Workflow clustering and Cloud elasticity*

The edge-zeroing approach discussed previously, followed by the DCP algorithm, makes a lot of sense in the context of Cloud platforms. The first part of the two-step scheduling builds an efficient clustering of tasks that reduces the execution makespan (with good data-locality) using an unbounded number of virtual processors. The second step is in charge of mapping these clusters to the available resources.

However, in a Cloud context, we are not limited to a given number of resources, and the clustering performed during the first step is a good indicator of the number of resources needed for an optimal execution. Clustering algorithms are therefore interesting to use for Cloud deployment, providing they fit the communication paradigm of the targeted platforms.

Furthermore, when resources available for deployment are unbounded, the clustering of a given workflow remains valid regardless of any other workload. Indeed, tasks which are not part of the same graph of tasks, and have no dependency path between them, do not affect each other. They could have a negative impact on one another if they were to exploit resources required by the other, but if potential resources are unbounded like in Cloud platforms then this issue is irrelevant.

II.2.1.B *DCP scheduling on Cloud platforms*

A typical workflows structure is the fork-join pattern (see Figure 6). In these workflows, an entry task first sends pieces of data to n independent child tasks (fan-out). Afterwards, these n child tasks are sending back the results to a final exit task (fan-in).

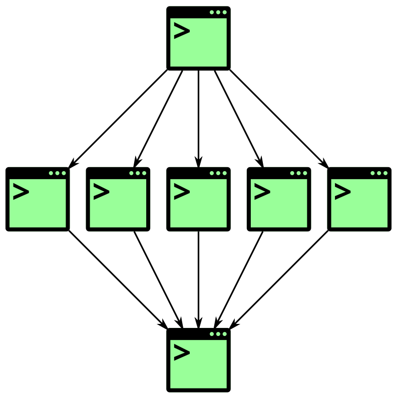


Figure 6: Structure of a fork-join DAG.

When scheduling such a DAG using DCP, the algorithm takes action to achieve a high level of parallelism and thus places most tasks on distinct nodes. However, when executing the resulting task placement on a Cloud infrastructure, we see that communications, on the one hand, between the first task and the DaaS (upload of n files to the DaaS) and, on the second hand, between the DaaS and the final task (download of n files from the DaaS) drastically suffers from contention. Figure 7 shows that while DCP plans for all transfers to be simultaneous, the network congestion drastically reduces the performances of both uploads to the DaaS and downloads from the DaaS. This results in transfer times about n times slower than predicted by DCP.

This gap between the communication model underlying DCP and the reality of prevailing distributed platforms explains why such scheduling algorithms, despite making a lot of sense, are actually not efficient for tasks placement on Cloud platforms.

A closer look at the top- and bottom- level equations (see Equation 8) shows that these formulas do not consider the possible impact that simultaneous transfers could have on one another. In fact, they disregard any form of contention and consider a clique (see Figure 8) where any pair of nodes can exchange data without being affected by the rest of the network. Even worst, this model considers that any number of transfers between the same two nodes can take place at the same time without them having to share the bandwidth.

While such a topology manages to depict the point-to-point communications in small clusters, the gap with new distributed platforms is tremendous and explains the incapacity of DCP to predict communications in DaaS-base Cloud platforms efficiently. This leads to poor performances of the resulting task placements.

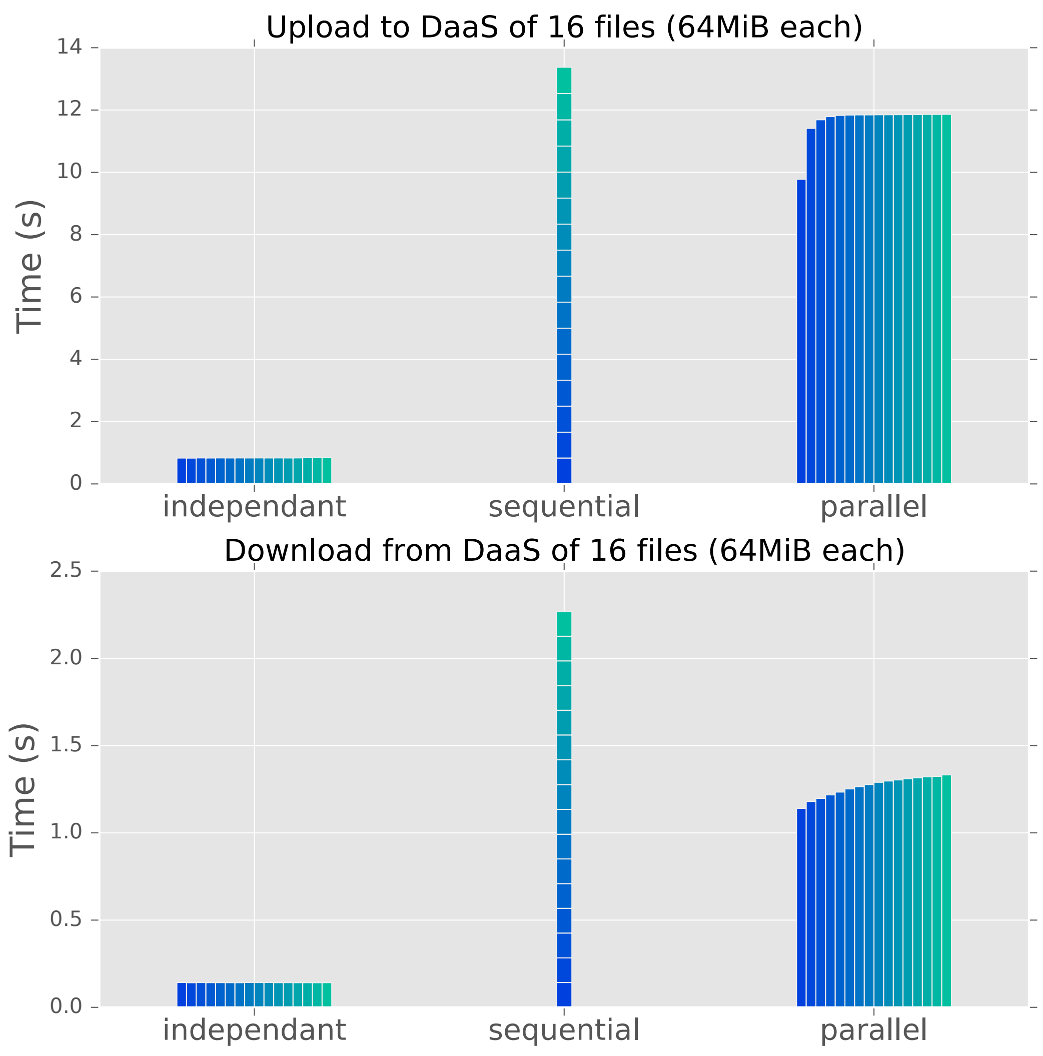


Figure 7: Transfer times of 16 files from and to a DaaS: (left to right) predicted by DCP, experimental for sequential transfers, experimental for parallel transfers. Experiments were carried out using Grid'5000 testbed. Used nodes were on the Sagittaire cluster and the DaaS was a 10G chunk reserved on storage5k.

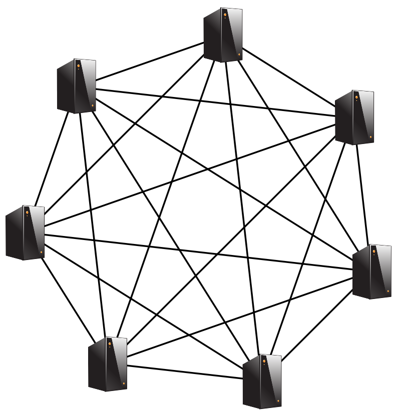


Figure 8: A clique network topology, which corresponds to the communication model underlying DCP's computation.

II.2.2 A new network model for a new clustering heuristic

As discussed in the introduction, communicating through a DaaS induces a centralisation of the communications. In an effort to build an efficient workflow management mechanism for DaaS-based Clouds, we need a scheduling policy both task and data-transfer oriented. Therefore, we want a better model to predict the behaviour of data transfers on this platform.

Our analysis is that this centralisation will lead to contention at the links between a node and the DaaS. While the DaaS is a critical element, commonly located near the centre of the network, the computing stations can potentially be very far from it. This led us to a model where a node ability to place and retrieve data from the DaaS is constrained by the bandwidth between the node itself and the core of the network. This topology (see Figure 9) neglects the contention that can happen at the DaaS level.

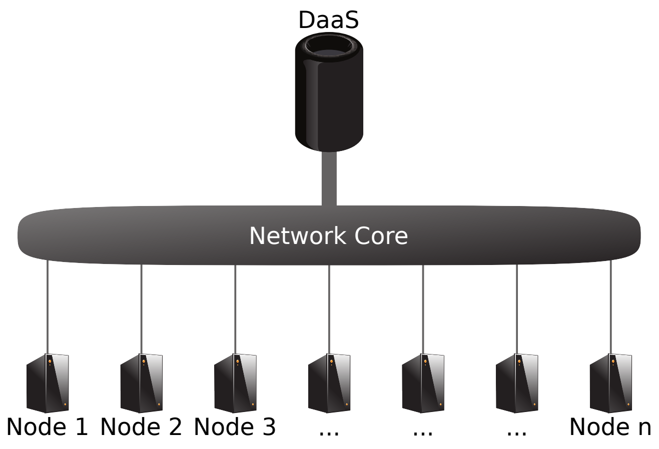


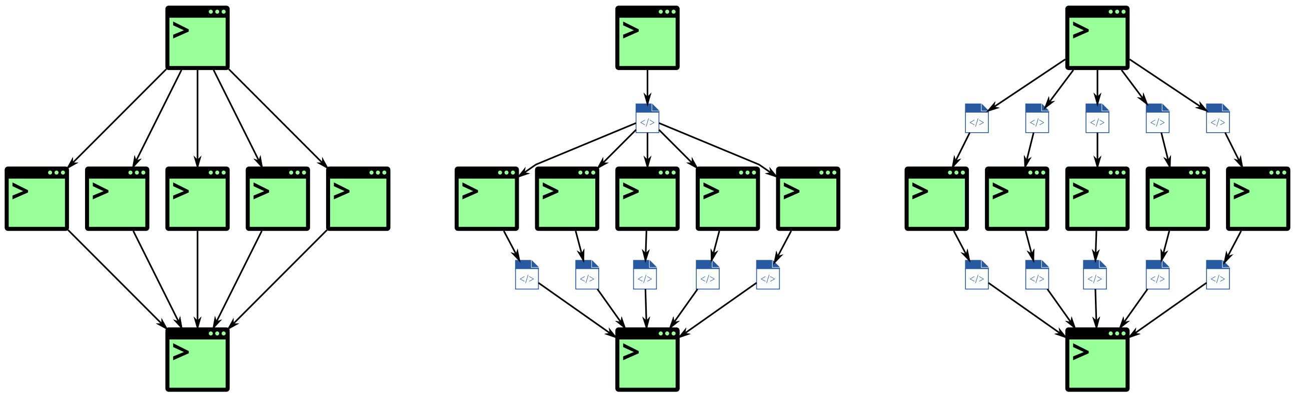
Figure 9: A generic model of DaaS-based network topology.

This model is generic enough for it to be independent of the actual Cloud deployment and VM migrations, as long as the nodes bandwidth is not overestimated.

II.2.3 A data-centric representation of workflows

As we discussed earlier, workflows are usually represented as weighted graphs of tasks, with the weights on the nodes representing the computational cost of the tasks (in flops – floating point operations –) and the weights on the edges representing the cost of the communications (in bits of data transferred). This model was discussed in Section II.1. However, as we moved to DaaS-based Cloud platforms, our representation of workflows needs to evolve accordingly.

While previous representations focused on the amount of data to be transferred between tasks, a more relevant approach would be to focus on data objects.



(a) Legacy representation (b) Our data-centric representation (single data upload) (c) Our data-centric representation (multiple data upload)

Figure 10: Legacy and data-centric representations of fork-join DAGs.

If we consider a fork-join distribution pattern, there is a substantial difference between sending a single piece of data to n different agents (see Figure 10b), and sending n different pieces of data to the same n different agents (see Figure 10c). Whilst in the first case we have a single upload from the initial task to the DaaS and n parallel downloads from the DaaS, in the second case we have a single task (the initial task) uploading the n different pieces of data to the DaaS at the same time, which would cause contention between the initial task and the core of the network where the DaaS is located.

Notation	Space	Description
$\mathcal{T}$		Set of all tasks
$\omega(t)$	$\mathcal{T} \mapsto \mathbb{R}$	Computational cost of t
$\mathcal{D}$		Set of all pieces of data
$d.src$	$\mathcal{T}$	Producer of data d
$d.dst$	$\wp(\mathcal{T})$	Consumers of data d
$\omega(d)$	$\mathcal{D} \mapsto \mathbb{R}$	Communication cost of d
$alloc$	$\mathcal{T} \mapsto$ VMs	Task placement

Table 1: Workflow and Clustering notations.

We therefore extend the workflow model to give a “data-centric” representation of workflows. It includes details about the different pieces of data produced and consumed by the tasks. Our representation (Table 1) is an acyclic-oriented bipartite graph, with nodes from one side representing weighted tasks and nodes from the other representing pieces of data weighted by their size. Edges have no weight and only represent dependencies between pieces of data and their producers/consumers. Figure 10 shows how two very different communication patterns have the same legacy representation. According to our network model, we expect bouts of network congestion when a single task uploads or downloads multiple files. Thus, we can expect some contention for the final task downloads in both cases (Figure 10b and Figure 10c). Yet, only the second case (Figure 10c) should undergo upload contention for the initial task.

Section II.3

DaaS-aware DCP

In the previous section, we discussed the behaviour of DCP and its inadequacy to schedule workflows on DaaS-based Cloud platforms efficiently. We also proposed a new network model which can be used to improve DCP ability to schedule workflows on modern platforms as well as a data-centric representation of workflows.

Using the unmodified structure of the DCP algorithm (see Algorithm 1, Section II.1.3), our objective is to use the platform and workflow models developed in Section II.2.3 to modify the way communications affect workflow clustering.

DCP uses the top- and bottom- levels formulas (see Equation 8) which estimate communication using the $c_{alloc}(u \to v)$ formula (see Equation 6). This is the part that needs being modified in order to match our communication model.

$$
\begin{aligned}
c_{alloc}(u \to v) &= 0, && \text{if } alloc(u) = alloc(v) \\
c_{alloc}(u \to v) &= \sum_{\substack{d \in data \\ u \in d.src \\ v \notin d.dst \\ islocal_{alloc}(d)=0}} \omega(d) && \text{(Eq. 11a)} \\
&\quad + \sum_{\substack{d \in data \\ u \in d.src \\ v \in d.dst}} \omega(d) + \max_{\substack{d \in data \\ u \in d.src \\ v \in d.dst}} \omega(d) && \text{(Eq. 11b)} \\
&\quad + \sum_{\substack{d \in data \\ u \notin d.src \\ v \in d.dst \\ islocal_{alloc}(d,v)=0}} \omega(d) && \text{(Eq. 11c)}
\end{aligned}
$$

The modification described in Equation 11 involves the computation of the worst case latency between tasks depending on their placement. If tasks u and v are placed on the same node, the communication cost between them is null (Equation 11a). On the other hand, if u and v are placed on different nodes, we have to consider the upload time of all data produced by u and the download time of all data required by v. The worst case being when the tasks produced by u and required by v are the last to be uploaded by u and the first to be downloaded by v (Figure 11).

In Equation 11, the first sum (Equation 11a) corresponds to the upload by task u of all pieces of data not required by v. The second line (Equation 11b) corresponds to the interlaced upload by task u and download by task v of all pieces of data produced by u and consumed by v. Finally, the last sum (Equation 11c) corresponds to the download by task v of all the required pieces of data produced by tasks other than u. All these are visible in Figure 11.

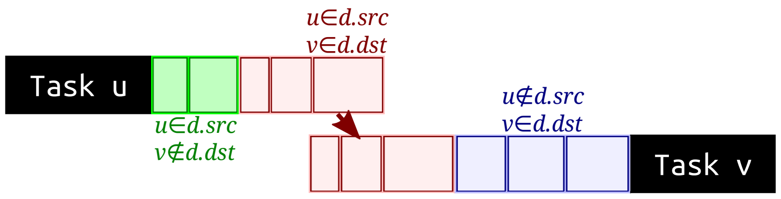


Figure 11: Preview of the communications between two tasks for a data-based workflow on a DaaS-based platform (see Equation 11).

In the third sum of Equation 11 (Equation 11c), it is not necessary to consider the download of data produced on the same node. Similarly, in the first sum (Equation 11a), we do not have to consider the upload of pieces of data which are consumed locally (all consumers are on the same node as the producer). The *islocal* predicate is computed as follows:

$$islocal_{alloc}(d, v) = \begin{cases} 1, & \text{if } alloc(d.src) = alloc(v) \\ 0, & \text{otherwise} \end{cases} \tag{Eq. 12a}$$

$$islocal_{alloc}(d) = \prod_{v \in d.dst} islocal_{alloc}(d, v) \tag{Eq. 12b}$$

This evaluation of the communication-induced dependencies between two tasks corresponds to a worst-case scenario. It is most likely that a specific ordering of the communications could give us better results but, as always, the construction of the clusters assumes a worst-case scenario to avoid providing nonsensical structures for placement.

In addition to the critical path computation itself, we also have to update the notion of valid schedule. Previously, a schedule σ only had to satisfy simple communication constraints (Equation 7). However, as we model the communications with more detail, the schedule also has to account for new constraints.

$$tl_{alloc}(v) = \max \begin{pmatrix} tl_{alloc}(u) + \omega(u) + c^{total}_{alloc}(u \to v) & \forall u \in Pred(v) \\ avail^{up}_{tl}(alloc(u)) + c^{total}_{alloc}(u \to v) & \forall u \in Pred(v) \\ avail^{down}_{tl}(alloc(v)) + c^{down}_{alloc}(v) & \\ avail^{cpu}_{tl}(alloc(v)) & \end{pmatrix} \tag{Eq. 13a}$$

$$bl_{alloc}(u) = \max \begin{pmatrix} \omega(u) + c^{total}_{alloc}(u \to v) + bl_{alloc}(v) & \forall v \in Succ(u) \\ \omega(u) + c^{total}_{alloc}(u \to v) + avail^{down}_{bl}(alloc(v)) & \forall v \in Succ(u) \\ \omega(u) + c^{up}_{alloc}(u) + avail^{up}_{bl}(alloc(u)) & \\ \omega(u) + avail^{cpu}_{bl}(alloc(u)) & \end{pmatrix} \tag{Eq. 13b}$$

with:

$$c^{up}_{alloc}(u) = \sum_{\substack{d \in data \\ u \in d.src \\ islocal_{\mathcal{C}}(d)=0}} \omega(d) \tag{Eq. 13c}$$

$$c^{down}_{alloc}(v) = \sum_{\substack{d \in data \\ v \in d.dst \\ islocal_{\mathcal{C}}(d,v)=0}} \omega(d) \tag{Eq. 13d}$$

$$\begin{aligned} \underset{alloc(u) \neq alloc(v)}{c^{total}_{alloc}(u \to v)} &= \sum_{\substack{d \in data \\ u \in d.src \\ v \notin d.dst \\ islocal_{\mathcal{C}}(d)=0}} \omega(d) + \sum_{\substack{d \in data \\ u \in d.src \\ v \in d.dst}} \omega(d) \\ &+ \max_{\substack{d \in data \\ u \in d.src \\ v \in d.dst}} \omega(d) + \sum_{\substack{d \in data \\ u \notin d.src \\ v \in d.dst \\ islocal_{\mathcal{C}}(d,v)=0}} \omega(d) \end{aligned} \tag{Eq. 13e}$$

When computing the top- and bottom- level for a task v, we not only look at the levels of the predecessors or successors of task v, but we also consider the CPU, uplink and downlink usage of the nodes that host v and the associated tasks (see Equation 13 and Figure 12). This is just an extension of the issue of independent tasks belonging to the same cluster which required the addition of implicit dependencies in DCP.

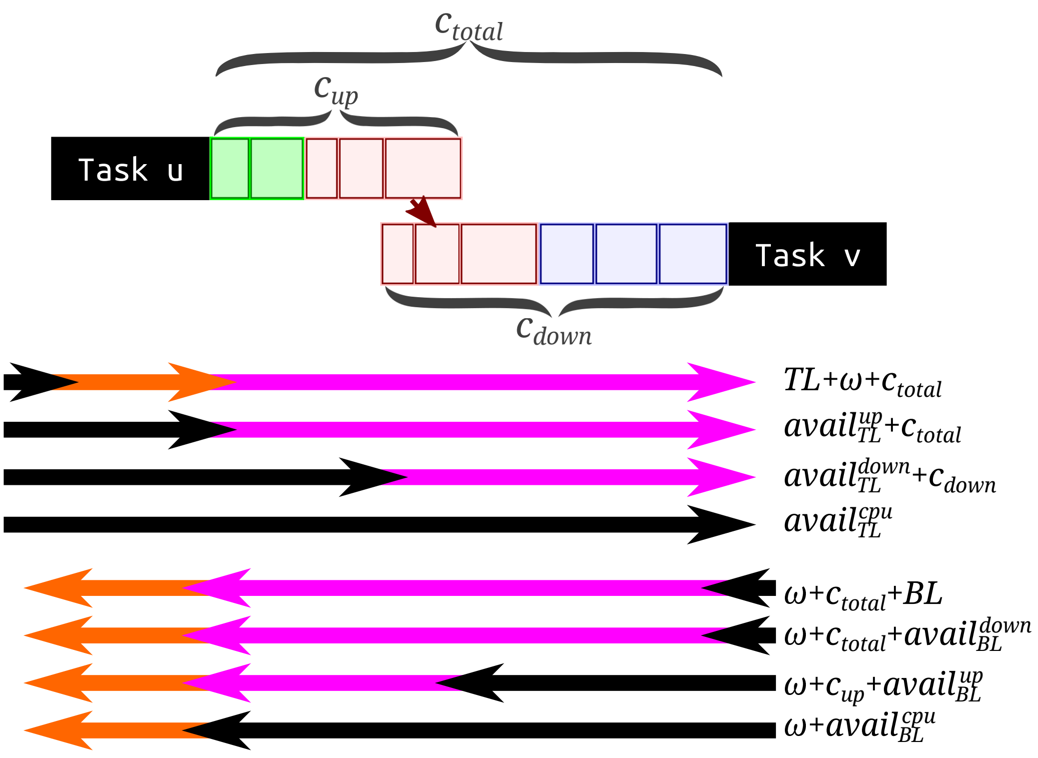

c_{total}
c_{up}
Task u
Task v
c_{down}
$TL+\omega+c_{total}$
$avail^{up}_{TL}+c_{total}$
$avail^{down}_{TL}+c_{down}$
$avail^{cpu}_{TL}$
$\omega+c_{total}+BL$
$\omega+c_{total}+avail^{down}_{BL}$
$\omega+c_{up}+avail^{up}_{BL}$
$\omega+avail^{cpu}_{BL}$

Figure 12: Schedule constraints for the DaaS-based network model.

As previously mentioned, we retained the structure of the DCP algorithm (Algorithm 1), to which we added our tailor-made formulas to compute the critical path. This gives us a generic task placement scheme which can deal with any DAG and which accounts for potential network congestion.

DAG	**Algorithm**	**#Nodes**	**Makespan** (t)	**Cost** (core×t)
Single Data Fork-join, as showcased in Figure 10b, with $n = 16$	One task per node	18	22.024	67.204
	Single node	1	18.000	18.000
	Legacy DCP	14	18.024	56.168
	DaaS-aware DCP	2	13.012	20.012
Multiple Data Fork-join, as showcased in Figure 10c, with $n = 16$	One task per node	18	37.024	82.204
	Single node	1	18.000	18.000
	Legacy DCP	14	33.803	70.156
	DaaS-aware DCP	5	14.000	26.048

Table 2: Cost and makespan details of different clustering policies for single-data and multi-data fork-join DAG. Gantt chart are shown in Figure 13

Section II.4

Results

In the previous sections, we described models for DaaS-based platforms and workflows as well as a variant of DCP that fits this model. In order to validate the relevance of this clustering algorithm to deploy DAGs on DaaS-based Cloud platforms, we are now going to compare it with a fully distributed scheme (all tasks are deployed on their own node, with no clustering) as well as with DCP. We will consider two fork-join cases which have the same description according to the legacy representation (see Figure 10) but for which the new representation can help make more adequate decisions. True values were obtained on a simulated Cloud platform using SimGrid which has been shown to efficiently model concurrency and link sharing in large-scale networks [Casanova et al., 2014].

II.4.1

Comparison of clustering heuristics on simulated infrastructures

Figure 13 shows both predicted and experimental Gantt charts obtained using different placement policies on different platforms. DCP leads to a very high level of communication parallelism to efficiently use many nodes and achieve a low makespan. However, using such a clustering on a simulated Cloud platform showed that congestion limits the actual data parallelism, leading to a much longer makespan.

We also see that results obtained using our model are very close to these simulated by SimGrid. This shows that our model makes adequate clustering decisions based on a realistic approximation, leading to good results. The lower level of parallelism given by our algorithm not only reduces communication-induced latency but also limits the number of nodes to deploy. Further results also show that the different data distribution patterns in single-data and multiple-data fork-join DAGs lead to relevant clustering decisions (see Table 2).

II.4.2

Economical outcomes

While all deployments of a DAG, regardless of clustering, correspond to the same tasks being executed and therefore to the same amount of core-hours used, changing the clustering can affect the deployment cost. It is important to note that the primary objective of static clustering algorithms such as DCP

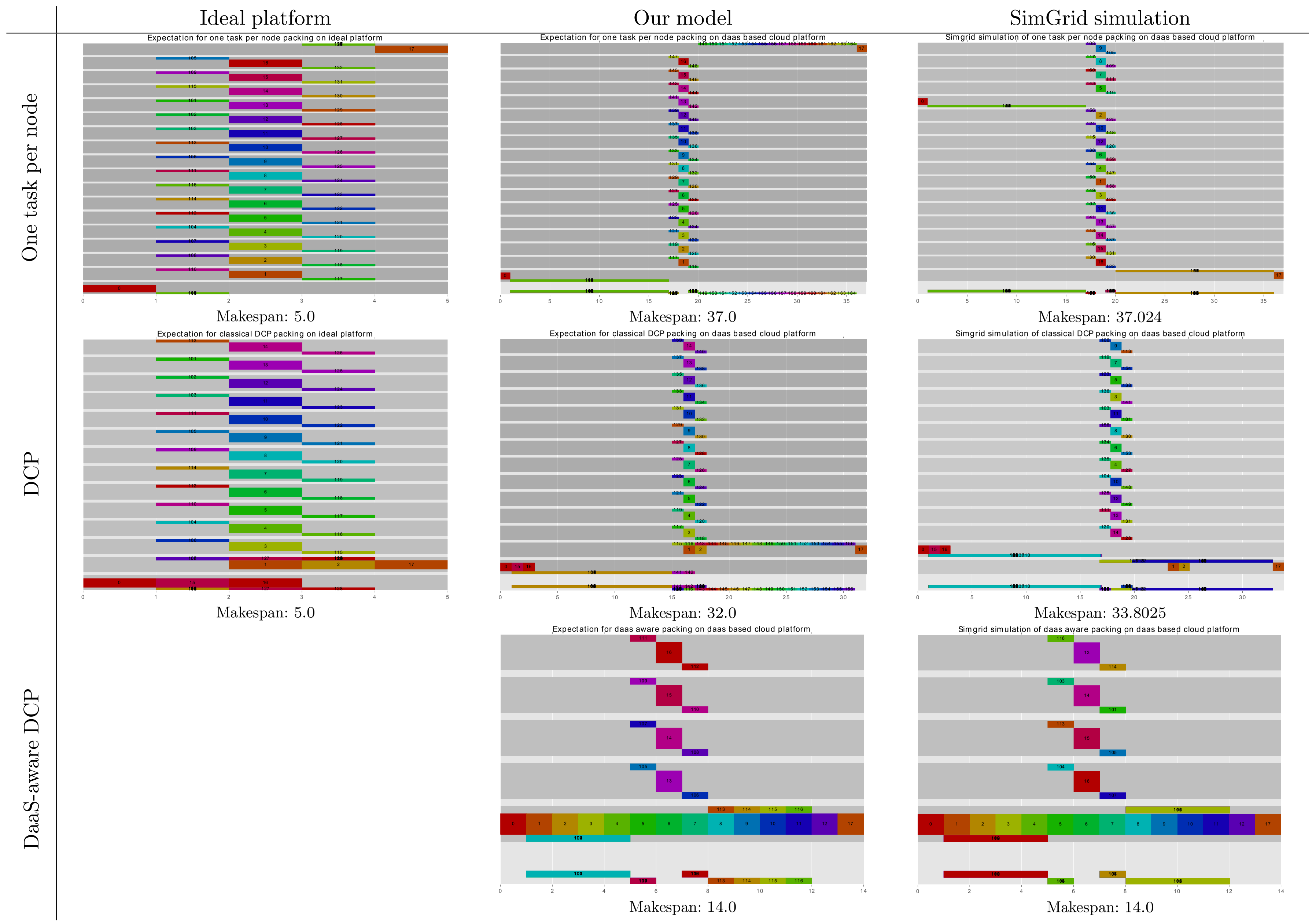


Figure 13: Comparison of the different clustering policies (Gantt charts and their associated makespan) for multi-data fork-join DAG ($n = 16$). For each sub-figure, the X axis represents time (arbitrary units). Grey rows represent cores. Boxes in these rows are, from top to bottom, downlink utilisation, tasks execution and uplink utilisation.

is to reduce the global makespan, assuming unlimited resources. In a Cloud context, this translates to an assignment of tasks to nodes, with a potentially considerable number of nodes deployed. In a context where nodes are billed proportionally to the number of core-hours used, we could hope that, as the same amount of computation is achieved, the cost would roughly be the same, with any difference being caused by constraints on the reservation time (e.g. at least one hour).

However, these assumptions do not take into account the time during which a node is retrieving data before running a task or sending data after having run a task. During this time, we have to pay for the node even though we do not use its computing potential. Avoiding network congestion and ensuring smooth data transfers is a way of limiting this waste of CPU time. Table 2 shows the number of nodes used, the makespan, and the total deployment cost for our two models of fork-join DAGs using different scheduling algorithms. It is visible that, in addition to reducing the makespan of the DAG, using the right clustering algorithm can help reduce deployment costs.

For our experiments, we used synthetic fork-join workflows where the computation time of each task was equivalent to the transfer time of one piece of data from/to the DaaS using the full bandwidth of the node. This ratio between computation and transfer times highlights congestion issues. In this case, single-node clustering achieves good results, as parallelism rapidly causes network congestion. With other ratios, we observe similar behaviours; the DaaS-aware DCP being the fastest of all, only beaten on the cost aspect by the very slow single-node approach.

Section II.5 Discussions

II.5.1 Heterogeneous platforms

While our DaaS-based DCP clustering yields interesting results for the scheduling of workflows on homogeneous elastic platforms, it does not handle heterogeneity. The issue of task clustering for data-locality purposes plays against the logic of placing each task on the type of machine that most closely fit their needs in a heterogeneous environment. If task u runs faster on an instance type t_1, if task v runs faster on an instance t_2 and if u and v exchange a lot a data, then compromises have to be made.

The HEFT approach is interesting when dealing with fixed size platform. I hoped to build on top of HEFT, similarly to what I did with DCP, but unfortunately HEFT does not match the Cloud paradigm. This is because HEFT is not an edge-zeroing approach but rather a list-scheduling heuristic. Each task, in order of priority, is placed on the node where it would finish the earliest, given previous task placement, data transfers and machine specifications. In an elastic Cloud environment, this placement decision would have to consider both the already instantiated machines, but also potential new machines that could be deployed on demand. This would include every machine type available in the Cloud environment.

While this is not necessarily ill-formed, the environment is such that, at the time of writing this, instances types are different fractions of the same hardware. Moving from a S to an M to a L instance, we see an increase in computing power with the only drawback being the deployment cost. A scheduling approach should consider cost saving as part of its objectives otherwise it would always make sense (from a scheduling point of view) to use big instances over smaller ones. This falls back to a homogenous approach with only big instances, which DaaS-aware DCP solves.

While this is a negative result, other approaches are possible. I believe a possible solution would be to start by performing the clustering using a homogeneous model of large machines and then perform a relaxation to try to reduce the size of the machines used to run non-critical clusters. This is definitely material for future work.

II.5.2 Toward scheduling of the clusters

The whole DaaS-based DCP clustering is a pre-processing, that can be performed ahead of schedule. It requires some details about the platform, but, as it is independent of the current workload, it remains valid for further executions on the same platform.

The next question in our way to an autonomic workflow engine is "what to do with the clustered workflows?" The clustering solves the data-locality, but we still need a placement mechanism. We want this placement mechanism to enable a multi-tenant approach, so it will have to handle many clusters belonging to many independent workflows. We also need a mechanism to deploy the Cloud resources autonomously and efficiently. This is the topic of Chapter III.

For both of these mechanisms to perform, we use the information computed during the clustering step to build metadata for the clusters. In particular, the top- and bottom- levels provide precious incite on the cluster execution window. A user might submit a DAG G at time t_0 and assign a deadline t_1 at which they expect the results. The DaaS-based DCP will provide a clustering $\mathcal{C}$ and the associated tl and bl. We then deduce the minimum makespan MS[16] from these levels. A deadline t_1 is only realistic if $t_0 + MS \le t_1$. The actual deadline is therefore:

$$deadline = \max(t_1, t_0 + MS) \qquad \text{(Eq. 14)}$$

For each cluster, we evaluate a set of metrics that will help the scheduling and platform allocation. The most important ones are the expected earliest execution time (ASAP) and the latest execution time (ALAP) for the cluster:

$$ASAP(p \in \mathcal{C}) = \min_{alloc(v)=p} (t_0 + tl(v)) \qquad \text{(Eq. 15a)}$$

$$ALAP(p \in \mathcal{C}) = \min_{alloc(v)=p} (deadline - bl(v)) \qquad \text{(Eq. 15b)}$$

Users willing to get the results as soon as possible can set $t_1 = t_0$ and rely on the middleware to do as best as possible. On the other hand, jobs can be set in a best-effort mode by selecting $t_1 = +\infty$. Such jobs will never become urgent. The scheduler will grant them time slots when possible, but the platform deployment mechanism will not consider them when evaluating the need for new resources.

[16] $MS = \max(tl(v) + bl(v))$

Chapter III

Autonomic scheduling and platform management

In this chapter, we discuss the design of our autonomic scheduler and platform manager which will complement and build upon the work described in Chapter II. Thus, it completes the design of the autonomous workflow engine we designed in Chapter I.

After Chapter II, which focused on the offline analysis of workflows and the pre-computation of task clusters (to solve the data locality issue), this chapter focuses on the real-time part of the workflow engine. The goal here is to achieve efficient platform management and scheduling of the task clusters produced in Chapter II.

Section III.1

The need for autonomic decision making

Designing a workflow engine, our objective is to build a framework of a multi-tenant computing platform which unites two aspects of scientific computing: the description of jobs using workflows and the usage of elastic Cloud resources. These models for job definition and virtual resources provisioning can be leveraged to provide improved middleware capability and better user experience.

The representation of jobs as structured workflow rather than black boxes is paradigm shifting. Rather than running jobs as monolithic tasks, the middleware can parallelise the execution in an application agnostic manner. Parallelism is deduced solely from the workflow structure. Knowledge of the application details is no longer required to perform efficient parallelism, and generic solutions can be built.

Similarly, the Cloud paradigm provides new levers for the efficient use of computing resources. While it can be used to deploy any platform, including clones of old models, the Cloud paradigm proposes an innovative framework for building elastic computing platforms that fit the workload better.

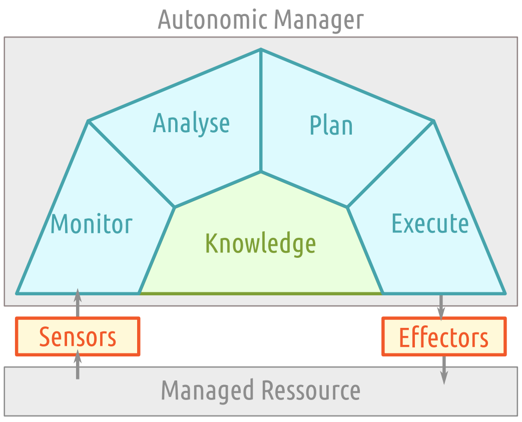


Figure 14: Overview of the MAPE-K architecture.

Still, the advantages of these representations are only relevant if adequate decisions are made. This decision-making problem is already well recognised for the execution of workflows. Correct execution requires the scheduler to account for dependencies between tasks and schedule them in accordance with the dependencies. Keeping track of the already executed tasks and with knowledge of the workflow structure, the scheduler decides which tasks to run next. This mechanism is an autonomic control loop in disguise.

Autonomic control (contraction of autonomous and automatic) is a concept that focuses on the use of feedback loops for the self-management of systems. The MAPE-K approach (Figure 14) defines a set of components that, together, form a closed loop: Monitoring, Analysis, Planning, Execution and Knowledge. According to this architecture, the combination of sensors (for self-monitoring) and effectors (for self-adjustment), associated to a known model of the system, makes the system self-manageable.

In a self-managed system, human operators are no longer in charge of taking the relevant decisions. Instead, they are in charge of designing policies which, when enacted by the effectors, allow the system to stay inside a given working envelope. IBM [McBride et al., 2009] distinguishes many properties as being part of self-management: self-configuration, self-healing and self-optimisation.

Building an autonomous workflows engine goes much further than just scheduling the tasks within a workflow. We also want to apply the same autonomic approach to the issue of platform deployment. This capability, which is still outside the scope of many middleware, would allow better integration with Cloud computing infrastructures. This would result in better platform scaling, providing a better quality of service at a lower cost. It would also remove the burden of platform management for human operators.

Current usage of Cloud resources mostly relies on technicians or engineers allocating more resources when required by the users and removing resources when asked by the billing department. Having an autonomic loop in charge of platform deployment would increase the reactivity in both cases. Users would feel as if resources are always available; the resources would only be deployed and paid for if there is a real need and technicians would be free to focus on other issues.

Moreover, our intuition is that rather than having two independent control loops, the fact of connecting workflow scheduling and platform management, and allowing them to share information, should help in making efficient control decisions. More particularly, some events in the scheduling loop should act as good indicators on the status of the platform and could benefit the deployment loop.

Section III.2 A multi-layer scheduling approach

As seen in Chapter II, DAG scheduling is a well-known topic. Many middleware with workflow support [Caron et al., 2002, Deelman et al., 2005] implement dependency tracking mechanisms to deploy the underlying tasks correctly and use various algorithms to achieve efficient deployment parallelism. Still, many approaches can be used to improve task placement. One of the critical factors is data-locality, which can be improved by forcing the colocalisation of tasks with data dependencies, or by duplicating tasks that produce large datasets. While minimalistic workflow scheduling is easy, achieving efficient scheduling and placement, in accordance with the workflow structure, is a multi-criteria optimisation problem.

One of the challenging goals we set is to have a multi-tenant approach. We want multiple users to share the computing platform, each one of them being free to ask for the execution of any workflow at any time. A direct consequence is that the workload can change without prior notice, thus voiding any prior plans. This refrains us from relying on a compute-intensive optimisation approach. Each time the situation changes due to an isolated user action, recomputing an expensive algorithm (which would consider the entire workload) would infringe on the scalability of the platform. Chapter II was the first consequence of this choice: A static clustering of workflow, while not optimal, allows us to achieve a certain degree of data-locality optimisation during the pre-processing of the workflow.

Unfortunately, the benefits of task clustering also come with a cost. Rather than scheduling tasks we have to work with clusters that are more complex objects and which dependency structure can contain cycles. This more layered vision of workflows, organised around the task clusters rather than simple tasks, requires a similarly layered approach to scheduling.

III.2.1 High level description

As described in Chapter II, clusters of tasks are designed such that the tasks they contain are to run on a single machine to achieve good data locality. The mapping of the clusters to the nodes must therefore consider the constraints of the whole cluster and anticipate the execution of all the tasks it contains. After the mapping of the clusters onto the nodes has been performed, the nodes will be in charge of the scheduling of the tasks in the clusters they were assigned.

This vision produces a two-level scheduling mechanism:

- The core scheduler is responsible for managing the unassigned clusters of tasks and for distributing them to available nodes.
- The node schedulers are responsible for managing and executing, at the node level, the tasks in the clusters that the code scheduler assigned them.

This multi-level design comes with scalability improvement. The use of node schedulers contributes to decentralising the scheduling mechanism, hence reducing the load on the centralised core scheduler and increasing the scalability of the platform.

In our approach, the design of these agents resolves around the daemon concept. They are entities providing interfaces to services and able of notifying one another. These notifications are responsible for the coordination between agents and the proper working of the platform as a whole. An overview of the interactions between agents in this multi-layer architecture is visible in Section III.2.1.

III.2.1.A *Node schedulers*

The node scheduler is the agent responsible for the management of a node. Nodes represent the overwhelming majority of actors in the platform. Their role is to execute the tasks (grouped in clusters) submitted to the platform and assigned to them by the core scheduler. Each node contains a local memory space to hold input and output files used and produced by the different tasks. Therefore, in addition to a task queue, each node also runs agents managing the download and upload of the relevant pieces of data.

When available, nodes request work from the scheduler. If pending work is available, the scheduler will assign a cluster of tasks to the requesting node. Otherwise, the node will suffer from work shortage. Work shortage is a marker of a potentially overscaled platform. This information, produced by the scheduling loop will later help in the event-based monitoring of the platform by the deployment loop (see Section III.3).

Unlike the core scheduler which has a global view of the available resources in the platform, the node scheduler has a local vision and takes scheduling actions at a local level. In addition to the computing resources (CPU / GPU), this local vision also focuses on data exchanges.

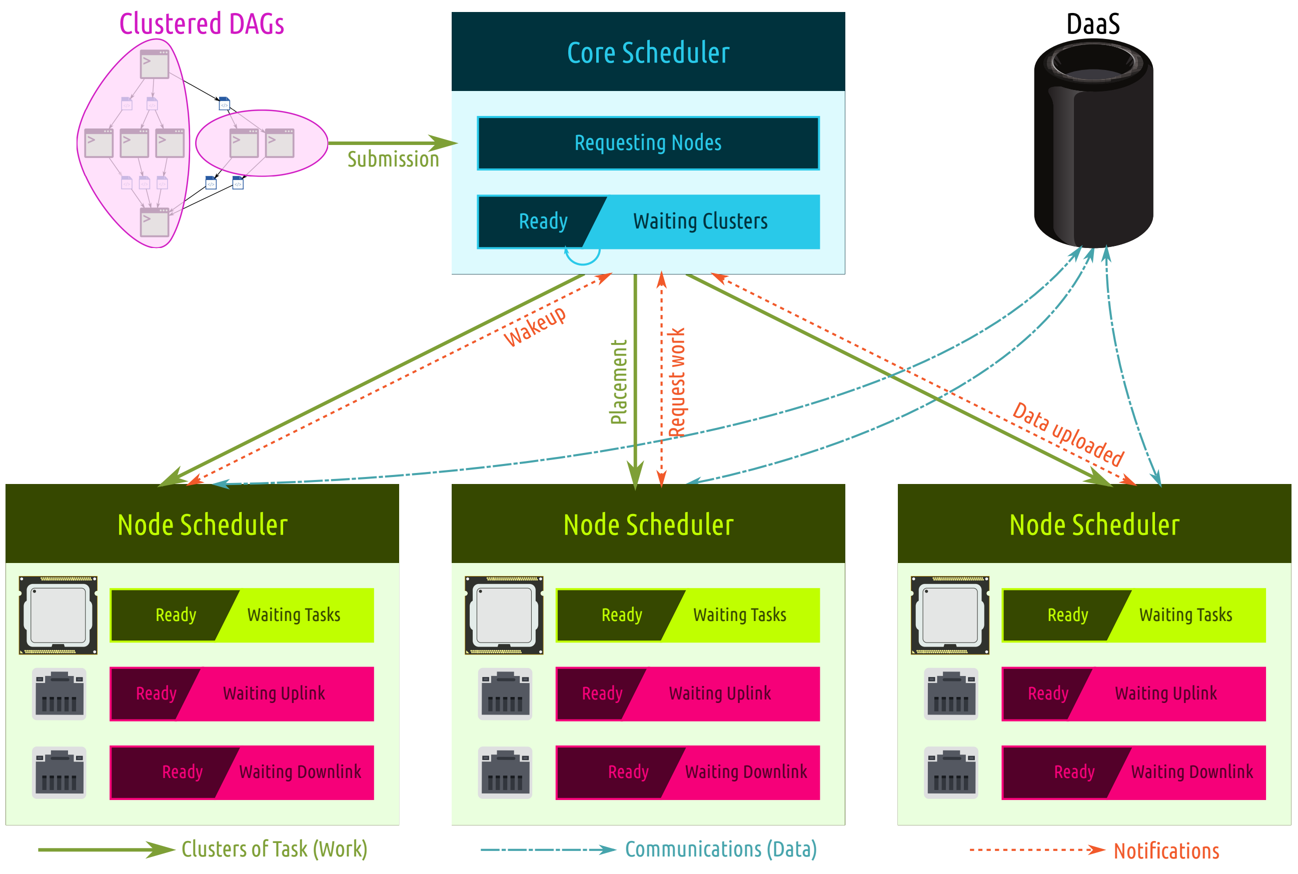


Figure 15: Overview of the interactions between agents in the multi-layer scheduling loop.

Data notification

In a DaaS-based Cloud infrastructure, input and output data have to transit to and from the DaaS. The nodes are responsible for these transfers as they are the ones initiating the communications. Therefore, in addition to a task queue feeding their CPU/GPU, each node has an uplink and a downlink queue. These queues contain identifiers of the pieces of data that will be exchanged with the DaaS. Entries in these queues can be considered as communication tasks. While compute tasks are performed by the computing power of the node, communication tasks are performed by the networking resources, which themselves are subdivided into uplink and downlink. Similarly to work shortage causing a node to request work from the core scheduler, downlink availability will cause the node to advertise its availability. This partial decoupling of communications and execution helps prefetching datasets and feeding work to the nodes, leading to an improved platform usage. Depending on the DaaS implementation, nodes would either have to subscribe to notifications for data availability or rely on the core scheduler to perform this role.

Let's imagine a task t_1 running on node n_1 that requires a piece of data d produced by a task t_2 on node n_2. If no other piece of data is required by tasks on node n_1, and if d is still unavailable, the downlink manager on n_1 will fall asleep due to lack of work. As its downlink is available, n_1 would have advertised the availability to the core scheduler, but given its task queue might be filled, the core scheduler is unlikely to assign it any more work for now. Fortunately, t_2 is running on n_2, which leads to the availability of d on n_2. Thanks to meta information coming with the DAG, n_2 knows that d is required by t_1, and, as t_1 is not in its work queue, it knows that d should be uploaded to the DaaS. At this point, either n_1 has a way to subscribe to d, and will wake up on its own, or, most likely, n_2 will have to notify the core scheduler of d's availability on the DaaS. The core scheduler, knowing which node t_1 was assigned to, is capable of notifying n_1 which will then start downloading d from the DaaS.

Whereas this whole process might sound complicated and expensive, it is a simple notification of task finalisation. The difference being that, rather than notifying the end of the compute tasks, we notify the end of the communication tasks. Ultimately, it is the availability of data on the DaaS that determines when a task (and the corresponding cluster) becomes available. This process is only performed for pieces of data that are shared across clusters. As the clusters are designed around the concept of data locality, we can expect a load reduction over what would happen in a more traditional workflow engine.

III.2.1.B — *Core scheduler*

The core scheduler is responsible for the centralised vision of the workload and the distribution of clusters of task to the nodes. It keeps track of all clusters waiting to be deployed and execute the logic that places them on the nodes which have the resources required to execute them before their respective deadlines. A cluster can either be ready, if the data required to start the execution of at least one of its tasks is available, or waiting. Once a cluster has been scheduled onto a node, the scheduler keeps track of the cluster to nodes mapping. This mapping will later be used to forward notifications relative to the tasks they were assigned.

The core scheduler also keeps track of nodes which have required work. Nodes will come to request work (see Section III.2.1.A), and the code scheduler will then be in charge of feeding them clusters of task to compute. We discuss the details of the algorithms in charge of this assignment in Section III.4.2.A.

In a nutshell, the core scheduler has the knowledge of the pending workload and the available computing resources. This knowledge puts it in charge of the high-level scheduling. This scheduling is constrained by the quality-of-service requirement (expressed in the clusters) and the resources availability (advertised by the nodes themselves). The quality of the meta-data generated during the static analysis step is essential for the core scheduler to make the adequate decisions.

Cluster status update

As discussed previously, we need a mechanism to keep track of data availability on the DaaS. Unless the DaaS provide such a service, it is the role of the core scheduler to dispatch notifications coming from the nodes. In addition to notifying the nodes that data they require is now available, the core scheduler also uses this availability information to keep track of cluster status. As shown in Section III.2.1, the core scheduler keeps track of both “ready” and “waiting” clusters. Meta information about the expected time at which a cluster should transition from the waiting to the ready state is computed during the static analysis. Still, it is the availability of data required to compute the underlying tasks that triggers this state transition. Because of the internal dependencies between the tasks in a cluster, we cannot expect all tasks in a cluster to be ready before placing the cluster onto a node. In fact, most of the tasks in a ready cluster are likely to require still unavailable data. It is the execution of the first tasks in a cluster that will produce, locally, the data required by the following ones.

So as to account for clusters status update, the core scheduler should keep track of a few mapping. First, it needs the task to cluster mapping, which has been computed by the static analysis. Then, it needs to keep track of the number of unsatisfied data dependencies for each task. This mapping is initialized with the number of data objects required by the task (specified in the DAG), and is updated each time some piece of data is made available on the DaaS. Last but not least, the core scheduler has to keep track of which node it placed the clusters on.

Upon finishing a task, the node that computed it will have a look at the pieces of data it produced. If they are required by other “child” tasks, further consideration is required:

If the piece of data is required by local tasks (tasks in the node’s waiting queue), a local notification is required. The local vision of the task is updated, and if all dependencies are now available the task transitions from the waiting to the ready state.

If the piece of data is required by tasks that are not in this node’s queue, then a global notification is required. The node starts by scheduling an uplink job which will be in charge of uploading the data to the DaaS. The uplink manager then takes over so that the task manager can move on to another task. Once the uplink manager has finished uploading the data to the DaaS, and unless the DaaS provides a notification mechanism, the node will notify the core scheduler of data availability. At this point, the core scheduler will have to go through all tasks which require this piece of data, and check whether they are in placed clusters or not.

For tasks in already placed clusters, the core scheduler will notify their node so the downlink manager can schedule a download.

For tasks in yet unscheduled clusters, the core scheduler updates their unsatisfied dependencies mapping. If this counter reaches zero (meaning all dependencies for this task are now available) then the task is ready. If the task’s cluster is still waiting, the core scheduler updates it to ready so as to authorise its placement.

III.2.2 Challenges and subtleties

The innovative nature of the multi-layer scheduling mechanism described here above leads to many new and unforeseen behaviours. In particular, the use of asynchronous agents, in conjunction with the complex dependencies between clusters, tasks and pieces of data, requires adequate metrics and fine-tuning to avoid improper decision making. Hiccups in this automation can lead to synchronisation issues ranging from non-compliance with the desired QoS to deadlocks of the whole system.

III.2.2.A *Asynchronous agents and shared memory*

The distribution of the scheduling, which can be qualified as both inter-nodes (core scheduler vs node schedulers) and intra-node (task manager vs uplink manager vs downlink manager inside a node scheduler), can lead to many synchronisation issues. Using remote procedure calls to access unprotected memory is dangerous as it can lead to data-races. In the meantime, protecting the memory access with critical sections solves the data-race issue but can lead to deadlocks if the calls are synchronous.

This issue was solved through two system-wide policies:

- Synchronous calls should only be used to read memory or to raise flags that will then be processed by an iteration of the main thread loop on the receiving side.
- Communication between agents should be asynchronous. Actors requiring action from someone else should notify them using the wake-up mechanism, move on to do other work or go to sleep and wait to be notified when further action is required.

These policies are applied to inter- and intra-node distribution. Implementation details are given in Chapter IV.

III.2.2.B *Priorities in the node and cluster queues*

Thanks to the static analysis and the construction of the clusters of task, the issue of data locality is not an issue that the multi-layer scheduling engine has to deal with. Still, respecting the QoS required by the user and translated into cluster-level requirement demands that the nodes have the required resources available, to prefetch the data and execute the tasks before their respective deadlines.

Despite the need for proper network resources availability, we order both the clusters and the requesting nodes queues according to their requirement and availability regarding computing power.

The queue of requesting nodes is sorted by the availability of their CPU. If a node has an empty task queue, this value is set accordingly. On the other hand, nodes with tasks in their task queue advertise the expected availability based on the expected duration of the remaining tasks. Even if the queue is not empty, requesting work can make sense if downlink resources are available or if all tasks in the queue are locked (see Section III.2.2.C). This ordering policy sorts the available resources in a least-used-first approach. The first ones in the queue are these that have either been waiting for longer, or those that will be available sooner. Providing multiple nodes can run a cluster, deciding which one to use has consequences both for the execution of other workflows and for the evolution of the platform deployment. Placement options are discussed in Section III.4.2.A.

The queue of ready clusters is sorted according to the latest they can be started without breaking the constraints computed during the static analysis. Using this ordering, the position of clusters in the queue corresponding to the urgency of their deployment to a node. If some nodes are late and imperil the correct execution of the workflow they are part of, they will be at the beginning of the queue and can be easily identified by a deadline lower than the current time. Similarly, the deadline of the first cluster in the queue gives an idea of the timeframe before any cluster becomes urgent. The ordering plays a role not only in the scheduling of clusters but also in the autonomic deployment loop discussed in Section III.3. Details about the use of this ordering by the placement algorithms are shown in Section III.4.2.A.

III.2.2.C *"Locked" nodes*

The locked node issue is particular to the multi-layer scheduling engine. More precisely, it is an issue with dependency cycles between clusters of tasks. It can cause deadlocks when the engine tries to schedule weakly constrained workflow (workflow with remote deadlines).

When building the clusters, the initial goal is to determine the highest level of parallelism which can be achieved without running into network-induced constraints. The goal is to minimise the possible turnaround time of a workflow. In a second time, the user's requirements are used to relax the constraint on the clusters, giving more flexibility to the core scheduler. Yet, this flexibility can in some cases become an issue.

Let's imagine a workflow (Figure 16) with four tasks t_1, t_2, t_3 and t_4. t_1 produces a large piece of data d_{l1} which is required by t_3 and a small piece of data d_{s1} required by t_4. Similarly, t_2 produces a large piece of data d_{l2} which is required by t_4 and a small piece of data d_{s2} required by t_3. In order to avoid the latency caused by the transfer of the large pieces of data d_{l1} and d_{l2}, the clustering will end up being:

$$\{c_1 = \{t_1, t_3\}, c_2 = \{t_2, t_4\}\}$$

As both t_1 and t_2 have no input dependencies, both clusters are ready from the start. If the required deadline is short, both cluster will be assigned to different nodes and everything will be fine. On the other hand, if the deadline required by the user is very far, we can imagine that one single node could be enough to execute the whole workflow sequentially.

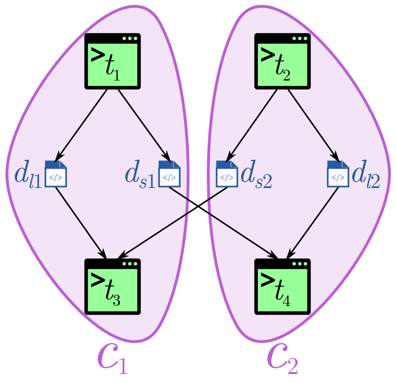


Figure 16: Example of workflow that can result in "locked" nodes

Let's now assume we have one single node available. The core scheduler will place c_1 on the node and the execution of t_1 will start. After the execution of t_1, d_l1 will be kept locally and d_s1 will be sent to the DaaS. We are now facing a situation where neither the task queue nor the downlink queue are empty. The task queue contains t_3 and the downlink queue contains d_s2. The node could be considered as busy but cannot perform any work until c_2 is assigned to a node and t_2 is executed. This is what we call a "locked" node. No other node is available, but the constraints are such that the locked node could run c_2 and achieve the required deadline.

The solution to this deadlock is to have the locked nodes, with queues containing only unavailable task, request work. Any new cluster of tasks assigned to a locked node has lower priority over the waiting tasks, but will be able to use available resources following an opportunistic behaviour. Thanks to the use of predictive availability metrics, the core scheduler can still assign work to the locked nodes and thus unlock the situation. The critical point here is that nodes must be able to require work whenever any local resource is unused, even if the associated queue is not empty. The core scheduler then considers future availability to place cluster opportunistically. Failure to do so can result in deadlocks.

Section III.3

An autonomous deployment mechanism

Just like task scheduling, resource management can be seen as a control issue. This section focuses on the description of an autonomic loop to solve the issue of elastic Cloud platforms management in conjonction with the multi-layer scheduling engine.

III.3.1

Deployment planning

According to the MAPE-K model, building an autonomic deployment manager requires the construction of dedicated components for monitoring, analysis, planning and execution. These components are usually designed around a mathematical model that is simple enough to work with, yet catches the complexity of the input parameters. Such model commonly relies on differential equations to capture the essence of the retro-actions.

In our case, it feels natural to start the design of the autonomic loop with a description of both the workload and the platform. Both of these factors affect the execution, and will subsequently be modified when performing the deployment plan.

However, unlike previous work on autonomic platform adaptation [Zhou et al., 2016], we are here facing a very complex environment. Tasks (clusters) are not fungible in that they have different durations, start time and deadlines. Aggregating them in single metric seems vain. The platform is also harder to model compared to previous work. We are not dealing with servers providing transactional services, with a treatment capability of n requests per minute. Instead, we are dealing with machines that have a work queue that might not be empty. Nodes are therefore non-fungible and must be described individually.

Considering an environment containing clusters with different durations, earliest starts and deadlines, as well as nodes with different availability estimation, proves tricky. It is far from the mathematical representations used in previous autonomic loops. In particular, quantifying the retro-action (how would the system be affected by an additional node) requires a vision of the dependency between clusters and a conjoint representation of all the control loops (including the multi-layer scheduling) acting on our platform.

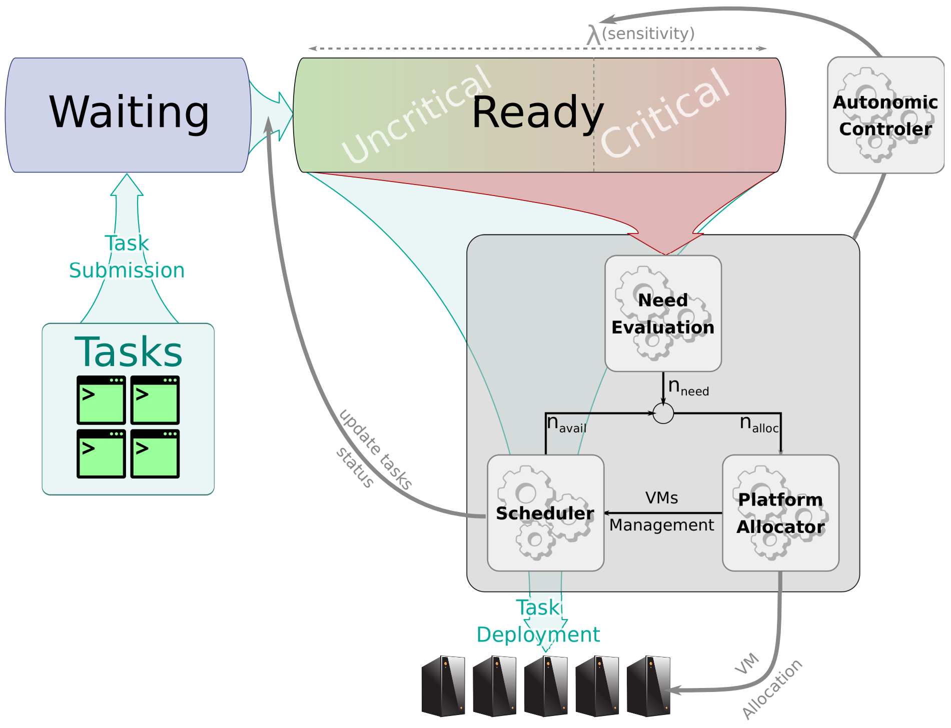


Figure 17: Initial design for an autonomic deployment loop.

A tipical approach is to use an estimator to determine how many resources are required to perform the ready clusters'. Comparing this value to the number of available resources (Figure 17) provides insite on the suitability of the current deployment.

In our case, the difficulty posed by the irregularity of workflows and the fact that we must consider the busyness of existing nodes pushed us toward a need evaluation component that considers both the workload and the current platform (Figure 18). The question that remains is "how to perform this need evaluation?"

Our solution is based on an analogy between compute tasks and fluids. Clusters of tasks are similar to time-constrained fluid incomes and computing resources are similar to constant-flow sinks. This analogy helped us build block placement heuristics that are used to perform the need evaluation through a simulation of the platform of the next clusters. Depending on the complexity of the heuristics used in this simulation we can manage a balance between accuracy of the prediction and reactivity of the system.

Section III.4.2.C describes the list scheduling algorithm used for the need evaluation. Using a simple approach we managed to achieve good reactivity with a simulation time significantly lower than 0.1 second in all the runs performed as part of the simulations presented in Section III.4.3.

Similarly to the placement algorithm used by the core scheduler, the loop design is modular, and the simulation algorithm (described in Section III.4.2.C) used for need evaluation could be the subject of further studies and optimisation. In particular, it would be interesting to see how the similarity between the need evaluation algorithm and the actual scheduling loop affect the deployment efficiency.

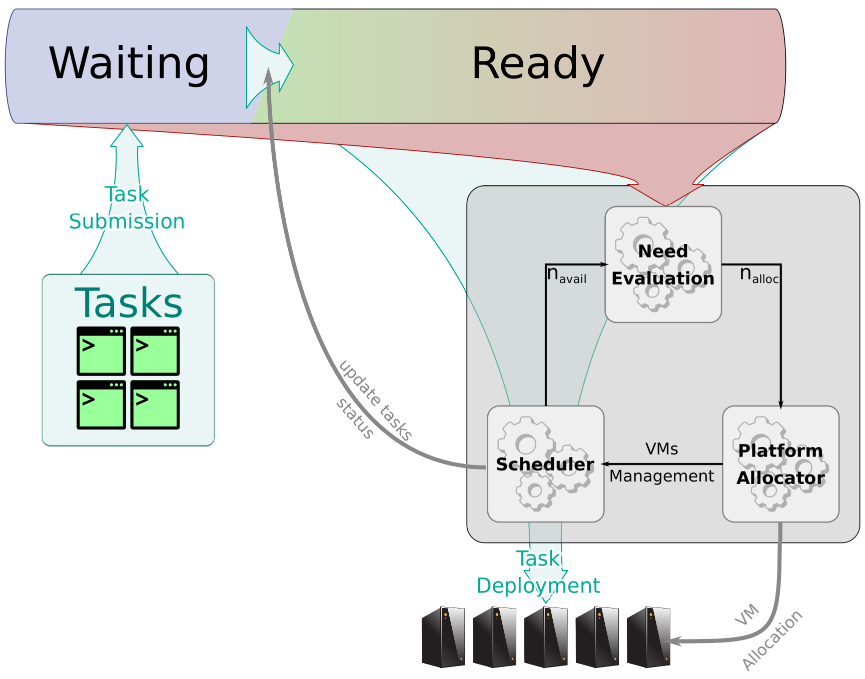


Figure 18: Updated design for an autonomic deployment loop.

III.3.2 A platform manager connected with the scheduling engine

Rather than having a single loop controlling the platform deployment, which could very well be a viable solution, our design uses two such loops. One is a decentralized loop which is responsible for the down-scale adaptation of the platform. The other one is a centralized loop, which requires more knowledge to control the up-scale adaptation of the platform.

Both deployment loops are autonomic systems that have to monitor the entity they manage. This is where the deployment control loops meet the scheduling control loop. The multi-layer scheduling engine is distributed and is aware of the workload distribution across the platform. It is the first one to notice misfit of the platform. Rather than building sensors for the deployment loops, we bind it to the scheduling engine. We use the notifications generated during scheduling as triggers, so the deployment loops follow the same asynchronous logic as the scheduling loop.

In addition to the agents involved in the workflow scheduling loop (core and node schedulers), we added a “Deployer” agent responsible for the analysis, planning and execution for the up-scaling control loop. The organisation of our platform is described in Figure 19.

III.3.2.A *Evolution of the node schedulers*

Nodes represent the overwhelming majority of actors in the platform. As discussed previously, their role is to execute the tasks (grouped in clusters) submitted to the platform. When available, nodes request work from the scheduler. If no pending work is available, the node will suffer from work shortage.

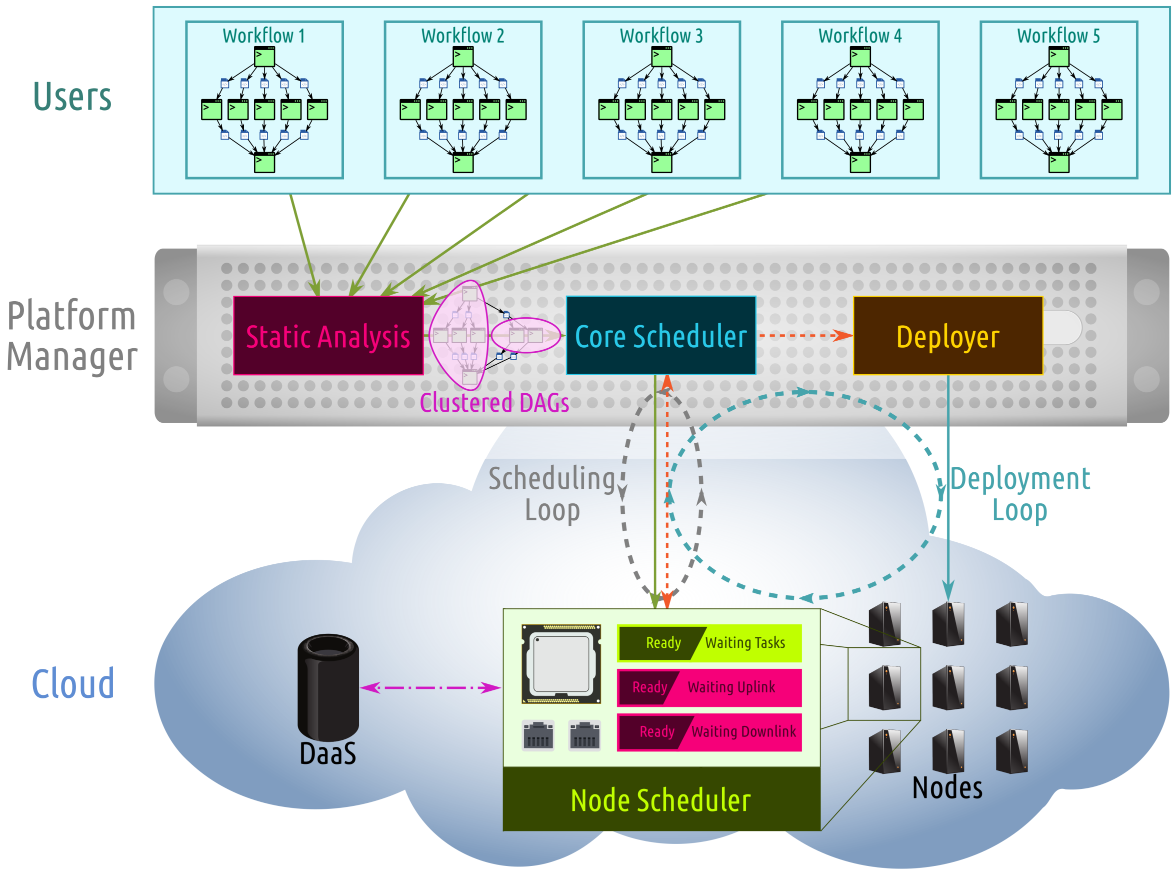


Figure 19: Autonomic scheduling and allocation loops.

From a deployment point of view, having nodes starving is an unsatisfactory situation. Keeping them alive has a cost, which is wasted if the nodes are not used during this timeframe. Thus, unless instructed otherwise, unused nodes should decide to shut down by themselves to save on deployment cost. Therefore, each time a node is unused it does not only ask the core scheduler for work, but also initialises a “suicide timer” deciding when to shut down if no work is received by then.

The node determines the suicide timing following the Cloud provider billing policy. In the case of instances billed on an hourly basis, the node will plan to shut down just before the start of the next hour. Staying up until then does not add any cost which was not already committed. However, staying up after that commits to the payment of an additional hour. Staying up only makes sense if there is work to perform during this extra hour.

This new behaviour is key to the down-scaling of the platform, and it is, in fact, an autonomic loop which is circumscribed to the node. Thanks to workload monitoring and analysis, each node can take action to deallocate itself and therefore modify the platform allocation to fit the global requirement better. On the other hand, the up-scaling mechanism relies on a dedicated agent, the deployer, which is described in Section III.3.2.B.

III.3.2.B — *A new agent: the deployer*

The deployer is a new entity responsible for the up-scale adaptation of the platform. We already saw that the nodes “commit suicide” when they are under-used. The down-scale adaptation of the platform is therefore decentralised. However, we still lack a mechanism to handle the allocation of new nodes.

The deployer manages the allocation of new VMs according to a schedule. This schedule is computed by the deployer, and updated upon notification by the core scheduler of relevant changes in the workload. Construction of this schedule requires the knowledge of the availability of nodes as well as the list of queued clusters, with their deadlines and predicted readiness. It also requires platform details such as the time required to boot a new node. With this knowledge, several algorithms can be used to evaluate the needs and compute a deployment schedule accordingly.

The deployer is responsible for monitoring, analysing, planning, and execution of a control loop which is responsible of allocating new nodes when required.

- **The monitoring step** gathers information about the current workload and the current status of each node;
- **The analysis step** performs a minimalistic list scheduling to evaluate the needs and potential lack of resources;
- **The planning step** uses the schedule produced during the analysis step to decide when to allocate new nodes;
- **The execution step** allocates new nodes following the schedule.

The cost and accuracy of this mechanism depend on the complexity of the scheduling performed during the analysis. This part of the framework is modular, and we already have different policies which impact the decisions taken, and therefore on the efficiency of the control loop. We consider that this scheduling algorithm should follow a logic similar to that of the scheduling control loop. However, implementing a different policy could also be interesting as it could make decisions that might be helpful to modify the behaviour of the scheduling loop. With simple heuristics we already achieve good results as shown in Section III.4.3, yet future work could investigate different scheduling methods for the analysis step.

III.3.3 Unfolding the platform adaptation mechanisms

The logics of both the down-scale and the up-scale mechanisms are shown in Figure 20.

Down-scale adaptation mechanism

When the platform is over-provisioned according to the current workload, nodes are executing tasks faster than new tasks are being submitted. The resulting work shortage causes some nodes, which have requested work from the core scheduler and have initialised suicide timers, to reach the end of their suicide timer without having received any work from the core scheduler. At this point, these nodes stop running to avoid the waste associated with staying up without executing any work. This decentralised mechanism helps to maintain a properly scaled platform.

While it is the nodes themselves that decide when to stop running, the core scheduler has a key role in feeding them work and keeping them alive. Therefore, the scheduling algorithm could decide to keep some nodes alive using specific signals. It could also refuse to feed some nodes with work to force them to deallocate. Such features are not part of our implementation of the framework but they could be the subject of further studies.

Up-scale adaptation mechanism

While the down-scaling mechanism is decentralised and solely based on a local vision, the up-scaling mechanism requires a global estimation of future needs by the deployer.

The up-scale adaptation mechanism is in charge of increasing platform size following spikes in the workload which cannot be handled by the platform in its current state. As such, the estimation of needs, which is the bedrock of the up-scale adaptation mechanism, is only required to run when there is a change in the workload. More specifically, the submission of new workflows to the platform requires the re-evaluation of needs, taking these new workflows into account.

When a relevant change in the workload is noticed, the scheduler informs the deployer. Using the information provided by the scheduler (status of nodes, list of waiting clusters, . . .), the deployer uses its evaluation algorithm to estimate the need for new nodes. If new nodes are required to address the workload, the deployer will decide when to start these nodes. This schedule is then executed by the deployer to rescale the platform. It is followed to the letter unless replaced by an updated run of the need evaluator.

Section III.4 Validation of the model through simulation

Before implementing the loops described in this chapter in a real workflow engine, we wanted to evaluate the efficiency of this approach and solve issues, such as the “locked” node issue discussed in Section III.2.2.C, that stayed unnoticed during the initial design.

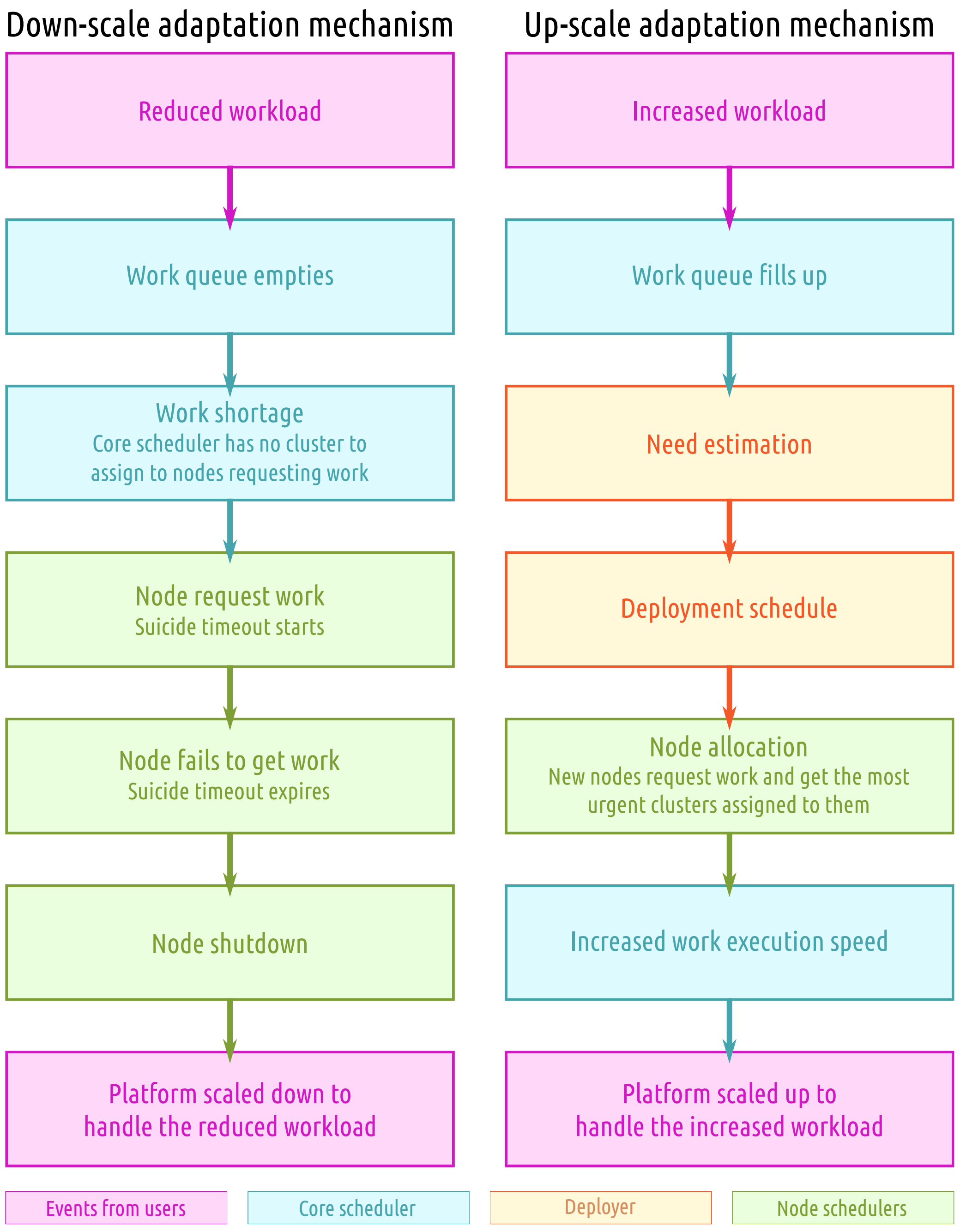

Down-scale adaptation mechanism
Reduced workload
Work queue empties
Work shortage
Core scheduler has no cluster to assign to nodes requesting work
Node request work
Suicide timeout starts
Node fails to get work
Suicide timeout expires
Node shutdown
Platform scaled down to handle the reduced workload
Up-scale adaptation mechanism
Increased workload
Work queue fills up
Need estimation
Deployment schedule
Node allocation
New nodes request work and get the most urgent clusters assigned to them
Increased work execution speed
Platform scaled up to handle the increased workload
Events from users
Core scheduler
Deployer
Node schedulers

Figure 20: Details of the two platform adaptation mechanisms. The relations between the different agents involved is shown in Figure 19 (same color legend).

III.4.1 Implementation of the simulator

This evaluation requires the development of a simulator. This simulator is in the continuity of the one already written to evaluate the efficiency of the clustering algorithm in Chapter II. Written in Python, it implements classes representing all agents, each implementing the required features.

The simulator is single-threaded and does not perform actual computation or data transfers. Instead, it keeps track of current time using a global variable. The real-time aspect is modelled using a queue of events. Events are dequeued and processed in chronological order. Processing an event corresponds to a transition function in one of the agents. This transition function will update the agent status and eventually schedule new events.

III.4.2 Algorithmic details

In this section, we detail the most critical parts of the simulator. The framework developed in this chapter is designed as a generic structure. Other implementations would vary depending on the technologies used. The section provides pseudocode for the most important algorithms. They should be recognisable in all implementations of the framework. We later discuss the differences between our Python-based simulator and the implementation in an existing middleware.

III.4.2.A *Core scheduler*

The core scheduler (see Section III.2.1.B) is arguably the most critical part of the scheduling loop. In addition to handling cluster placement onto the nodes, it is responsible for synchronisation between nodes through the data notification mechanism.

Workflow registration

The first step in a workflow lifecycle is the registration step (Algorithm 2). That is when the workflows joins the rest of the workload, metrics are initialised (line 2 to 5), and clusters are scheduled for placement (lines 6 to 11).

Core scheduling loop

The core scheduler, like most agents in the engine, is built around a wakeable daemon thread running a service upon request. The service run by the core scheduler is unsurprisingly called the *core scheduling loop*. Described in Algorithm 3, this loop is awakened by either new clusters being ready or nodes requesting work (see Algorithm 4, Algorithm 5 and Algorithm 6). The core scheduling loop is responsible for cluster placement, which can be achieved using different methods.

As part of the implementation, we designed three procedures for task placement, each with their advantages and drawbacks. I recommand using either the frontfill (Algorithm 4) or the backfill (Algorithm 5) but not both at the same time. The unlockfill (Algorithm 6) is an additionnal mechanism that does not replace the other basic cluster placement algorithms but helps solving the “locked-nodes”

Algorithm 2 Core scheduler – workflow registration

```
1:  $dag \leftarrow$ Parsed workflow
2:  $clustering \leftarrow$ clustering computed by the DaaS-aware-DCP optimisation
3:  $schedule \leftarrow$ schedule computed by the DaaS-aware-DCP optimisation
4:  $deadline \leftarrow \max(now + schedule.\text{makespan}(), dag.\text{requiredDeadline}())$
5:  Register $dag$, and meta-data
6:  for all $cluster \in clustering.\text{allClusters}()$ do
7:      $waitingClusters.\text{insert}(cluster)$
8:      if $cluster$ contains an entry task then
9:          updateStatusToReady($cluster$)
10:     end if
11: end for
12: Wakeup need evaluation loop
13: Wakeup core scheduling loop
```

Algorithm 3 Core scheduler – core scheduling loop

```
1:  if backfill is enabled then
2:      Run Back-fill placement algorithm                ▷ see Algorithm 4
3:  end if
4:  if frontfill is enabled then
5:      Run Front-fill placement algorithm               ▷ see Algorithm 5
6:  end if
7:  if unlockfill is enabled then
8:      Run Unlock-fill placement algorithm              ▷ see Algorithm 6
9:  end if
10: Go to sleep
```

issue. One should keep in mind that the multi-layer scheduling approach is a modular framework and not a set of definitive algorithms. Therefore, future work could focus on designing better algorithms for specific parts of the framework while keeping its general structure intact.

Cluster placement method: Frontfilling

The idea behind the front-filling method (Algorithm 4) is to place clusters by order of priority onto the first available node that meets its requirements. This mechanism serves all requesting nodes equally, which tend to distribute the workload across the platform.

Algorithm 4 Core scheduler – Frontfill cluster placement

```
1: for all $(alap, cluster) \in$ readyClusters do
2:     for all $(ready, node) \in$ requestingNodes do
3:         if $ready \leq alap$ or $alap \leq now$ then
4:             $node.\text{placeCluster}(cluster)$
5:             break
6:         end if
7:     end for
8: end for
```

Distributing the ready clusters among all available resources means that lower priority clusters, which could potentially be placed on nodes that are currently busy, will take up fresh resources. This distribution can affect the effectiveness of the down-scaling mechanism (see Section III.3.3) as work is widely distributed and prevents nodes from shutting down. It maximises the future availability of nodes but also means that urgent workflows submitted later on might not find available resources.

The higher level of parallelisation of the method also implies that future workflows will have more widely available platform in the long run.

Cluster placement method: Backfilling

Another idea is to save available resources now, and consolidate a few instances by delaying clusters. The back-filling approach (Algorithm 5) once again considers clusters in order of priority, but for each one of these, we select the requesting node with the latest availability that is compatible with cluster placement.

Algorithm 5 Core scheduler – Backfill cluster placement

```
for all ($alap, cluster$) ∈ readyClusters do
    $target \leftarrow NULL$
    for all ($ready, node$) ∈ requestingNodes do
        if $ready \leq alap$ then
            $target \leftarrow node$
        end if
        if $alap \leq now$ then
            $target \leftarrow node$
            break
        end if
    end for
    if $target \neq NULL$ then
        $target$.placeCluster($cluster$)
    end if
end for
```

This policy aim is to consolidate node queues rather than distributing work equally. It helps with platform down-scaling, but also leads to more oscillations of the platform auto-scaling engine. It also preserves more resources for the execution of unexpected urgent workflows while limiting them to a narrower platform (some nodes being filled with work for a long time).

Cluster placement method: Unlock-fill

The unlock-fill approach is not a standalone placement method. It is indeed designed to complement the main front-fill or back-fill placement. In Section III.2.2.C we discussed the locked node issues and explained that a node that has its queues filled with waiting tasks should require work. This is where the unlock-fill procedure (Algorithm 6) comes into play.

Once placement has been performed using the primary procedure, we look at all the nodes still requesting work. If any of these still requesting nodes is locked, then we ask it to run an unassigned ready cluster (if any). Knowing which cluster to assign is still an open issue, and optimisation could be performed. Yet, as the primary objective of this procedure is to avoid deadlocks, the design was focused on avoiding this issue and not on optimising the last resort placement. This mechanism only comes into

Algorithm 6 Core scheduler – Lockfill cluster placement

```
1: for all $(ready, node) \in$ requestingNodes do
2:     if $node$.isBlocked() then
3:         for all $(alap, cluster) \in$ readyClusters do
4:             $node$.placeCluster($cluster$)
5:             break
6:         end for
7:     end if
8: end for
```

play if the primary placement procedure, which is where optimisation should first be achieved, failed to schedule all clusters and left locked nodes requesting.

Data availability notification processing

Last but not least, we have to detail the response of the core scheduler to notification that a data object is now available on the DaaS (Algorithm 7).

Algorithm 7 Core scheduler – Process data notification

```
1: for all $task \in data$.outputs() do
2:     $cluster \leftarrow$ task2cluster[$task$]
3:     $node \leftarrow$ cluster2node[$cluster$]
4:     if $node$ is found then
5:         $node$.schedulerDownload($data$)
6:     else
7:         $dependencyCount[task] \leftarrow dependencyCount[task] - 1$
8:         if $dependencyCount[task] = 0$ and $cluster$ is waiting then
9:             updateStatusToReady($cluster$)
10:            Wakeup core scheduling loop
11:        end if
12:    end if
13: end for
```

When a node completes a task which produced a piece of data required outside the scope of this node, then the node uplink manager is charged with the upload of this piece of data to the DaaS. It is when this upload is finished that a notification is sent to the core scheduler. The core scheduler then updates the status of dependencies and forwards the message to any node that might require it. This mechanism is detailed in Section III.2.1.B.

III.4.2.B — *Node scheduler*

Despite holding the computing power of the platform, nodes are usually not part of the scheduling process and contain a minimalistic logic. Some middleware [Caron et al., 2002] use a push logic, where the scheduler feeds the worker depending on the advertised capability. Others [Anderson, 2004, Cappello et al., 2002] rely on a pull logic, where the workers request work and the scheduler provides them with what is available. However, in our multi-layer scheduling approach, the nodes have an internal scheduling mechanism that manages both compute and communication tasks.

The cluster placement mechanism follows a hybrid logic. Nodes first require work (pull logic), but the core scheduler does not necessarily provide them with work right away. The core scheduler can use its internal logic and keep a list of requesting nodes to place clusters where and when it is the most adequate (push logic). This hybrid model is possible because nodes can keep working while their request is still pending.

Algorithm 8 Node scheduler – Add cluster

```
Cancel suicide timer
Mark node as not requesting
$timmingReady \leftarrow \max(timmingReady, now + cluster.\text{prefetchDuration})$
for $task \in cluster$ do
    $taskQueue.\text{append}(task)$
    $dependencyCount[task] \leftarrow \text{length}(task.\text{inputs})$
    $timmingReady \leftarrow timmingReady + task.\text{duration}$
    for $data \in task.\text{inputs}$ do
        if $data$ is available locally then
            $dependencyCount[task] \leftarrow dependencyCount[task] - 1$
        else if $data$ is available on DaaS then
            $downlinkQueue.\text{append}(data)$
            Wakeup downlink manager
        end if
    end for
    if $dependencyCount[task] = 0$ then
        Wakeup task manager
    end if
end for
```

Once work is assigned to a node (Algorithm 8), the node's internal scheduling takes over and manages both the computing resources and the network interface. This internal scheduling is composed of three managers and their associated queues:

- The downlink manager (Algorithm 9) handles the retrieving of the pieces of data required to compute the tasks in the task queue. It is also in charge of waking-up the task manager when dependencies are met to compute a new task.
- The task manager (Algorithm 10) runs the tasks for which all dependencies are satisfied. It then informs the uplink manager of newly available pieces of data.
- The uplink manager (Algorithm 11) handles all pieces of data produced locally. It updates the dependency local control mechanism if the data is used locally. Otherwise, it notifies the core-scheduler which then forwards this notification to other nodes that might require it, thus closing the loop.

The notifications sent by other nodes are processed by Algorithm 12. Processing such a notification is as simple as adding a job in the downlink queue and waking up the downlink manager to ensure the job gets executed.

Algorithm 9 Node scheduler – Downlink manager

```
1: while $downlinkQueue$ is not empty do
2:     $data \leftarrow downlinkQueue$.pop()
3:     Download $data$ from DaaS
4:     $dependencyCount[task] \leftarrow dependencyCount[task]-1$
5:     if $dependencyCount[task] = 0$ then
6:         Wakeup task manager
7:     end if
8: end while
9: Require work from core scheduler
```

Algorithm 10 Node scheduler – Task manager

```
1: while $taskQueue$ is not empty do
2:     $task \leftarrow$ ready task ($dependencyCount[task] = 0$)
3:     if no such task then
4:         break
5:     end if
6:     Run $task$
7:     for $data \in task$.output do
8:         $uplinkQueue$.append($data$)
9:         Wakeup uplink manager
10:    end for
11: end while
12: Require work from core scheduler
```

Algorithm 11 Node scheduler – Uplink manager

```
1: while $uplinkQueue$ is not empty do
2:     $data \leftarrow uplinkQueue$.pop()
3:     $requireUpload \leftarrow$ false
4:     for $task \in data$.output do
5:         if $task$ in $taskQueue$ then
6:             $dependencyCount[task] \leftarrow dependencyCount[task]-1$
7:             if $dependencyCount[task] = 0$ then
8:                 Wakeup task manager
9:             end if
10:        else
11:            $requireUpload \leftarrow$ true
12:        end if
13:    end for
14:    if $requireUpload$ then
15:        Upload $data$ to DaaS
16:        Notify core-scheduler
17:    end if
18: end while
19: if no task is running and $taskQueue$ is empty then
20:     Start suicide timer
21: end if
```

Algorithm 12 Node scheduler – Process data availability notification

```
1: $downlinkQueue$.append($data$)
2: Wakeup downlink manager
```

III.4.2.C Deployer

The deployer is the heart of the up-scaling deployment loop. It is responsible for the computation and execution of a deployment schedule that determines if and when to provision new nodes from the Cloud provider.

When notified by the core scheduler of relevant changes in the workload, the deployer will update the deployment schedule. The deployment schedule is a simple list of future timestamps indicating when to startup new worker nodes.

Structure of the update mechanism

The update mechanism (Algorithm 13) contains four steps:

- Canceling the previous plan (if any);
- Capturing the current status of the workload and platform;
- Simulating the placement of the workload (and determining when additional resources are required to achieve the expected QoS);
- Scheduling a plan to deploy the resources that simulation identified as required.

Algorithm 13 Deployer – Update deployment schedule

```
1: Cancel last deployment plan
2: $workload \leftarrow$ Workload snapshot
3: $platform_{current} \leftarrow$ Platform snapshot
4: $platform_{extra} \leftarrow$ Empty platform
5: $allocationPlanning \leftarrow$ Empty list
6: $simulation \leftarrow$ new Simulation($workload, platform, allocationPlanning$)
7: $simulation$.schedule()
8: $simulation$.consolidate()
9: Schedule deployment using $allocationPlanning$
```

The environment captured in the second step is composed of a representation of the workload and the platform.

For the workload, we consider all the clusters, ready and waiting, which are not yet assigned to a node. For each one of them, we consider the amount of work they contain (duration), the time they are expected to be ready (which is set to *now* for the ready ones) and the latest they can safely start running.

For the platform we consider all the nodes, requesting or not, with the timestamp at which they are expected to be ready.

Simulation of future placement

The simulation, which corresponds to the analysis step of the MAPE-K autonomic loop, is divided into two parts.

This first part (Algorithm 14) places the clusters on the different nodes using a list scheduling heuristic. Between cluster placement, we update the workload status.

Algorithm 14 Deployer – Simulate list scheduling

```
1: while $workload$ is not empty do
2:     while $workload$ has ready cluster do
3:         $cluster \leftarrow workload$.popReady()
4:         $(instance, start, stop) \leftarrow$ self.place($cluster$)
5:         if $extendedwall \leq start$ then
6:             break                              ▷ option to limit the extend of the simulation
7:         end if
8:         $planning[instance]$.append($cluster, start, stop$)
9:         $workload$.popReady(timing=$platform$.earliest().available)
10:    end while
11:    $workload$.popReady()                     ▷ Transition one cluster from waiting to ready
12: end while
```

The placement heuristic (Algorithm 15) tries to put the clusters onto the already existing nodes ($platform_{current}$) first. It then moves on to the new nodes which we already know will have to be allocated ($platform_{extra}$). If this is not enough, then a new node is initialised and inserted into the list of nodes that will have to be allocated ($platform_{extra}$).

Algorithm 15 Deployer – Simulate cluster placement

```
1: $earliestboot \leftarrow now + bootDuration$
2: if $platform_{current}$.earliest().available < $cluster$.ALAP then
3:     $instance = platform_{current}$.pop()
4:     $isextra \leftarrow$ false
5: else if $platform_{extra}$.earliest().available < $cluster$.ALAP then
6:     $instance = platform_{extra}$.pop()
7:     $isextra \leftarrow$ true
8: else
9:     $instance \leftarrow$ new Instance(available = $earliestboot$)
10:    $isextra \leftarrow$ true
11: end if
12: $start \leftarrow$ max($instance$.available, $cluster$.ASAP)
13: $stop \leftarrow start + cluster$.duration
14: $instance$.available $\leftarrow stop$
15: if $isextra$ then
16:     $platform_{extra}$.push($instance$);
17: else
18:     $platform_{current}$.push($instance$);
19: end if
20: return ($instance, start, stop$)
```

New instances consolidation

Once cluster placement is finished, we try to consolidate the planning of the new nodes (Algorithm 16). For each new required node, we traverse the list of clusters that have been assigned to it and try to move them back as far as possible. It tells us when the nodes can be deployed at latest. This delays any spare time on these nodes and thus increases their reusability by upcoming workflows.

Algorithm 16 Deployer – Schedule consolidation

```
1: for all $instance$ in $platform_{extra}$ do
2:     $required \leftarrow +\infty$
3:     for all $cluster \in$ reverse($planning[instance]$) do
4:         $required \leftarrow \min(required - cluster.\text{duration}, cluster.\text{ALAP})$
5:     end for
6:     $start \leftarrow \max(required - bootDuration, now)$
7:     $allocationPlanning$.push($start$)
8: end for
```

III.4.3 Results

In this section, we will study the performance of our approach, combining both the static analysis and the autonomous workflow manager, for the execution of synthetic yet realistic workflows [Bharathi et al., 2008].

III.4.3.A *From fixed platforms to autonomic deployment*

A simple, non-autonomous approach to single or multi-tenants Cloud computing is to deploy a given number of nodes and to use a batch scheduler for the workload deployment. While such approach can free the user from the burden of managing an on-premise computing cluster, it does not benefit from all the dynamicity offered by Cloud infrastructure. More specifically, the quantity of resources on this Cloud cluster is fixed, and any modification requires human intervention. As a consequence, the platform deployment will most likely not match the varying workload submitted by users.

Even with users modifying their behaviour to try matching resources availability (submitting large jobs at night and during weekends), some critical jobs might still not find the computing resources they need to finish in time. On the other hand, nodes will most likely not be used 100% of the time, and having unused nodes in the platform is a waste of money. Besides, it is difficult to decide how many nodes should be allocated to such a platform. Having too few nodes leads to congestion during peak-hours and deadlines not being met, while having too many nodes means having unused resources and a higher bill at the end of the day. In fact, such platforms are generally allocated to match the available budget and not actual needs.

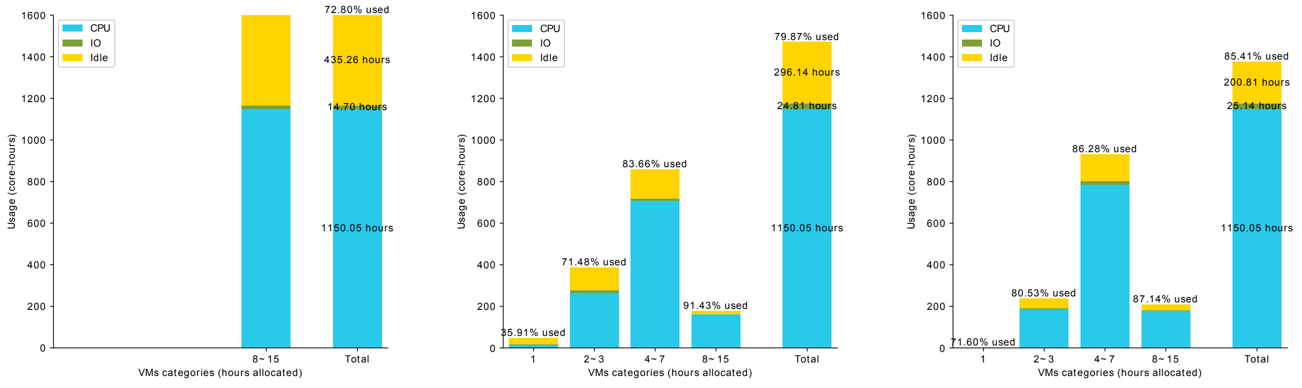
With our framework, the number of resources on the platform automatically adapts to match users' requirements. As such, more resources are allocated to meet users' requirements during peak hours, and resources are deallocated when not in use, thus reducing the deployment cost. During off-peak hours platform usage can drop drastically, and with our autonomic approach, all but the few required nodes will be shut down to ensure a high use-rate of these few nodes.

One of the goals of our framework is to have multiple users share the platform. The underlying idea is that if someone has a tight deadline, they will have to deploy many nodes to achieve it. Some of these nodes might be used for only a small fraction of the minimum allocation time. As a consequence, this user will have to pay for mostly unused nodes. On the other hand, if this user shares the platform, the queues of mostly unused nodes could be filled using workflows from other users who do not have tight deadlines. It helps increasing node usage and thus achieving higher efficiency.

To evaluate the effectiveness of our design, we compare the resulting deployment to other approaches. We based this evaluation on workloads containing many realistic workflows [Bharathi et al., 2008] submitted at various times. The deployment mechanism has no knowledge of future workflows before they are submitted. Once a workflow has been submitted, the platform manager can react to it and allocate new resources if needs be. The analysis part of our MAPE-K loop (introduced in Section III.1) is performed using a simple ASAP (As Soon As Possible) list scheduling algorithm.

III.4.3.B *Execution of a dense workload*

Our first workload is a dense workload combining 100 realistic workflows distributed over an extended period. By dense we mean that the quantity of work is large enough, and evenly distributed so that most machines are used most of the time. For such a workload, the number of machines required does not change much, and fixed allocation can achieve decent result if they are scaled correctly in the first place.



(a) Fixed platform
200 persistent nodes

(b) Independent platforms
396 dynamic nodes

(c) Shared platform (Our framework)
298 shared dynamic nodes

Figure 21: **Dense workload:** VMs usage, grouped by VM allocation time, during the execution of a dense workload of 100 Epigenomics workflows with various deadlines.

Figure 21 shows the benefits of our autonomic approach (Figure 21c) over a fixed platform (Figure 21a). Each figure shows the cumulated usage of VMs grouped by their total allocation time.

- **CPU usage:** CPU is performing tasks (in blue);
- **IO usage:** CPU is idle, but nodes are either prefetching data or sending results (in green);
- **Idle time:** node is waiting for new tasks (in yellow).

To perform the same workload within the required deadlines, a fixed platform would have required 200 nodes and taken the most of 8 hours, resulting in a total bill of 1600 core-hours. On the other hand, our autonomic approach used 298 instances. By allocating and deallocating these instances depending on real-time requirements, these 298 instances were only allocated for a total of 1358 core-hours. This 15.1% decrease in resource usage also corresponds to the same decrease in deployment cost.

While the submission of workflows follows a uniform distribution, the different deadlines (submitted by different users), as well as the shape of workflows, cause some bursts in the workload. These bursts are the reason why we need an adaptive approach.

Even with sustained workloads, setting up a fixed platform of the proper size is not easy. The platform is likely not to have the optimal size. *De facto*, most institutions tend to over-scale their platforms to avoid congestion. In this example, if the fixed platform contained under 200 nodes, we would have noticed resources shortage resulting in workflows not meeting their deadline; and if the platform contained over 200 nodes, we would have wasted more resources as more nodes would have been available and unused.

Figure 21 shows the benefit of a shared platform (Figure 21c) over multiple independent platforms (Figure 21b) where each worklows realies on an independent autonomic deployer and scheduler. Considering a workload of 100 workflows with various submission dates and deadlines, we see that sharing the platform helps reducing the overall deployment cost by 10.9%.

One of the visible consequences of platform sharing is the reduction of the number of VMs allocated for only 1 hour and which are the most subject to waste. Among the many VMs allocated for only 1 hour in our independent platforms run, we observe almost 65% of wasted time. These are VMs that have been allocated to answer urgent needs and ensure the deadlines of some workloads are met. They are not reused by other users' further submission. On the other hand, our shared platform run shows more VMs allocated for 4 ~ 7 hours. These are VMs that are used by many workflows. This workload allowed them to stay up longer and to limit the waste fraction under 20%. Finally, in the shared platform model, some VMs were able to acquire the work of others. This behaviour explains the smaller runtime of VMs running for more than 8 hours. The consequence of the redistribution of work is the fact that some workflows were able to send their results long before their expected deadline through the used of resources allocated to other jobs that would not have been allocated as early otherwise.

All in all, the execution of workflows on independent platforms required a total of 1525 core-hours while the use of a shared platform only required 1358 of these same core hours. This 10.9% decrease in the volume of computing resources allocated also represents a 10.9% decrease of the deployment cost.

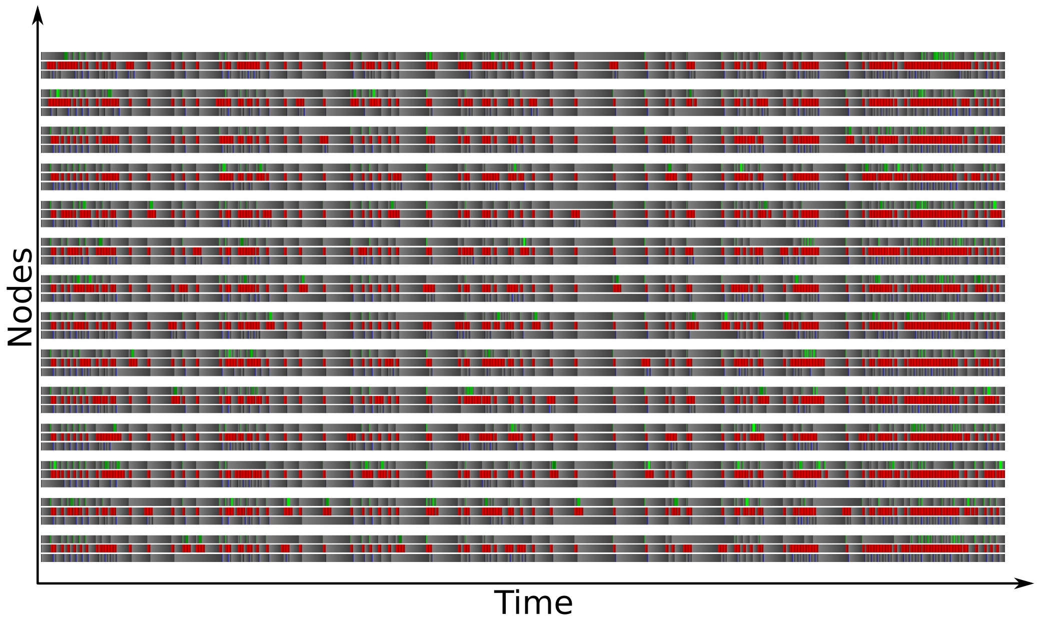
III.4.3.C *Execution of a sparse workload*

Unlike our dense workload model, real workloads are usually not uniform. Users tend to submit short jobs (with tight deadlines) during the daytime, while long-lasting jobs (with remote deadlines) are likely to be the only ones running during nights and weekends.

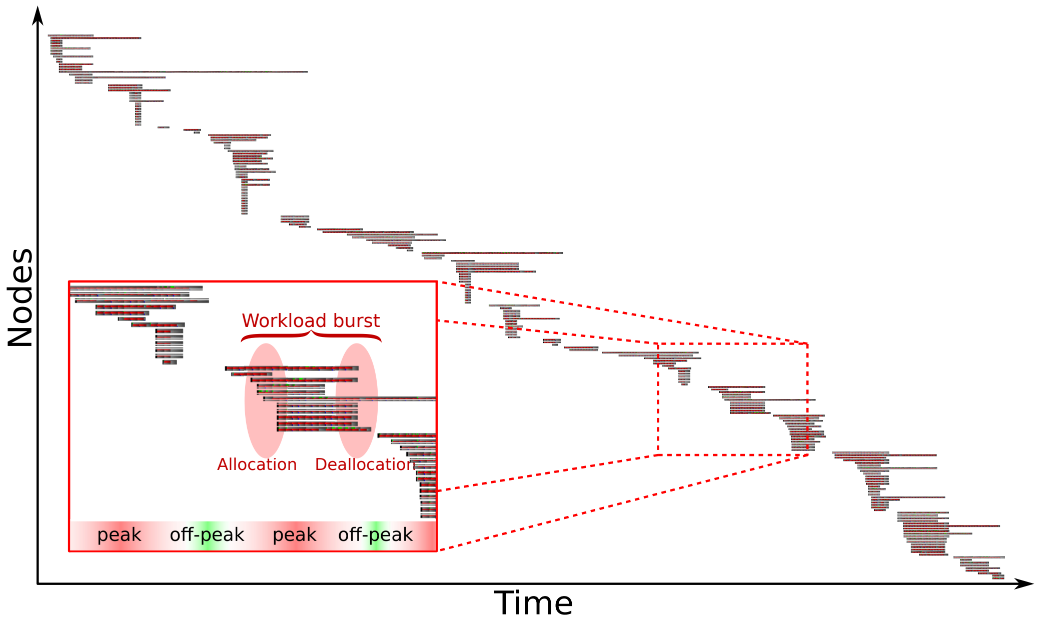
Therefore, we designed a second, sparse, workload combining 100 workflows distributed over an extended period. By sparse we mean that the quantity of work, compared to the duration of the experiment, is small enough so that we can observe burst periods (*peak*) and calm periods (*off-peak*) where little to no resources are used.

Our framework's foremost objective is to rescale the platform between these different phases automatically.

Figure 22 shows the usage of all nodes during the execution of this workload on a fixed platform (Figure 22a) and on a shared platform using our framework (Figure 22b). Each VM in the platform is represented by a group of 3 lines, with the horizontal axis representing time. These lines represent the use of the CPU (red when busy) as well as the downlink (green when busy) and uplink (blue when



(a) Fixed platform
14 persistent nodes



(b) Shared platform (Our framework)
203 shared dynamic nodes

Figure 22: **Sparse workload:** Gantt diagram of VMs, during the execution of a sparse workload of 100 Fork-join workflows with various deadlines. For each VM (line) the CPU compute time is shown in red.

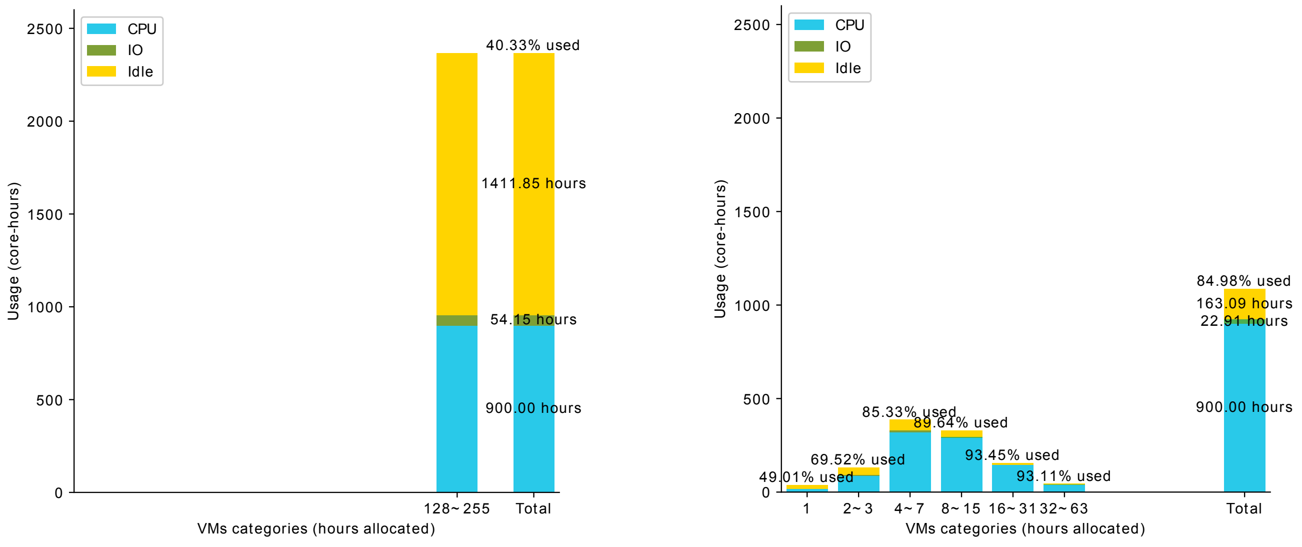
busy). Idle time is shown in grey.

We can see that Figure 22a contains 14 nodes, with some peak hours where all nodes are busy (shown in red) while off-peak hours are periods where nodes are mostly idle and are visible in grey. Using 14 nodes is an arbitrary compromise between the idle time where none of these nodes is busy and the time where all 14 nodes are not enough to meet the workload. Having more nodes would further increase the waste during off-peak hours while having fewer nodes would further decrease the QoS.

Figure 22b shows results using our approach. As we have many more instances (203), it is difficult to see individual node activity. However what we can see is the allocation and deallocation of nodes throughout the experiment. During peak hours, many nodes are allocated and used simultaneously to face the increased workload. This is shown as groups of nodes (parallel lines) sharing a common time-frame. Transitioning from peak hours to off-peak hours, we see the deallocation of under-used nodes. Overall, bursts in the workload are visible in this Gantt chart as groups of nodes with a limited lifespan. Still, some nodes from these groups stay alive longer to take care of the reduced load during the off-peak hours which follow the bursts.

Our framework automatically allocates resources to meet all deadlines. On the other hand, the fixed approach required extensive trial-and-error to optimise the number of nodes required to meet these deadlines, and therefore achieve the same QoS as our framework. In real deployments, this number is likely not to be the right one, leading to either a poor QoS (deadline not met) or a higher deployment cost.

Figure 23 shows the VMs usage, grouped by VM allocation time, during the execution of our sparse workload. While total CPU time is the same (same workload), having an auto-scaling shared platform helps reducing the wasted (Idle) time and therefore decreasing deployment cost. In this example, using our framework helped reducing costs by 54.3% from 2366 to 1081 cores-hours. This shows that our framework achieves much better results than an optimal fixed platform which cannot react to the unavoidable disparities in the workload.



(a) Fixed platform
14 persistent nodes

(b) Shared platform (Our framework)
203 shared dynamic nodes

Figure 23: **Sparse workload:** VMs usage, grouped by VM allocation time, during the execution of a sparse workload of 100 Fork-join workflows with various deadlines.

Section III.5 Discussions

In this chapter we described how to use the clustered representation of workflows built in Chapter II to build an autonomous workflow engine. Three objectives influence the design of this workflow engine: autonomous platform adaptation, multi-tenant platform sharing, and deployment efficiency (QoS and cost). This was made possible by the use of a two-step scheduling approach.

The use of a two-step scheduling means that workflow control is partly distributed to the nodes. They are not just computation resources for the execution of the services required by the workflows but are also part of the control mechanism of the whole platform. This move toward more decentralisation offloads some work away from the scheduler and should help with the scalability of the infrastructure.

III.5.1 Future works

While mechanisms described in this chapter are sufficient to build an autonomous workflow manager with the required features, it is a modular design, and future work should hopefully be able to improve on the current features while keeping the same decentralised architecture of agents.

III.5.1.A *User requirements*

The users describe the required Quality of Service as a deadline for the execution of their workflows. Placement and scheduling is an optimisation problem that tries to minimise or maximise a metric while respecting constraints. In our case, the engine tries to achieve a minimal deployment cost while respecting the deadlines. Tight deadlines will force the allocation of resources, leading to a higher cost, while workflows with loose deadlines will only be executed if it does not induce additional cost (best effort).

Another possible approach would be to try minimising the makespan with constraints on the deployment cost. Future work should focus on the implementation of this approach. As our approach targets multiple users, the two approaches should coexist at the same time.

Keeping track of budget is not easy, but could be part of the meta information available with the DAG. A common vision is that the overall budget should first be divided between the clusters (during the static analysis step) and used to determine whether placement on a node is relevant. This cost should then be updated when a cluster of the same DAG is finished. However, measuring the cost of execution of a particular DAG might be a complex problem when workflows that have to remain within a specific budget share nodes. Need evaluation would also be more difficult, as the need for new nodes would become a more relaxed requirement.

Currently, users of platforms with limited resources can use very far deadlines to avoid allocation of new nodes and use the existing ones in a best-effort mode. Administrators could enforce constraints on the required deadlines so to avoid that the users request too expensive computations.

III.5.1.B *Heterogeneous platforms*

In the conclusion of Chapter II we discussed the need for heterogeneous platform support and the difficulty of such a feature. This problem also affects the vision of the platform we discussed in this chapter. If we imagine that the static analysis step provides a type of VM to use for each cluster we have two possibilities:

- Run independent platforms, each with its dedicated scheduler, platform manager and need evaluator. Each platform would be in charge of a family of nodes matching the same VM requirements, and would only consider the clusters of tasks that are to run on this family of nodes. Still, clusters would have to push notifications to clusters of different types that are part of the same workflow. This notification process and dependency control should thus be shared by all sub-platforms.
- Enhance the need evaluator and the core scheduler to consider these heterogeneous resources. The need evaluator could consolidate needs (allocating an M instance instead of two S or anticipating the placement of an L cluster on an available XL node). The placement job, performed by the core scheduler, would also be much more difficult.

While the first approach is simple, and only requires marginal evolutions of the agents, implementation of the second approach would require tremendous work. The topic of heterogeneous platform management, from offline static analysis to online platform management, could be an exciting topic for a future PhD candidate.

Chapter IV

Implementation

After having designed the offline precomputation of task clusters (see Chapter II) and the online scheduling of these clusters, which goes in pair with the autonomic management of the platform (see Chapter III) it is now time to move forward and implement these features in a real system.

Section IV.1 The DIET middleware

IV.1.1 Remote procedure calls

DIET is a middleware whose development started in 2002. It uses a hierarchical design to provide a scalable platform based on the GridRPC[17] paradigm. With a usage similar to simple library calls, remote procedure calls are designed to provide a way for developers to access external resources easily.

Remote procedure calls were implemented as part of multiple middleware [Caron et al., 2002, Tanaka et al., 2003, Yarkhan et al., 2007], each one with its distinct interface. The absence of interoperability between middleware led to the standardisation of the GridRPC API by the Open Grid Forum in 2007 [Seymour et al., 2007].

This approach initially designed for private grid platforms raised a renewed interest with the emergence of commercial Cloud solutions. In 2014 Amazon introduced AWS Lambda[18], a mechanism that provides access to AWS resources through remote procedure calls.

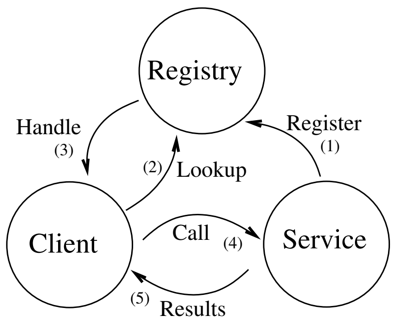


Figure 24: Overview of the GridRPC paradigm

According to this paradigm (Figure 24), performing remote procedure call in a supervised grid infrastructure requires five steps:

1. Registration: Service providers notify the registry of the services they provide;
2. Lookup: When a client wants to use a service, he contacts the registry to ask who provides this service;
3. Handle: The registry answers with a pointer to a node which provides the service required;
4. Call: The user uses the pointer provided by the registry to execute a remote procedure call on the service provider;

[17]Grid Remote Procedure Call
[18]AWS Lambda: https://aws.amazon.com/lambda/

5. Results: Once the service call is done, the service returns any subsequent results to the client.

IV.1.2 Overview of DIET

In the DIET architecture (Figure 25) services are called SeDs (Server Daemons) and the registry is called the MA (Master Agent). Additional agents, called LA (Local Agents), which are not part of the GridRPC description, can be added to the tree structure to improve scalability. As shown in Figure 25, multiple MAs can be linked together to form a federation of platforms.

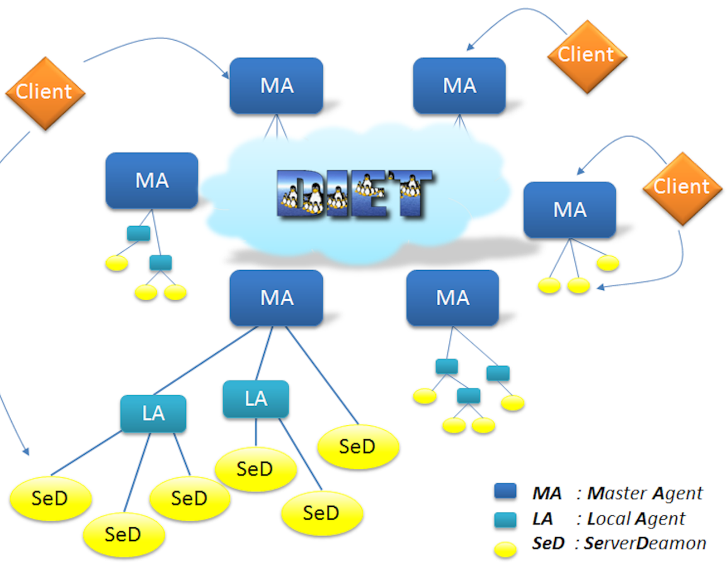


Figure 25: Overview of the DIET architecture

To simplify the submission of procedure calls by the users, DIET hides the Lookup/Handle part of the GridRPC paradigm. When a user asks for a service execution, the MA takes over, traverses the platform to find an available SeD that provides the required service, forwards the call to the SeD and returns the results to the user once done. The user has no handle and does not need to contact the SeD.

In addition to simple procedure calls, DIET allows the user to submit entire workflows. This feature uses the GWENDIA [Caron et al., 2009] workflow description syntax. Application designers only have to express their application structure using simple services and a generic client will then parse the workflow and execute it.

The structure shown in Figure 25 contains everything needed for traditional RPC calls but misses the MADAG. The MADAG is an extension to the MA which is in charge of supervising workflow (structured as DAGs) execution on the platform. It keeps track of task readiness and triggers the RPC call to the required services. Prior to my work, multiple task submission policies were already implemented. All these policies rely on a HEFT based priority assignment and lack task clustering. Achieving good data locality falls under the data manager responsibility. However, the data manager is oblivious to the workflow structure and cannot plan effectively without any outlook on future workload.

This implementation relies on pairs of task queues (for the ready and waiting tasks) for each workflow. Each time a task gets ready, the MADAG sends a signal to the user that he can perform the call for this task, and a thread is launched. This thread waits for notification of task termination, which is provided by the user. It will then update the other tasks status. This implementation is thread-heavy (one thread per running task) and centralised (the MADAG performs the entire scheduling). Still, it does not achieve the optimisation that could come with a centralised vision of the platform.

IV.1.3 Why DIET needs autoscalling

While the workflow support in DIET is not perfect, it provides a framework compelling enough to answer the needs of scientists [Amar et al., 2006].

Yet, DIET approach to automating platform deployment is questionable. The *BatchSed* is a mechanism, included in DIET, which allocates new SeD using the OAR interface. However, these SeDs are only used for one single service call before being deallocated. This is the drawback of requiring systematic communication of the computed dataset. With the SeD being reset between tasks, no data locality is possible. Such policy increases the makespan of the workflow and wastes many resources to the configuration of expandable nodes. While it makes some sense in the context of the Grid'5000 platform, it would be an inadequate policy for the provisioning of resources in the context of a commercial Cloud.

DIET needs a mechanism to automate the deployment of re-usable nodes in order to constitute a shared platform that fits the users' needs. A single node is enough to deploy both the MA and the MADAG, so a minimalistic deployment requires very few resources. SeDs can dynamically join and leave a MA, but they require a smart control loop to automate that process at runtime following the workload.

DIET provides a toolbox with many exciting features like encryption and firewall support. These features would greatly help with the deployment of large-scale distributed platforms, but reimplementing them would require much engineering. A DIET-based implementation of the multi-layer scheduling and platform management loops would naturally benefit from these features and facilitate the deployment of large-scale autonomous platforms.

Section IV.2 An upgraded platform

The implementation of the autonomic workflow engine in the DIET middleware calls for a slight evolution of the structure. The new features are implemented in specific autoscale agents and no modification has been applied to the MA / LA / SeD hierarchy. Thus, existing DIET deployments are natively compatible with the autoscale approach, providing the additional autoscale agents are attached to it.

The transition to the autoscale architecture (Figure 26) requires the deployment of three agents: the bridge autoscale, the SeD autoscale, and the upgraded MADAG autoscale.

IV.2.1 Bridge and SeD autoscale

The bridge autoscale is a stateful agent that bridges the gap between the DIET platform and the elastic Cloud API. It is in charge of spawning new SeDs or shutting them down. When spawning a new SeD, the bridge has to instantiate the new VM, configure it and spawn the DIET SeD with the required configuration file so that it can join the platform.

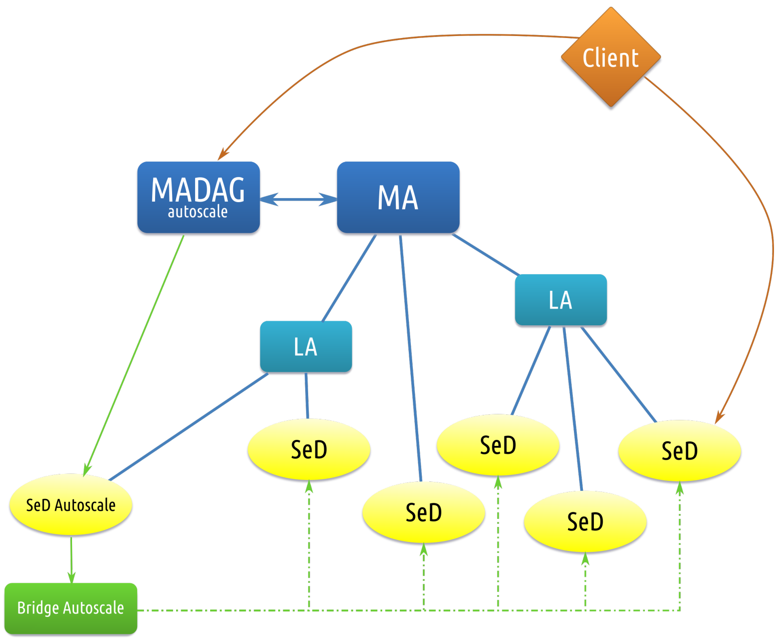


Figure 26: Overview of the DIET autoscale architecture

To deploy the autoscaling DIET platform on Grid'5000 [Balouek et al., 2012] I have built a bridge (in Python3) that uses Execo [Imbert et al., 2013] to interact with the OAR layer of Grid'5000. The bridge is a daemon that answers requests coming from the MADAG. These requests use the DIET call mechanism to reach the bridge through the SeD autoscale. The SeD autoscale is, therefore, a proxy between the stateful bridge, which exposes a ZMQ socket[19] and the DIET call mechanism. Bridge and SeD autoscale come in pairs to form the "autoscale interface". While my implementation (Figure 27) uses two separate programs, we can imagine implementations where the autoscale interface uses a single program which fills both roles.

Having a pair of bridge and SeD autoscale means no provider-specific mechanism is part of the MADAG. Therefore, moving to another provider would only require the implementation of a bridge that exposes platform management primitives as DIET services.

IV.2.2 MADAG autoscale

The MADAG autoscale contains all the platform-independent mechanisms required in the multi-layer scheduling and the autonomous platform manager. It is built on top of the existing MADAG code and is selected at runtime through a command line option. The communication mechanism enforced in DIET matches its push scheduling logic: SeDs answer to queries coming from the MA but are not capable of initiating a communication. This prohibits the construction of asynchronous agents like the node scheduler, which would not be able to notify and wake up the core scheduler on the MA. A workaround would require a clock that queries the nodes regularly, but it would slow down the scheduling process and waste resources.

A better decentralisation would be possible using dedicated CORBA objects on top of the core and node schedulers but would require a massive rework of the SeD, which is beyond the scope of this work.

[19] http://zeromq.org/

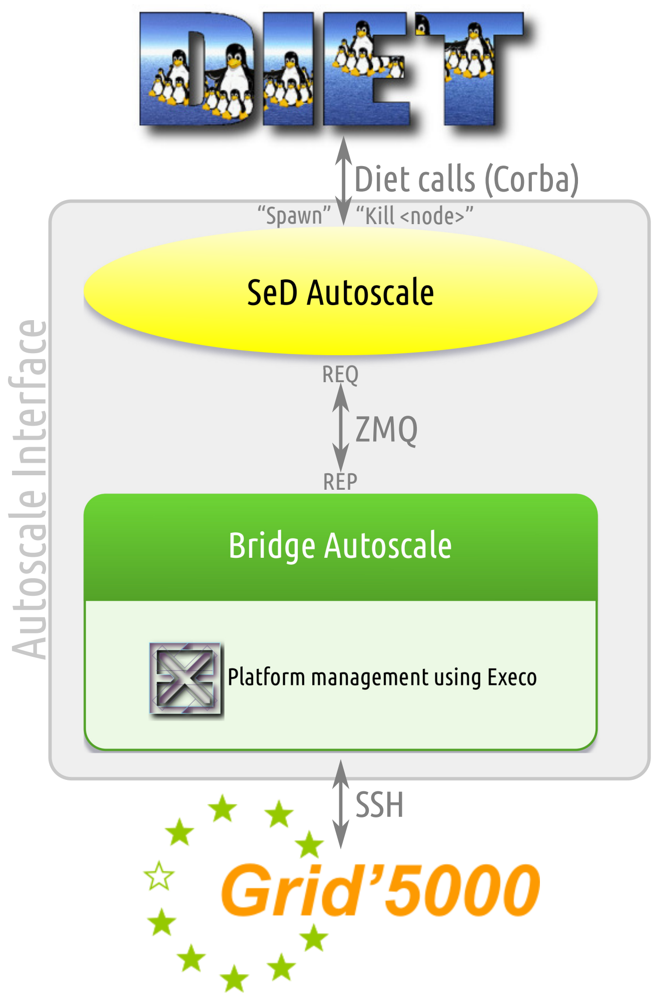


Figure 27: Structure of the autoscale interface used for platform management on Grid'5000

The current implementation of my work in DIET centralises all agents described in Chapter III in the MADAG. It provides all the features of my work. This centralised construction also helps solving all synchronisation issues in a more reactive environment. Moving the node schedulers from the MADAG to the node should be considerably simpler after this initial centralised implementation, and is proposed as future work on the DIET platform.

Section IV.3

Implementation of the MADAG autoscale

IV.3.1

Tools

Before going deep into the development of the MADAG autoscale, and the details of all objects implementing the multi-layer scheduling and platform manager, we are going to have a brief view at some tools that will prove very valuable.

DagInfo

The *DagInfo* structure computes and stores useful details that are not explicitly accessible in the DIET *Dag* representation of workflows. It includes a topologically ordered list of tasks. It also contains mappings linking the tasks to the pieces of data they consume and produce, as well as mappings linking the pieces of data to the tasks that consume and produce them. Last but not least, it computes and stores details about the size of the different pieces of data.

We mostly need this structure because the DIET representation is not a full DAG, but rather an unstructured set of tasks with communication ports, and without any mechanism to quickly associate a task port to the other task ports representing the same piece of data on other nodes of the DAG.

Clustering

Both the static analysis step which is used to optimised data locality, and the multi-layer scheduling that follows it, rely on a clustered view of workflows. The structure used to represent a workflow clustering is named *Clustering* and is implemented using the union-find data structure [Galler and Fisher, 1964]. The implementation is designed to be seamlessly moved and copied so that computation of the DaaS-aware DCP optimisation (see Chapter II) is efficient and straightforward.

DagSchedule

The `DagSchedule` is used to compute an execution planning for a specific workflow given a specific clustering. It implements top level and bottom level evaluation required by the DCP algorithm. Following the logic discussed in Chapter II, the evaluation of communication durations is left as a virtual function. Alongside the `DagSchedule` pure virtual class comes the `DCPDagSchedule` specialisation which implements the communication evaluation developed specifically for DaaS-based Cloud platforms.

Singletons

Many objects in the DIET code are intended to exist in one sole instance and to be easily accessible. For example, there should be only a single core scheduler in a MADAG and agents should have an easy interface to access it. The singleton class provides a simple mechanism to make any class behave as a singleton. Non-default initialisations of the underlying objects are still possible thanks to the use of variadic templates.

Asynchronous objects

The scheduling and deployment loops rely on asynchronous agents with notification capability and timers. This real-time vision is not common to C++ code. Therefore I wrote a set of virtual classes that implement these features and can be inherited. This helps with code separation, maintenance and readability.

All asynchronous objects are built using the `PolymorphicThread` class which is a mere thread abstraction on top of the threads provided by the standard library. It could however be updated to use any other thread representation without needing to update all the different types of agents down the road.

The `NotifiedPulser` (see Figure 28) is an object which, once started, runs a `tick` method every time it is notified. It uses a semaphore to handle the notifications, thus avoiding the use of spinlocks. The included termination method prevents further ticking and stops the thread at the end of the current tick (if applicable).

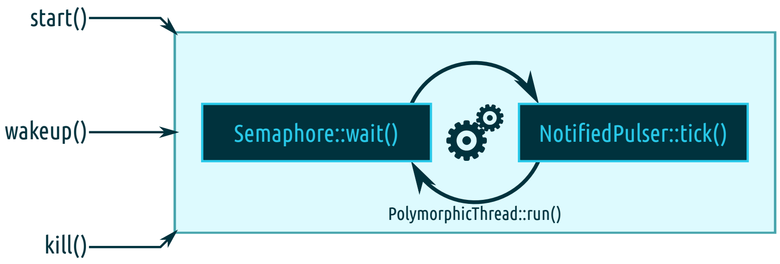


Figure 28: Structure of the `NotifiedPulser`. The `tick` is implemented by the child class.

The `DelayedNotifiedPulser` (see Figure 29) is an upgrade to the `NotifiedPulser` which implements a minimum time between two consecutive wakeups of the object. Notification can still be done at any time using a non-blocking method, but the next call to the ticking method might run after a delay. This behaviour is used in the deployment loop to avoid recomputing deployment plan too often and reduces the instability of the deployment loop.

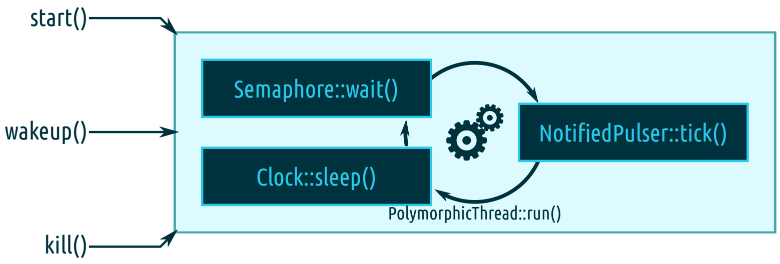


Figure 29: Structure of the `DelayedNotifiedPulser`. The timer duration is specified in the constructor and the `tick` is implemented by the child class.

The `SelfDeletingTimer` is used for scheduling interruptable delayed calls. Upon initialisation, a thread is started and waits until a specific time point to execute some code. This timer can be cancelled, and memory is self-managed, meaning that the timer will deallocate itself at the end of the waiting period. This avoids memory leaks caused by cancelled deployment plan and suicide timers.

Other – simple to use – asynchronous agents were designed but not used. They remain in the Diet codebase for further use.

IV.3.2 MultiWfAutoscale

The various policies implemented in the MADAG correspond to different implementations of the virtual `MultiWfScheduler` class. This virtual class defines the interface that multi-workflow managers have to implement. The multi-workflow managers are in charge of handling the concurrent execution of multiple workflows. When starting the MADAG, an option allows the administrator to select which policy to instanciate.

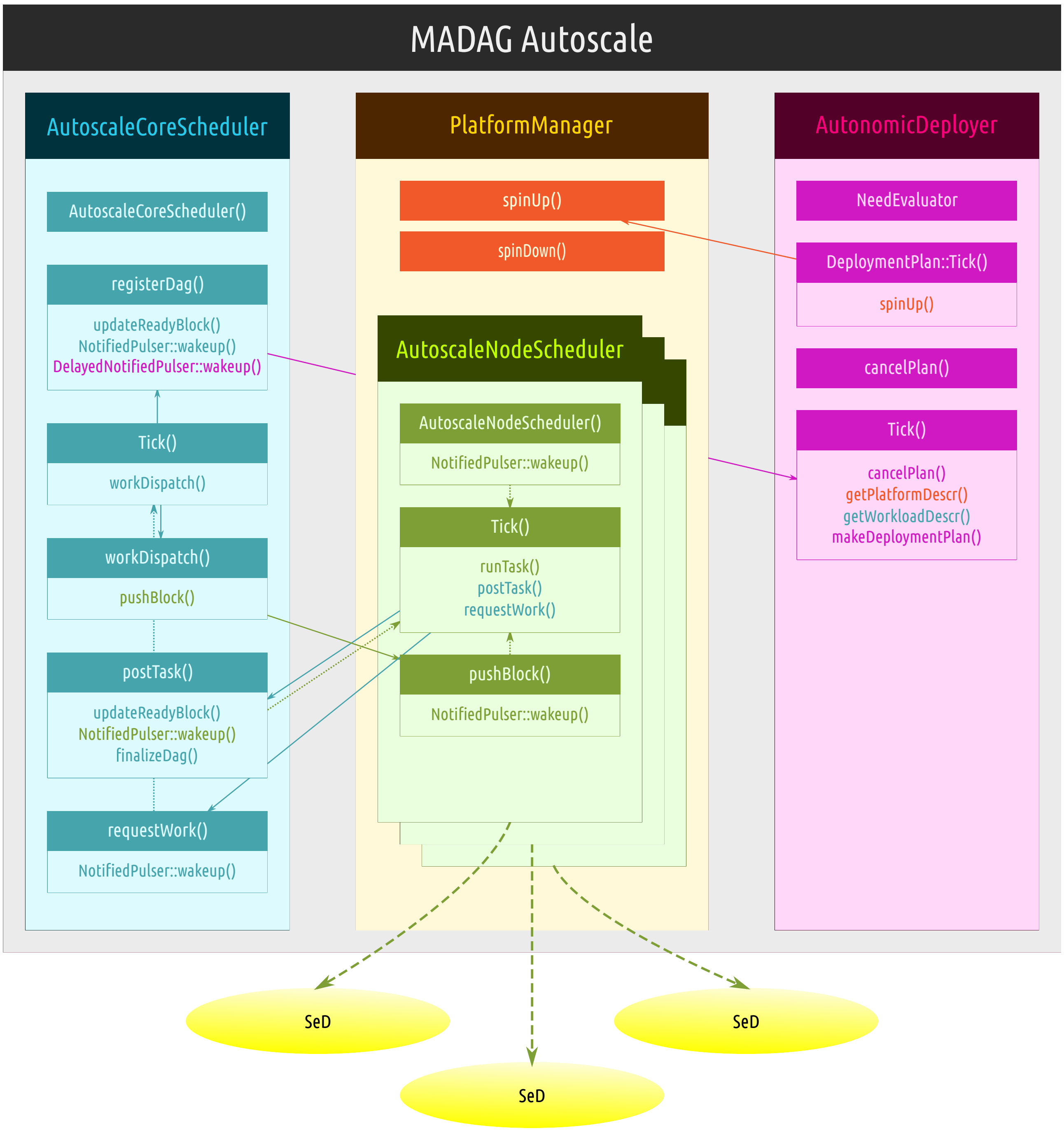


Figure 30: Overview of the MADAG autoscale internal structure

All the existing implementations of this interface, that handle the specific scheduling policies, have kept the scheduling logic inside the scope of this object. This will however not be the case of our approach. Our implementation, the `MultiWfAutoscale`, is used as a proxy between the part of the MADAG that exposes workflow support to the users, and the implementation of the multi-agent scheduling and deployment control loops (see Figure 30).

IV.3.3 AutoscaleCoreScheduler

The `AutoscaleCoreScheduler` is a `NotifiedPulser` `Singleton` which implements the core scheduler. It handles workflows received by the MADAG autoscale trough the `MultiWfAutoscale`, computes the task clustering, and then schedules the clusters of tasks as specified. It also acts as a router that dispatches notifications of data availability to the nodes that require them.

IV.3.4 AutoscaleNodeScheduler

Nodes that have been required to be part of the autoscaling platform are represented, in the MADAG, by `AutoscaleNodeScheduler` objects. They implement the node schedulers described previously but, due to Diet limitations, they are run on threads inside the MADAG and not on the node they manage as the decentralised approach proposes. The `AutoscaleNodeScheduler` is a `NotifiedPulser` that handles scheduling at the node level and manages the down-scaling loop. Instances of the `AutoscaleNodeScheduler` are stored inside the `PlatformManager` and identified by the unique identifier of the SeD they represent. Thanks to this identifier, the `AutoscaleNodeScheduler` can trigger the execution of the tasks in its task queue on the right SeD, and short-circuit the MA placement logic.

IV.3.5 PlatformManager

The `PlatformManager` stores all information regarding the current deployment of the autoscaling platform. Besides, it provides an interface to start and stop nodes. The `spinUp` method, when triggered by an active deployment plan, makes a call to the SeD autoscale to instantiate a new SeD and creates the associate `AutoscaleNodeScheduler`. The `AutoscaleNodeScheduler` calls the `spinDown` method when the suicide timer is reached. The deallocation instruction is forwarded to the SeD autoscale and terminates the local agent that managed it.

IV.3.6 AutonomicDeployer

The `AutonomicDeployer` is a `DelayedNotifiedPulser` that implements the upscaling deployment loop as described in Chapter III. It retrieves description (snapshots) of the workload and platform, evaluates future needs and instantiates a deployment planning. The deployment planning is a self-deleting `Pulser` which calls the `spinUp` method of the `PlatformManager` to instantiate new nodes whenever required.

Section IV.4 Discussions

At first glance, the hybrid strategy proposed by the multi-layer scheduling approach seems incompatible with the DIET communication pattern, and the implementation was in fact very challenging. Hopefully, the proposed design is generic enough so that, despite the distributed implementation not being possible, we managed to implement all the features of this approach in a centralised implementation. Moreover, this centralised implementation in the MADAG autoscale paves the way for future evolutions of the middleware.

IV.4.1 Advantages and shortcomings of this approach

The new MADAG autoscale provides new features to the DIET platform. The most emblematic of it is the ability to adapt the computing platform to the workload. The evolution of the scheduling process provides a way to solve the data-locality issue with limited optimisation cost at runtime. Still, this radical change of approach raises a few issues.

DIET implements a mechanism to evaluate the performances of the different services provided by the SeDs. This mechanism is used to estimate the task completion time used by the different implementations of the `MultiWfScheduler`. Estimation of the task completion time, as well as data transfer duration, are required by the DaaS-aware DCP clustering of workflows (see Chapter II). When working in the context of a dynamic platform, we can face the situation where a workflow is submitted while no SeD is running (empty platform). The deployment of the adequate computing resources requires the need evaluation algorithm to consider a clustered vision of the workload. Building this clustering requires a prior estimation of the tasks and data transfers durations which, in the DIET paradigm, requires SeDs running. This conundrum was solved by discarding estimation mechanism provided by DIET and including the required metadata in the workflow description.

Another logic mismatch between the DIET paradigm and the MADAG autoscale vision lies in the placement of work on the SeDs. In the implemented centralised approach, tasks are run on specific SeDs by the `AutoscaleNodeScheduler`. They use the SeD unique identifier as a permanent handle. However, the submission still goes through the MA. If a user submits a traditional remote procedure call to a service proposed by a SeD which is part of the autoscale platform, the MA might assign this task to a node that is expected to be entirely dedicated to the MADAG workload. If the MA is expected to manage both traditional calls and workflows submitted to a MADAG with auto-scaling capability, then the services of both parts of the platform should have different names. For example, a platform dedicated to linear algebra should separate the `matmult` service proposed by basic SeDs to answer DIET calls from the `autoscale-matmult` service used by workflows and proposed by the SeDs deployed upon request by the `AutoscaleManager`.

IV.4.2 Future works

Coming from the previous multi workflow schedulers, where each running task requires a dedicated thread, the `AutoscaleNodeScheduler`-based approach reorganises but does not reduce the amount of resources used at the MADAG level. This initial evolution did not significantly affect the scalability of the multi workflow scheduling, but it refactored the code around node-specific objects. Unlike the previous code, which had a blurry structure, this MADAG autoscale contains objects that can trivially be mapped to the resources they manage.

This implementation work, which gave new features to the DIET platform, is only the first step toward further evolutions of the middleware. The next step is to rewrite part of the SeD base code. Harnessing the power of the nodes, and having the SeDs being part of a decentralised scheduling process (following the multi-layer scheduling logic) would increase the scalability of the platform.

Chapter V

An applicative use-case: WASABI

In the previous chapter, we designed and built an autonomous workflow manager, from the overall structure to its actual implementation in the DIET middleware. However, all evaluations were performed using synthetic workflows. While these workflows point out the strengths and weaknesses of my approach, the next step is to evaluate the efficiency of the platform when deploying real scientific usecases.

A preliminary presentation of my work during a BioSyL[20] meetup led to discussions with researchers from the LBMC[21]. They were developing an application called WASABI and part of the application was a minimalistic workflow engine. This application felt like a good candidate for our approach, which was still unrealised at the time.

Section V.1 WASABI: Waves Analysis Based Inference

WASABI (Waves Analysis Based Inference) is a framework for the reconstruction of gene regulatory networks (GRN). Gene Regulatory Networks play an essential role in many biological processes, such as cell differentiation. Their identification has raised high expectations for understanding cell behaviours. Many computational GRN inference approaches are based on bulk expression data, and they face common issues such as data scarcity, high dimensionality or population blurring [Chai et al., 2014]. WASABI designers believe that recent high-throughput single cell expression data [Pina et al., 2012] acquired in time-series will allow to overcome these issues and give access to causality, instead of "simple" correlations, to dissect gene interactions. Causality is very important for mechanistic model inference and biological relevance because it enables the emergence of cellular decision-making. Emergent properties of a mechanistic model of a GRN should then match with multi-scale (molecular/cellular) and multi-level (single cell/population) observations. A better understanding of the structure of GRNs, which is the objective of WASABI, could provide useful incite for the development of new treatments for illnesses such as leukaemia.

WASABI is based upon the idea of an iterative inference method. It relies on a divide-and-conquer type of approach, where the complexity of the problem is broken down to a "one-gene-at-a-time" problem which is easier to solve than previous approaches and can be parallelised.

The process starts with the acquisition of in-vitro data. Cells are cultivated, and samples are extracted at different steps of the differentiation process. Gene expression (RNA dosage) is measured for each of those time steps, and the results are stored in a large matrix. That is where WASABI comes in. At each step, WASABI builds and evaluates many possible variations of the GRN and selects the ones that most accurately fit the measurements. This process is repeated in waves, hence the name WAves Analysis Based Inference.

At the beginning of the process, one would expect (and an initial assessment confirmed) that the number of suitable networks will sharply increase. In this growth phase, an efficient parallelisation is paramount. In a second phase, one expects that most of the "bad" networks will have accumulated so many errors that they will diverge from the experimental results, and it will, in the end, result in the selection of a manageable amount of networks.

[20] http://www.biosyl.org/
[21] http://www.ens-lyon.fr/LBMC/

V.1.1 WASABI workflow description

WASABI, as an application, is composed of many steps, each one divided into many instances of the same elements. The control flow, which links these many elements into the complete application, can express this iterative structure through a DAG of successive fork-join patterns. Figure 31 shows the general shape of the WASABI workflow.

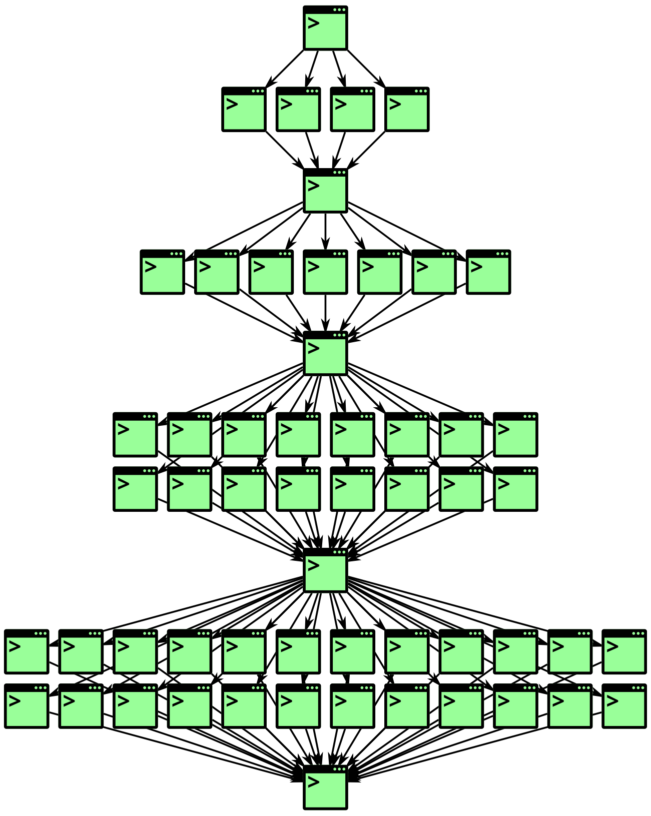


Figure 31: General shape of the WASABI workflow (succession of fork-join patterns). The real WASABI workflow we observed contained 9 steps for a total of 5309 tasks accounting to 4250.1 hours of computation.

In reality, the one we worked with contains 9 steps of size 5, 6, 36, 252, 1000, 1000, 1000, 1000, 1000. This adds up to a total 5309 tasks (including synchronization tasks) and 10598 control flow dependencies. Total runtime for all these tasks is 4250.1 hours on the IN2P3 Cmputing Center[22] workers[23]. All these information, as well as details about individual tasks (such as runtime) were extracted from traces of previous runs on the IN2P3 computers. This allowed us to build a description of WASABI as a workflow. This description was translated to the Pegasus workflow syntax that our simulator uses.

[22] https://cc.in2p3.fr/

[23] worker = one processor (core) on an Intel(R) Xeon(R) E5-26xx CPU

V.1.2 Why does WASABI need DIET and autoscalling ?

Current execution of the WASABI workflow is performed on the IN2P3 computers. This kind of large computer clusters, owned by research institutions, is handy to scientists, but also has drawbacks. The first one is the cost, which means only a few such systems are available, and access to those machines is limited. Some projects are lucky enough to have large quotas, but when they have expired, moving to another platform can be difficult.

Moreover, these platforms have specific hardware, and if an application requires additional hardware, like GPUs, it could be difficult to get access to a platform meeting the requirements.

Last but not least, if the project becomes successful enough that entities outside academia want to use it, then other platforms will be required. This is a challenge for WASABI, which could be useful to many biology researchers who do not necessarily have access to large clusters.

Thus, WASABI needs a mechanism to achieve deployment on on-demand Cloud infrastructure. Autonomous deployment would be an attractive feature of this deployment engine. In addition to these innovative features, the use of a workflow engine would also simplify the development of WASABI. The developers can focus on the WASABI logic and do not have to worry about the scheduling issues, which is not their area of expertise.

Section V.2 Running WASABI

In this section, we are going to evaluate the performances of different platform management policies with regard to the execution of the WASABI workload. Although the results presented here stem from this specific workload, they should be representative of what to expect with most scientific or industrial application that is described as a workflow.

V.2.1 Comparison to fixed deployment for a single workflow execution

In order to estimate the efficiency of our deployment, we will compare it to a more traditional “fixed deployment” method. In a fixed deployment, the user books a given amount of resources that are then used by a batch scheduler. Whenever a task is ready to run and a node is available, the task will be placed on the node to be computed. It is clear that, with this approach, the more resources there are, the faster the workflows will be executed, up to the point where the sequential dimension of the workflow prevents the scheduler from achieving any more parallelism. Still, having more resources also means that more CPU time is wasted during the synchronisation parts of the workload. Once all computations are done, the user has to release the resources.

Assessing total cost and runtime of a workflow for a specific deployment is a complex operation that requires trial and error. Simulation can help with this process, but it requires knowledge of the platform specifications, which users not familiar with computer science might not have. What our framework offers is a platform where users can specify their workflows and the required wall time for each of them. The platform is then automatically deployed to meet the expected QoS while achieving the lowest cost possible.

The results in this section have been achieved using the simulator developped earlier. Details about the WASABI workflow, including its structure, task duration time and communication details, were extracted from traces of runs on existing HPC structures (IN2P3). Estimations of the deployment cost for those simulations were obtained assuming their deployment on Amazon EC2 instances of similar performance. The down-scaling mechanism was tunned to match Amazon EC2 billing policy.

Simulated Platform	#VMs	Walltime	VM duration	Total usage	Cost	Efficiency
Fixed (100)	100	47:10:51	48h	4800 core-hours	\$110.400	88.54%
Fixed (150)	150	35:04:51	36h	5400 core-hours	\$124.200	78.71%
Fixed (200)	200	28:00:49	29h	5800 core-hours	\$133.400	73.28%
Fixed (250)	250	27:17:42	28h	7000 core-hours	\$161.000	60.72%
Fixed (300)	300	24:16:44	25h	7500 core-hours	\$172.500	56.67%
Fixed (350)	350	23:24:56	24h	8400 core-hours	\$193.200	50.60%
Fixed (400)	400	21:48:05	22h	8800 core-hours	\$202.400	48.30%
Autonomous (18h)	851	18:09:14	Variable	4915 core-hours	\$113.045	86.47%
Autonomous (20h)	711	20:08:05	Variable	4734 core-hours	\$108.882	89.78%
Autonomous (24h)	653	24:06:05	Variable	4811 core-hours	\$110.653	88.34%
Autonomous (28h)	676	28:05:46	Variable	4726 core-hours	\$108.698	89.93%
Autonomous (32h)	663	32:03:59	Variable	4676 core-hours	\$107.548	90.89%

Note: Cost were computed assuming Amazon EC2 *t2.small* instances for an on-demand price of \$0.023/hour.

Table 3: Platform statistics for different runs of a single WASABI workflow. The workflow contains 5309 tasks for a total of 4250.1 core-hours

Our first results show platform performances and costs for the deployment of a single WASABI workflow. This workflow contains 5309 tasks for a total of 4250 core-hours. Ideally, we would like to pay only for these 4250 hours of computation, but the billing policy of Cloud providers is such that we will also be charged for some unused time when we cannot perfectly use the hours allocated to each node. For example, a node used to compute a task that lasts 1 hour and 48 minutes will be billed for 2 hours, and we are thus wasting 12 minutes of CPU time. The critical path in this workflow is about 17 hours, meaning that no matter how many resources we have, we will not be able to get results faster than that using the type of node considered here.

For this experiment, we run simulations with fixed platforms of sizes varying from 100 to 400 nodes. This gives us a baseline of what current approaches achieve. The results in Table 3 and Figure 32 show a Pareto front which comes from the conflict between two contradictory objectives: maximizing platform performance and minimizing deployment cost. This confirms the idea that there is no ideal value for the size of a fixed platform. Selecting the size of a fixed platform results in a choice between performance and cost on this non-optimal front.

However, the dynamicity offered by the deployment control loop implemented in our framework leads to better results. By efficiently allocating and deallocating nodes we manage to free ourselves from this Pareto efficiency, and we achieve better results in term of deployment cost even when facing tight QoS constraints.

V.2.2 Multi-tenant/multi-workflows deployment: Example of a small lab using WASABI

In the previous section, we saw how our approach could efficiently deploy a computing platform to execute a single instance of the WASABI workflow. However, this context still requires quite many human interventions as the platform was dedicated to this workflow. This implies that, in order to

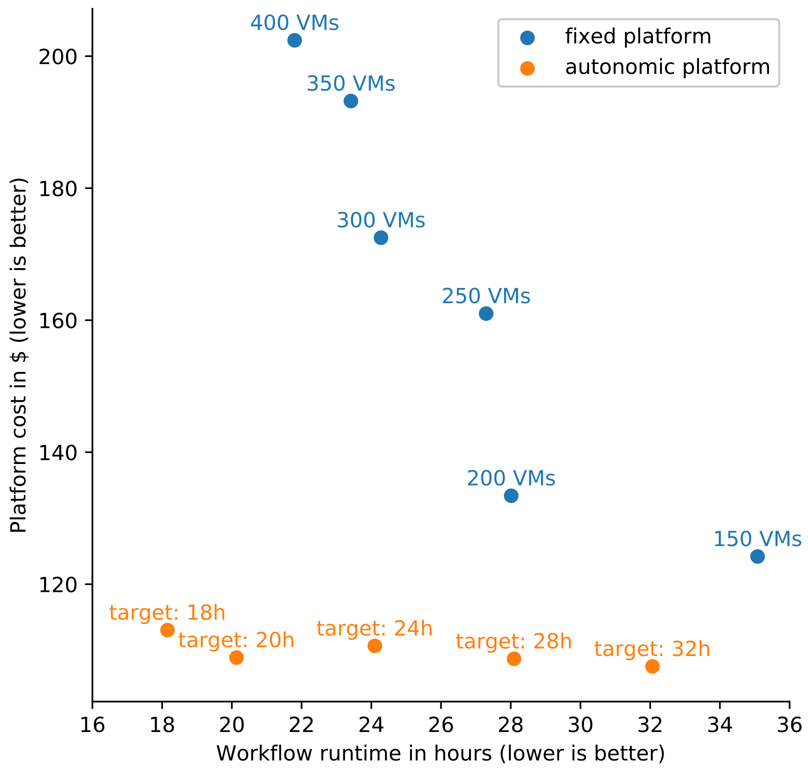


Figure 32: Cost/Makespan front for different runs of the WASABI workflow on various platforms.

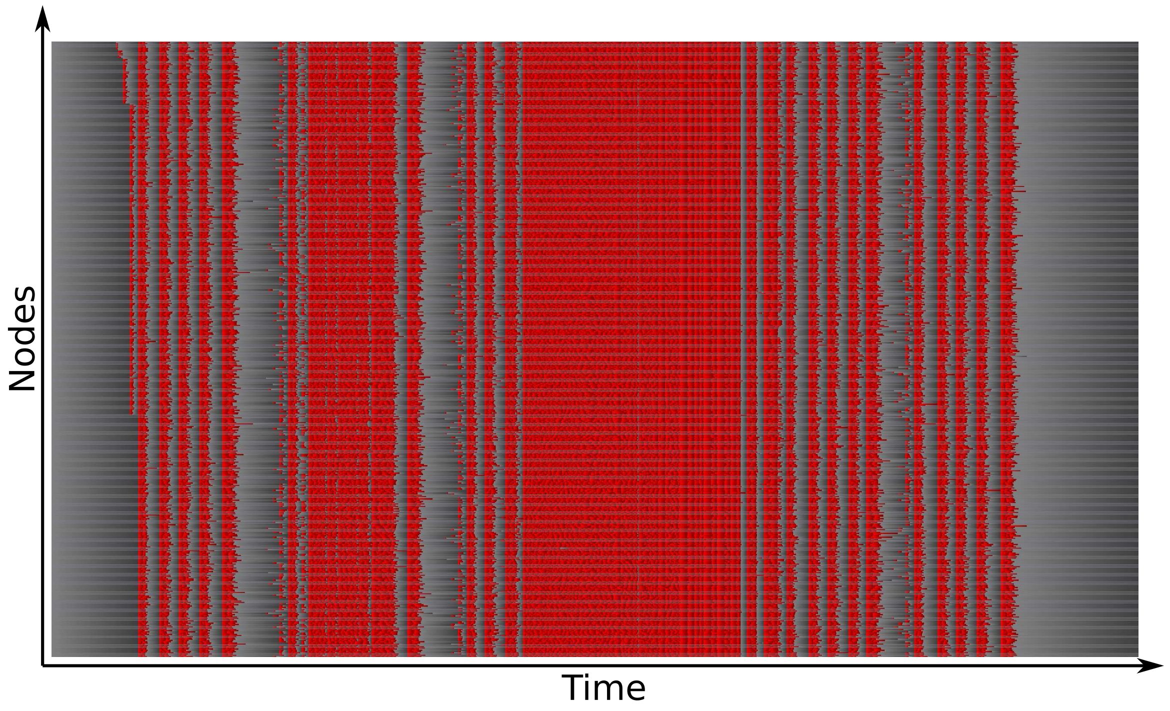
execute their workflow, the user must first deploy the platform manager. Even worse, in the case of a fixed allocation, the user has to decide how many nodes to deploy and not forget to take actions at the end of the run to shut down the platform.

We believe that the platform should be maintenance-free, and users should be able to submit workflows to an already existing platform manager. This perpetually running tool would be the entry where any user can submit their workflows. The platform would be handled automatically, allocating new nodes when needed, sharing nodes between workflows of different users and shutting nodes down when they are no longer necessary.

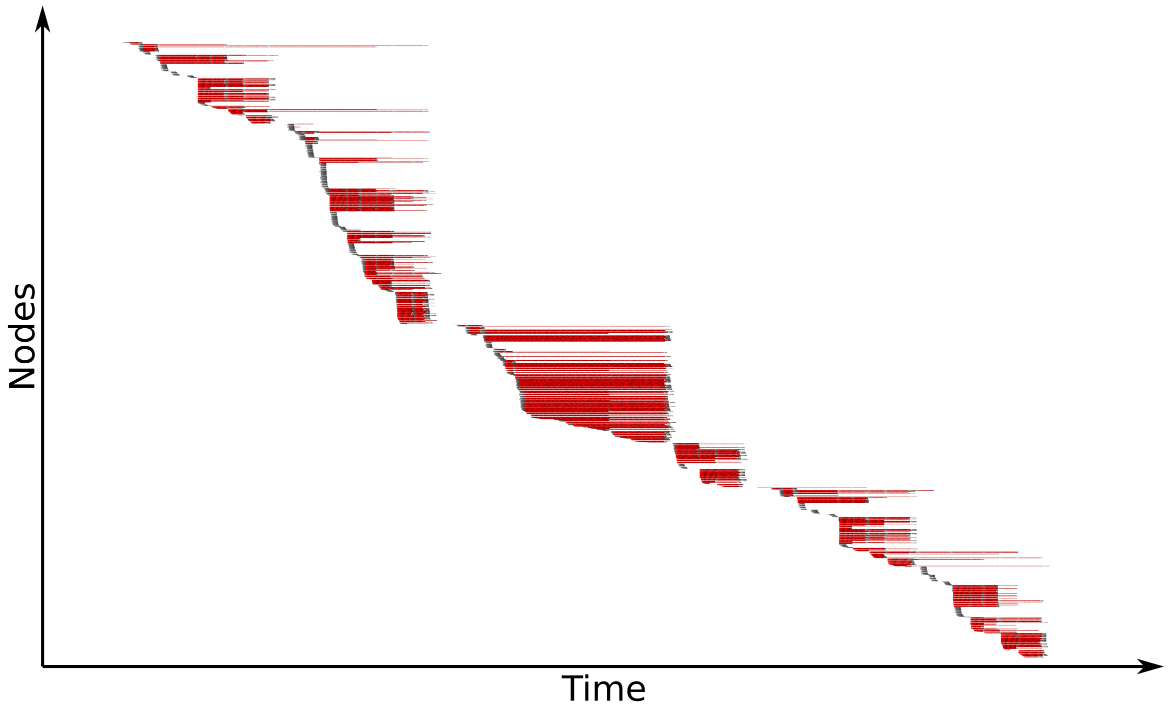
Today, sharing a computing platform, using the fixed deployment approach and a batch scheduler, is simple but very inefficient. In addition to the waste we could already witness when running a single workflow, we also have to consider all the wasted resources which are allocated during off-peak hours.

We simulated such a context at the scale of a small laboratory. In this example, we consider a research team who uses the WASABI application to analyse their results. Over a one week period, our imaginary lab produces data requiring 10 runs of WASABI. Researchers in this team submit the data for analysis when available. As such, the submissions are not distributed evenly throughout the week. In particular, we assume that all submissions will be performed during work days. While no new job is submitted during the weekend, computation can still be performed during the weekend. When using the autonomous approach, users ask the results to be available 24 hours after submission.

Statistics for this simulated workload is reported in Table 4. Gantt diagrams for both deployment



(a) Fixed platform



(b) Autonomous platform

Figure 33: Gantt diagram of VMs during the multi-tenants execution of WASABI workflows. 10 workflows are executed over a one week period, for a total of 42501.0 core-hours of computation. For each VM (line) computation time is shown in red and idle time in grey. Figure 33a shows results for fixed deployment of 500 VMs running for the full week while Figure 33b shows result for an autonomous deployment that spawned 4698 VMs over the week, some of which only ran for a single hour.

methods are shown in Figure 33. Once again, we see that the dynamic allocation of nodes has significant advantages over the previous approach.

Platform	#VMs	Fastest run	Slowest run	Total VM usage	Cost	Efficiency
Fixed	500	20:12:36	33:58:52	84000 core-hours	$1932.00	50.60%
Autonomous	4698	23:50:03	24:07:12	46563 core-hours	$1070.95	91.28%

Note: Cost were computed assuming Amazon EC2 *t2.small* instances for an on-demand price of $0.023/hour.

Table 4: Platform statistics for the multi-tenants execution of WASABI workflows. 10 workflows are executed over a 1 week period, for a total of 42501.0 core-hours of computation.

The most obvious advantage to our method is cost. Having a platform with 500 *t2.small* instances running for a whole week would cost $1932.00, with an efficiency of 50.60% with this particular workload. For the same work, our approach would reduce deployment cost by 44.57% and achieve an efficiency of 91.28%.

The second advantage to our method is the ease of maintenance resulting from the elastic deployment.

Some might argue that the fixed platform does not have the right size, and using 500 nodes is an empirical choice, and they would be right. However, as discussed previously, choosing which platform size to use is not an obvious choice. While having a 500 nodes platform leads to wasted resources, it is also not enough to ensure that sufficient resources are available to all users. During rush periods, workflows are fighting for resources. This is visible around the middle of Figure 33a when all nodes are busy. During this period, no resource is wasted, but work does not progress as users would expect, causing some workflows to finish with delays. While users would require the workflows to be computed within 24 hours, some took over 33 hours due to the limited number of resources available.

Unlike the fixed platform approach, the autonomous deployment allocates resources when needed, which limits waste during off-peak hours and guarantees that the users will have the results they expect with minimal delays. This is visible in Figure 33b. The burst in the workload causes the nodes to be allocated and deallocated in blocks. When many resources are required, the platform manager deploys many nodes. Once the work has been done, nodes start to suffer work shortage and start deallocating. We can see that the burst that caused the fixed platform to saturate is here triggering the allocation of many nodes. These nodes execute tasks from all active workflows and contribute to meeting the QoS users expect. In our example, this elastic allocation process improved delays to be at most 7 minutes 12 second.

We also notice that the efficiency achieved by our method in the multi-tenants context is slightly better than in a single workflow context. On a fixed platform, having multiple workflows causes conflicts and decreases the QoS. However, with our elastic deployment manager, having multiple workflows on the same platform is a good thing as the workflows can share nodes to maximise their use and reduce waste. QoS is maintained by the allocation of new resources when required.

Last but not least, the autonomous deployment makes the results availability more predictable. This is more comfortable for users who might require these results for specific deadlines. With a fixed platform, users are affected by each other's submissions, which might lead to frustration and conflicts.

Section V.3

Discussions

In addition to the previous results detailed in Chapter II and Chapter III, this chapter showed that the autonomic workflow engine builds during this PhD can achieve good results in the deployment of real workflows. While it is helpful in the deployment of single workflows (up to 40% gain depending on the targeted walltime), it is the multi-tenant approach that gives the most significant gains (more than 44% cost reduction as well as more predictable runtimes in our WASABI example).

The quality of service proposed by this approach relies on the ability to represent DaaS-based communications efficiently to achieve low makespan. However, it is the features of “deadline specification by the user” and “autonomous platform deployment” that seem the most appealing to both the users submitting jobs and to those paying the bill.

Nethertheless, the results shown in this chapter have all been produced through simulation. While we have a Diet implementation of the autoscaling workflow engine, some work is still required to refactor WASABI as SeD services. Large-scale experiments of WASABI on Grid’5000 using the autoscale version of Diet are planned but have not yet been conducted when writing these lines.

Chapter VI

Conclusion

Section VI.1

Contributions of this Ph.D.

Our objective when starting this work was to study the challenges relative to the execution of workflows on Cloud infrastructures. Three years later, we have answers to both theoretical and technical issues. These answers converge into an innovative middleware structure that we evaluated and implemented. While the proposed solution has limits, it is a step forward in the direction of autonomous platform management and middleware decentralisation.

In Chapter II we discussed the pre-processing of workflows with regard to the platform and communication models. We proposed a model for the estimation of communication latency in DaaS-based Cloud platforms and showed how it can be used to re-program an old clustering algorithm. This contribution produces clusters of tasks which help improve data-locality but are not commonly handled by Cloud schedulers.

In Chapter III we detailed the online part of the middleware. The proposed solution manages the execution of the clusters of tasks in a decentralised, multi-level, scheduling approach. In the meantime, the autonomous deployer manages the upscaling and downscaling mechanisms necessary to achieve platform autonomicity. All these features rely on the collaboration of many objects that have been designed to achieve decentralisation and thus scalability. Multiple algorithms that manage the synchronisation of distributed resources made it possible.

In the same chapter, we also evaluated the effectiveness of this middleware model on the execution of synthetic workflows. This not only proved that the dynamic allocation of resources is more effective than fixed deployment, but also that multi-tenant resource sharing further increases the efficiency. Gantt diagrams (Figure 22) showed the effectiveness of the upscaling and downscaling mechanisms. When scheduling sparse workloads, our approach managed to cut deployment costs by 50%.

While the decentralised infrastructure design emerged from the modelisation of DaaS-based Cloud, implementing this model in a brand new middleware would have required extensive engineering work. On the other hand, some middleware already have most components figured out and could evolve following the structure of our approach, but their design might not be compatible with the distributed nature of our solution. Chapter IV detailed the work required to implement our model into the Diet middleware and overcome the constraint of a hierarchical approach build for RPC tasks.

Finally, Chapter V focused on the execution of real scientific workflows. In addition to taking care of the burden of platform management, we showed that institutions using our approach for the management of computing resources can significantly decrease their bills (by 44% in our example) and ensure that all results are obtained within the required deadline.

All these contributions result in a complete solution that, when deployed, provides Workflow-as-a-Service capability on top of IaaS and DaaS resources. This will help biologists, physicists and other scientists who tackle the big challenges of humanity using workflows harness the computing power of Cloud solutions more efficiently.

Section VI.2

Opened discussions and future work

Our solution meets the requirements described in Chapter I. However, additional capabilities would further increase the usefulness of an autonomic workflow engine.

Scheduling, placement and need evaluation

Our approach is modular, and evolutions to the modules algorithms could improve the platform behaviour without changes to the overall structure. Future work could focus on improving the mechanisms responsible for cluster placement, tasks reordering, need evaluation, and many others.

For the high-level core scheduler, which is in charge of clusters to nodes assignment, we discussed two naive approaches (front-filling and back-filling) as well as an additional mechanism to avoid deadlocks. A more intelligent, non-greedy approach could certainly achieve must better results. Among the many things that could be considered, purposely starving some nodes might improve the downscaling mechanism.

The modular nature of the framework is such that iterative improvements of existing behaviours should be straightforward and not require any modification of the distributed structure. The evolution of the framework itself should only be necessary when considering major evolutions of the platform or workflow models.

Platform heterogeneity

Our approach is limited to homogeneous platforms, where all instances used for computation are identical. This matches the requirements of many applications that were initially designed to run on homogeneous clusters but fail to exploit the entirety of Cloud features. Adequate use of heterogeneous resources could increase the computing capability (using dedicated hardware for specific tasks) while limiting the deployment cost (using cheap, slow instances for non-urgent tasks). However, the management of heterogeneous platforms raises many difficult issues.

The static analysis would be significantly affected. As discussed in Chapter II, algorithms such as HEFT allow for the clustering of workflows on heterogeneous resources, but they fail to match the Cloud paradigm where any instance type can be deployed at any time. The workflow representation would also have to evolve in order to specify the requirements of each task. Some ideas are discussed at the end of Chapter II, but much work is still required before anyone can claim to have solved this issue.

The scheduler and platform manager would also have to evolve accordingly. A naive approach, dividing the platform into strict subsets with their own queues would be pretty straightforward. However, an heterogeneity-aware scheduling and deployment engine that could manage consolidation at runtime, and would select which instance to deploy at a given time, would require significant evolutions of both the need evaluator and the core scheduler.

Conditional workflows

Workflow support is restricted to unconditional acyclic graphs to tasks. Adding support for conditions would allow a broader audience to benefit from our progress with autonomous deployment.

Control structures like *if* statements within a workflow could be achieved by isolating the different branches, performing static analysis on each branch and having each branch's workflow enabled or cancelled by the engine. However, deadlines would be challenging to manage when the number of branches can be arbitrary high such as when dealing with *while* statements. Cross-branch optimisation would also be lacking.

Another interesting feature would be support for partial reduction. In some cases, a workflow might contain n parallel tasks, but the results of $m < n$ of these tasks could be enough to move on. There is currently no feature in our model (or in Diet's workflow model) for the support of such workflows.

Despite the limitations of our approach, the fact that we manage multi-tenant executions means that a workflow w could act as a user and submit child workflows $w'_1, w'_2, \ldots$. This could be leveraged to provide support for more dynamic workflow models with minimum evolution to the infrastructure.

Further integration

So far our solution was deployed "by hand" on the Grid'5000 testbed. While the deployment scripts make it easy for a computer scientist to start a computing platform, we are still far from a one-click solution available to anyone.

The integration of our code in a deployment tool such as TOSCA [Binz et al., 2014] would be an interesting step in this direction. Provided a Diet SeD is available with the required services, deploying a minimal platform to a commercial Cloud provider could be done in a matter of seconds. With the right bridge, the platform would then be able to grow by itself using the same deployment mechanisms.

Bibliography

[Amar et al., 2006] Amar, A., Bolze, R., Bouteiller, A., Chouhan, P. K., Chis, A., Caniou, Y., Caron, E., Dail, H., Depardon, B., Desprez, F., Gay, J.-S., Le Mahec, G., and Su, A. (2006). Diet: New developments and recent results. In et al. (Eds.), L., editor, *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006)*, number 4375 in LNCS, pages 150–170, Dresden, Germany. Springer. hal-01429979.

[Anderson, 2004] Anderson, D. P. (2004). BOINC: A system for public-resource computing and storage. *Proceedings - IEEE/ACM International Workshop on Grid Computing*, pages 4–10.

[Ayguadé et al., 2009] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418.

[Bala and Chana, 2011] Bala, A. and Chana, I. (2011). Article: A survey of various workflow scheduling algorithms in cloud environment. *IJCA Proceedings on 2nd National Conference on Information and Communication Technology*, NCICT(4):26–30.

[Balouek et al., 2012] Balouek, D., Carpen-Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. (2012). Adding Virtualization Capabilities to Grid'5000. Research Report RR-8026, INRIA. Ce rapport révisé a fait l'objet d'une publication, voir hal-00946971.

[Bharathi et al., 2008] Bharathi, S., Deelman, E., Mehta, G., Vahi, K., Chervenak, A., and Su, M.-h. (2008). Characterization of Scientific Workflows. *WORKS08*.

[Binz et al., 2014] Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Springer New York, New York, NY.

[Bonnaffoux et al., 2018] Bonnaffoux, A., Caron, E., Croubois, H., and Gandrillon, O. (2018). A cloud-aware autonomous workflow engine and its application to gene regulatory networks inference. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 509–516. INSTICC, SciTePress.

[Cappello et al., 2002] Cappello, F., Djilali, S., Fedak, G., Germain, C., Lodygensky, O., and Néri, V. (2002). XtremWeb : une plate-forme de recherche sur le calcul global et pair à pair. In Baude, F., editor, *Calcul réparti à grande échelle: Métacomputing*, number Chapitre 6. Hermes Science Publications. ISBN 274620472X.

[Caron et al., 2002] Caron, E., Combes, P., Contassot-Vivier, S., Desprez, F., Nicod, J.-M., Quinson, M., and Suter, F. (2002). A Scalable Approach to Network Enabled Servers. Research Report RR-2002-21, LIP - ENS Lyon. Also available as INRIA Research Report RR-4501.

[Caron et al., 2009] Caron, E., Desprez, F., Isnard, B., and Montagnat, J. (2009). Data Management Techniques. GWENDIA ANR-06-MDCA-009.

[Casanova et al., 2014] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. (2014). Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917.

[Chai et al., 2014] Chai, L. E., Loh, S. K., Low, S. T., Mohamad, M. S., Deris, S., and Zakaria, Z. (2014). A review on the computational approaches for gene regulatory network construction. *Comput Biol Med*, 48:55–65.

[Coffman and Graham, 1972] Coffman, E. G. and Graham, R. L. (1972). Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213.

[Couvares et al., 2007] Couvares, P., Kosar, T., and Roy, Alain and Weber, Jeff and Wenger, K. (2007). Workflow Management in Condor. *Workflows for e-Science: Scientific Workflows for Grids*, (March):1–523.

[Croubois, 2016] Croubois, H. (2016). Étude pour la conception d'une architecture autonomique et collaborative de gestion de workflow sur infrastructure dynamique. In *Compas, poster session.*

[Croubois and Caron, 2017] Croubois, H. and Caron, E. (2017). Communication aware task placement for workflow scheduling on daas-based cloud. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops PDCO (IPDPSW)*, pages 452–461.

[Deelman et al., 2005] Deelman, E., Singh, G., Su, M. H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237.

[Fahringer et al., 2005] Fahringer, T., Prodan, R., Duan, R., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.-L. L., Villazon, A., Wieczorek, M., Villaz´on, A., and Wieczorek, M. (2005). ASKALON: A Grid Application Development and Computing Environment. *GRID '05 Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131.

[Foster, 2006] Foster, I. (2006). Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513.

[Galler and Fisher, 1964] Galler, B. a. and Fisher, M. J. (1964). An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303.

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

[Gerasoulis and Yang, 1992] Gerasoulis, A. and Yang, T. (1992). A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291.

[Glatard et al., 2006] Glatard, T., Montagnat, J., Pennec, X., Emsellem, D., and Lingrand, D. (2006). MOTEUR: a data-intensive service-based workflow manager. Technical report.

[Heinis et al., 2005] Heinis, T., Pautasso, C., and Alonso, G. (2005). Design and evaluation of an autonomic workflow engine. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, volume 9, pages 2–5.

[Imbert et al., 2013] Imbert, M., Pouilloux, L., Rouzaud-Cornabas, J., Lèbre, A., and Hirofuchi, T. (2013). Using the EXECO toolkit to perform automatic and reproducible cloud experiments. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2:158–163.

[Kailasam et al., 2010] Kailasam, S., Gnanasambandam, N., Dharanipragada, J., and Sharma, N. (2010). Optimizing service level agreements for autonomic cloud bursting schedulers. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 285–294.

[Kwok and Ahmad, 1994] Kwok, Y. K. and Ahmad, I. (1994). A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 155–159.

[Lin and Wu, 2013] Lin, X. and Wu, C. Q. (2013). On scientific workflow scheduling in clouds under budget constraint. *Proceedings of the International Conference on Parallel Processing*, (October):90–99.

[Londoño-Peláez and Florez-Samur, 2013] Londoño-Peláez, J. M. and Florez-Samur, C. A. (2013). An Autonomic Auto-scaling Controller for Cloud Based Applications. *International Journal of Advanced Computer Science and Applications*, 4(9):1–6.

[Malawski et al., 2012] Malawski, M., Juve, G., Deelman, E., and Nabrzyski, J. (2012). Cost- and Deadline-constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 22:1—-22:11, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Mao and Humphrey, 2013] Mao, M. and Humphrey, M. (2013). Scaling and Scheduling to Maximize Application Performance Within Budget Constraints in Cloud Workflows. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 67–78, Washington, DC, USA. IEEE Computer Society.

[Mao et al., 2010] Mao, M., Li, J., and Humphrey, M. (2010). Cloud auto-scaling with deadline and budget constraints. *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48.

[McBride et al., 2009] McBride, J. R., Lupini, A. R., Schreuder, M. A., Smith, N. J., Pennycook, S. J., and Rosenthal, S. J. (2009). Few-layer graphene as a support film for transmission electron microscopy imaging of nanoparticles. *ACS Applied Materials and Interfaces*, 1(12):2886–2892.

[Nikravesh et al., 2015] Nikravesh, A. Y., Ajila, S. A., and Lung, C. H. (2015). Towards an Autonomic Auto-scaling Prediction System for Cloud Resource Provisioning. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, pages 35–45.

[Pandey et al., 2012] Pandey, S., Voorsluys, W., Niu, S., Khandoker, A., and Buyya, R. (2012). An autonomic cloud environment for hosting ECG data analysis services. *Future Generation Computer Systems-the International Journal of Grid Computing and Escience*, 28(1):147–154.

[Pautasso and Alonso, 2006] Pautasso, C. and Alonso, G. (2006). Parallel computing patterns for grid workflows. *2006 Workshop on Workflows in Support of Large-Scale Science, WORKS '06.*

[Pina et al., 2012] Pina, C., Fugazza, C., Tipping, A. J., Brown, J., Soneji, S., Teles, J., Peterson, C., and Enver, T. (2012). Inferring rules of lineage commitment in haematopoiesis. *Nat Cell Biol*, 14(3):287–94.

[Seymour et al., 2007] Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J. J., Group, G. W., Lee, C., Casanova, H., and Notice, C. (2007). A GridRPC Model and API for End-User Applications. *GFD-R.052, GridRPC Working Group*, pages 1–14.

[Subhlok and Vondran, 2000] Subhlok, J. and Vondran, G. (2000). Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel Distrib. Comput.*, 60(3):297–319.

[Tanaka et al., 2003] Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., and Matsuoka, S. (2003). Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51.

[Topcuouglu et al., 2002] Topcuouglu, H., Hariri, S., and Wu, M.-y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274.

[Wieczorek et al., 2009] Wieczorek, M., Hoheisel, A., and Prodan, R. (2009). Towards a General Model of the Multi-criteria Workflow Scheduling on the Grid. *Future Gener. Comput. Syst.*, 25(3):237–256.

[Wu et al., 2015] Wu, C. Q., Lin, X., Member, S., Yu, D., Xu, W., and Li, L. (2015). End-to-End Delay Minimization for Scientific Workflows in Clouds under Budget Constraint. 3(2):169–181.

[Wu and Gajski, 1990] Wu, M. Y. and Gajski, D. D. (1990). Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343.

[Yang and Gerasoulis, 1993] Yang, T. and Gerasoulis, A. (1993). List scheduling with and without communication delays. *Parallel Computing*, 19:1321–1344.

[Yang and Gerasoulis, 1994] Yang, T. and Gerasoulis, A. (1994). DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967.

[Yarkhan et al., 2007] Yarkhan, A., Dongarra, J., and Seymour, K. (2007). GridSolve: The evolution of a network enabled solver. *IFIP International Federation for Information Processing*, 239:215–224.

[Zhou et al., 2016] Zhou, N., Delaval, G., Robu, B., Rutten, E., and Mehaut, J. F. (2016). Autonomic parallelism and thread mapping control on software transactional memory. *Proceedings - 2016 IEEE International Conference on Autonomic Computing, ICAC 2016*, pages 189–198.