D3.4 Technical description of the DRIP subsystem

Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643963 (SWITCH project).

Start date of project: 01.02.2015. Duration: 36 months until 31.01.2018

Due Date:	31st July 2017
Delivery:	31st July 2017
Lead Partner:	UvA
Dissemination Level*:	PU
Type:**	R
Status:	Draft
Approved:	All partners
Version:	1.0

*Dissemination Level	
PU	Public
CI	Classified, information as referred to in Commission Decision 2001/844/EC.
CO	Confidential, only for members of the consortium (including the Commission Services)
**Type	
R	Document, report (excluding the periodic and final reports)
DEM	Demonstrator, pilot, prototype, plan designs
DEC	Websites, patents filing, press & media actions, videos, etc.
OTHER	Software, technical diagram, etc.

Contributors

Contributors	Role
Paul Martin, Cees de Laat, Zhiming Zhao	Editors
Junchao Wang, Huan Zhou, Yang Hu, Arie Taal, Spiros Koulouzis	Content contributors
Vlado Stankovski, Guadalupe Flores	Internal reviewers

Document history

Version	Date	Author	Description
0.1	30/6/17	Paul Martin	Compilation of inputs to deliverable.
0.2	4/7/17	Paul Martin	First integration pass complete. Some minor details to refine.
0.3	5/7/17	Paul Martin	Fixed mathematical typesetting and bibliography. Minor corrections to text.
0.4	6/7/17	Paul Martin, Spiros Koulouzis	Added API and data type details as appendices to document.
1.0	17/7/17	Paul Martin	Revised deliverable to address the comments of the internal reviewers.

Keywords

Cloud, programmable infrastructure, planning, infrastructure provisioning, application deployment, runtime control, time-critical applications.

Table of Contents

Executive summary

The SWITCH workbench is composed of three autonomous subsystems, each of which is primarily responsible for handling one of the three major phases of the SWITCH application lifecycle: *development*, *provisioning* and *runtime control*. The **Dynamic Real-time Infrastructure Planner** (DRIP) is the subsystem of SWITCH that handles the *provisioning* of virtual infrastructure for time-critical applications within cloud environments. In order to provision an infrastructure suitable for hosting a time-critical application however, it is necessary to produce a plan describing the topology and composition of a virtual infrastructure that can be realised using the services of either a single cloud provider or possibly a federation of providers. It is also necessary to be able to automatically retrieve and install application components on the provisioned infrastructure. The purpose of this deliverable is to describe technical aspects of the DRIP subsystem as implemented in the SWITCH public releases, building upon the design and development work described in earlier SWITCH deliverables, particularly Deliverable 3.2 "Design specification for the infrastructure planning service". It provides an updated description of the DRIP architecture and technologies, and provides in-depth descriptions of some of the key research developments that have been implemented over the past twelve months, including:

- Updated experimental evaluation of the DRIP planner for multi-deadline time-critical applications, originally specified in D3.2, now with support for the optimal configuration of software-defined networks—in particular determining the best placement of SDN controllers.
    - Experimental comparison of the DRIP planner algorithm (MEPA) with IC-PCP and CPI (see Section 3) is provided, based on the work in [Wang et al., 2017a].
    - A description of an extension to MEPA (TCPlanner), which identifies the best placement of SDN controllers, is also provided.
- A description of a dynamic service for gathering Cloud performance information, needed to unlock the full potential of QoS-aware infrastructure planning.
    - The process of provisioning and running infrastructure for testing Cloud resources is described.
    - A sample of the experiments performed is provided to better characterise the contribution of the service to SWITCH [Elzinga et al., 2017].
- An updated description of the DRIP provisioning system, focusing on multi-site provisioning that allows for the construction of 'virtual clouds' in a multi-cloud environment based on a TOSCA specification generated within DRIP.
    - The basic scheme for specifying a multi-site plan in TOSCA is described.
    - Data transfer across the Internet to provisioned infrastructure is experimentally analysed in order to better evaluate the feasibility of this kind of multi-site infrastructure [Zhou et al., 2016a, 2016b, 2016c].
- An updated description of the DRIP deployment agent, describing how DRIP can optimise the retrieval and installation of remote application components to make best use of available network bandwidth in limited time windows:
    - The deployment agent is based on the Deadline-aware Deployment System (DDS) proposed by [Hu et al., 2017].
    - DDS has been experimentally evaluated on private Cloud, demonstrating superior ability to schedule application component deployments on virtual infrastructure within a deadline over a number of common real-time scheduling algorithms.

The DRIP subsystem, along with the rest of the current SWITCH technology suite, can be found online at: https://github.com/switch-project.

1 Introduction

The SWITCH workbench consists of three subsystems, each taking primary responsibility for one of the three key parts of the time-critical application lifecycle on cloud infrastructure: *development*, *provisioning* and *adaptation*. The provisioning of a virtual infrastructure for a time-critical application requires careful planning of the host infrastructure based on a well-defined application specification. This specification must capture not only the application workflow, but also the constraints upon its components' operations, the requirements for monitoring those operations, and the *adaptability* of the application—the extent by and conditions under which the application topology can change as the application enters different 'modes' of operation. To plan for such applications, there needs to be careful selection of resources from one or more cloud providers, and these resources need to operate to the levels dictated by the applications' time-critical requirements. This requires an understanding of not only the core characteristics of the cloud resources (e.g. virtual machines) on offer, but also an understanding of the connectivity between components. Finally, in order to ensure adequate quality of service, time-critical applications on clouds need to be backed up by strong service level agreements that assert that the key characteristic properties of resources upon which planning is contingent are satisfied and maintained throughout the lifetime of a given application deployment.

The **Dynamic Real-time Infrastructure Planner (DRIP)** is responsible for the planning, validation and provisioning of the virtual infrastructure enlisted to support an application specified using the SWITCH Interactive Development Environment (SIDE). The virtual infrastructure (providing compute power, storage and network for the application) described by DRIP should implement the architecture of the application efficiently and in adherence with the constraints imposed by the developer. This infrastructure should be designed in full knowledge of the offerings provided by available cloud services, as well as the support services required to execute and manage the application at runtime. Furthermore, once DRIP has formulated an acceptable proposed infrastructure, it should automatically negotiate with cloud providers in real-time to provision the infrastructure with respect to an agreed set of service-level agreements (SLAs) that will satisfy (in principle) all quality of service (QoS) requirements. With those agreements in place and the infrastructure provisioned (which may extend beyond a single cloud), DRIP will then initialise the execution of the application and pass control over to the Autonomous System Adaptation Platform (ASAP) that will control it in tandem with the application developer.

[Laplante and Ovaska, 2011] define a real-time system as "a computer system that must satisfy bounded response-time constraints or risk severe consequences". The key notion is that of response time, the time between a system or system component receiving inputs and realising the required output behaviour. Our notion of 'time-critical application', as expressed in the original SWITCH description of work, refers specifically to distributed real-time applications that must satisfy one or more response-time constraints imposed on some subset of the application's constituent components, e.g. to respond within a certain time window to new sensor data (as in the case of the elastic disaster early warning pilot case provided by BEIA), to scale seamlessly to new users (as in the case of the unified communication platform case provided by WT), or to minimise the latency across the pipeline used to process video streams (as in the case of the cloud video studio pilot case provided by MOG). The distribution of components is of particular concern, because then the communication latency between components becomes just as important, if not more so, than the performance of the individual parts. To further compounding the challenge we face, the applications we are concerned with often have multiple overlapping response-time constraints on different parts of the application workflow. The SWITCH project must address multiple levels of deadlines on application execution, and it is the role of the DRIP subsystem within SWITCH to provision infrastructures that can guarantee sufficient performance across the entire topology of virtualised resources conscripted in a virtual infrastructure. Note that our concern is not with executing applications as quickly as possible, but with ensuring stable performance within strict boundaries in the most cost-effective manner feasible (where 'cost', particularly in private Clouds, might be measured in terms of metrics other than money, such as energy consumption).

The actual nature of individual response-time constraints varies. For example, often time constraints imposed on the acquisition, processing and publishing of real-time observations, not least in scenarios such as weather prediction or disaster early warning [Poslad et al., 2015]. The ability to handle such scenarios is predicated on the time needed for customisation of the runtime environment and the scheduling of workflows [Zhao et al., 2011], while the steering of applications during complex experiments is also temporally bounded [Evans et al. 2015]. Time constraints are imposed on the scheduling and execution of tasks that require high performance or high throughput computing (HPC/HTC), on the customisation, reservation and provisioning of suitable infrastructure, on the monitoring of runtime application and infrastructure behaviour, and on runtime controls. Failure recovery for deployed services and applications in real time is also important when supporting time-critical applications; time constraints are not only imposed on failure detection, but also on decision-making and recovery.

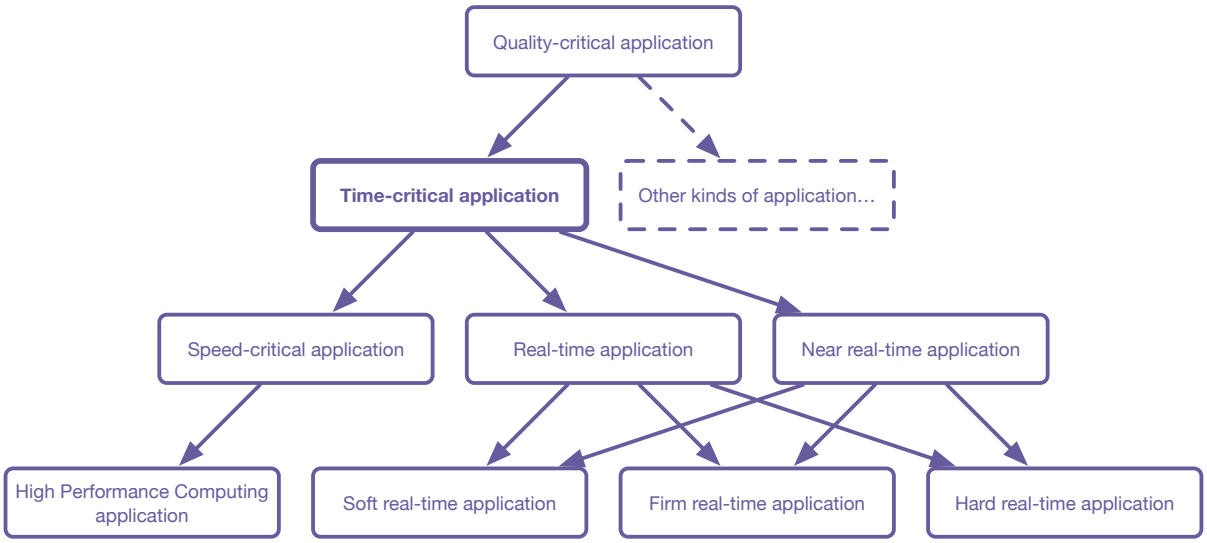

Figure 1-1 Terminologies related to time-critical applications.

Figure 1-1 defines a simple taxonomy for classifying temporal requirements. We have *speed critical* applications, where the objective is simply to minimise the completion time; these applications most suit the high-performance computing paradigm. Otherwise, *real-time* applications are often characterised by bounded response time constraints on inputs, with certain consequences upon failure to meet deadlines [Laplante and Ovaska, 2011]. Based on the impact of a system not responding on time, a real-time application is referred to as *hard* real-time when any deadline it misses leads to an immediate failure of the application, *soft* real-time when missing deadlines only leads to degradation of perceived performance, and *firm* real-time when individual missed deadlines will not lead to immediate failure, but too many misses notably will. *Nearly* real-time (NRT) applications are those with an intrinsic yet bounded delay introduced by data processing or transmission. Note that this does not make all NRT applications `soft'—such applications can still impose a hard requirement for processing to fall within the permitted bounds.

In the context in which SWITCH is mainly intended to operate, we expect most constraints to be soft or firm rather than hard. An application system where a single failure to respond within a specific time window is actively disastrous is probably not a good candidate for hosting in the Cloud, but a major objective of SWITCH is to tackle the problem of how the Cloud can be augmented with technologies such as SDN to raise the general quality of service that can be guaranteed by an application so as to make its use feasible for increasingly 'firm' applications. Thus DRIP must support a range of different kinds of deadline constraint, with varying levels of firmness (allowing DRIP to prioritise between softer and harder constraints).

In this deliverable, we report on the current implementation of the DRIP subsystem of SWITCH (Section 2), and examine in more detail the key research and development activities conducted in the previous twelve

months that have led to the realisation of the DRIP architectural design, essentially expanding on the equivalent treatment in the earlier Deliverable 3.2. These key activities can be summarised as follows:

- **The development of a QoS and SDN aware planner for multi-deadline application workflows** (**Section 3**); data-oriented workflows that capture the composition and dependencies of a time-critical application with multiple overlapping constraints on the response time of different subsets of application components. The planner uses information about the performance of specific kinds of virtual resource and their comparative running costs in order to determine the most cost-effective configuration of virtual infrastructure that will meet the deadline requirements of an application. This planner, which builds upon the state-of-the-art in critical path planning algorithms, is a key part of DRIP, able to process application specifications described in accordance with the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard for cloud applications, and also determine the optimal placement of controllers in SDN-enabled infrastructures.
- **The development of a dynamic Cloud performance measurement service** (**Section 4**) that can be used to inform and guide the DRIP infrastructure planner by testing different VM offerings against different kinds of application component, and collecting that information in order to produce the performance matrices needed by the planner to operate at its full potential. This service can gather performance information from various sources, including in principle the monitoring service provided by ASAP.
- **The development of a fast inter-Cloud provisioning service** (**Section 5**) that can partition virtual infrastructure topologies into parallel slices that can be provisioned in parallel on multiple physical infrastructures, potentially provided on different cloud platforms. This service acts as an intermediary between the application developer and one or more cloud providers, performing any necessary topological transformation whilst maintaining the critical logical dependencies between application components. Such fast provisioning may be necessary in cases where application adaptation (e.g. migration of virtual machines) or failure recovery cannot occur without significant restructuring of the infrastructure already provisioned, reducing both the time needed for such an operation, and the portion of the infrastructure that actually needs to be remodelled.
- **The design of a deployment service for fast retrieval and installation of application components from a remote repository** (**Section 6**) that can deploy application components onto planned virtual infrastructures, and provide a control interface for use by the ASAP subsystem. The SWITCH project has chosen Docker[1] as the default technology to wrap application components, because of its lightweight and efficient booting. We can also clearly see the use of containers as a trend in many different cloud projects—either provided directly on physical infrastructure or on virtual machines. To provide maximum compatibility with a range of cloud providers, we have chosen VMs as the basis for deploying Docker containers for our pilot cases.

Being able to plan compatible infrastructure topologies for a sufficient range of multi-deadline application workflows, to be able to ensure the fast provisioning of both intra- and inter-cloud virtual infrastructure, and to be able to deploy application components quickly and efficiently represent the key contributions of DRIP to the overall SWITCH project. The following section describes the implementation of DRIP that is to be found in the final SWITCH public release, which realises all the key functionalities of the DRIP design published in Deliverable 3.2.

[1] https://www.docker.com/

2 Implementation architecture and software

The DRIP subsystem is intended to provide machinery for realising a number of key actions during the lifecycle of a time-critical cloud application brokered using the SWITCH workbench: *infrastructure planning*, *infrastructure provisioning* and *application deployment*. However, DRIP also must provide a means to perform *application control* at runtime, and *resource discovery* throughout and outside of the main application lifecycle. These actions, and the core components of the DRIP subsystem that relate to them, are packaged together as a single decentralised system coordinated via a dedicated DRIP manager, as shown in Figure 2-1. On the upper level, the manager offers a RESTful API and component coordination capabilities, with the message broker validating and routing messages between components and the manager. On the lower level, the planner builds a provision plan based on specific constraints, the infrastructure provisioner interacts with different Cloud providers to offer the virtual infrastructure, and finally the deployment agent installs application components. The knowledge base stores information about Cloud services and other persistent data needed by DRIP and ASAP to function, assisted by the performance modeller, which helps DRIP build a working knowledge of current Cloud resources and their performance related to particular kinds of application component.

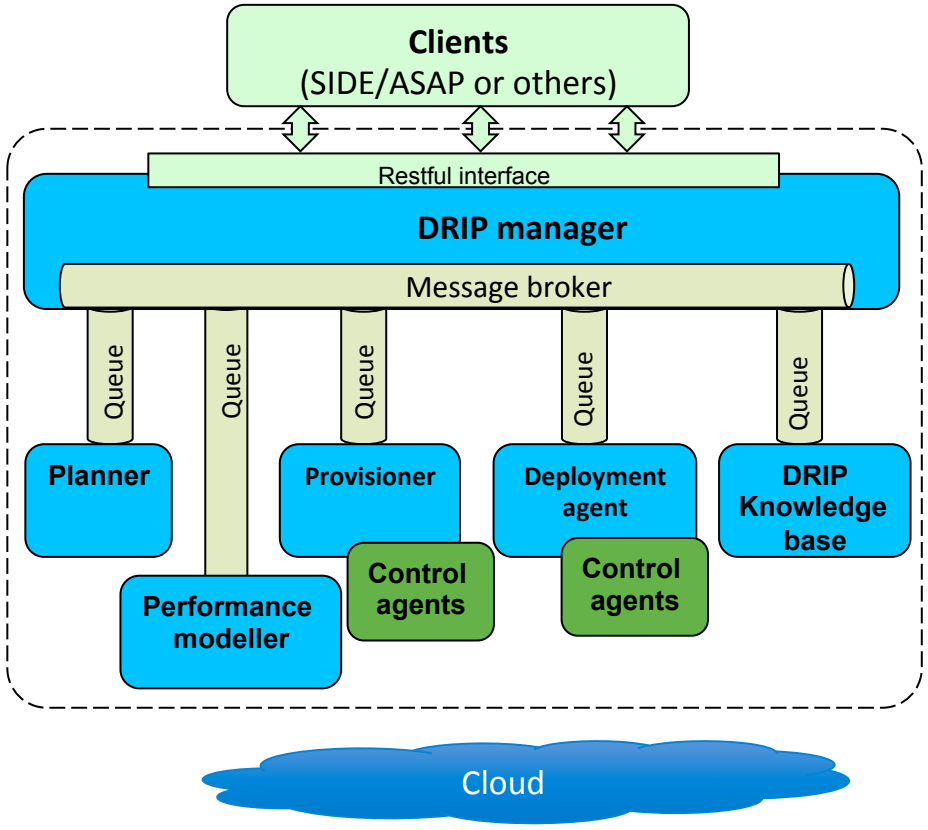

Figure 2-1 DRIP implementation architecture.

The DRIP service is made up of a number of components:

1. The **infrastructure planner** uses an adapted partial critical path algorithm to produce efficient infrastructure topologies based on application workflows and constraints by selecting cost-effective virtual machines [Wang et al., 2017a], customising the network topology among VMs, and placing network controllers for the networked VMs [Wang et al., 2017b].
2. The **performance modeller** allows for testing of different cloud resources against different kinds of application component in order to provide performance data for use by the infrastructure planner and other components inside and outside of DRIP [Elzinga et al., 2017].

3. The **infrastructure provisioner** can automate the provisioning of infrastructure plans produced by the *planner* onto underlying infrastructure services. The provisioner can decompose the infrastructure description and provision it across multiple data centres (possibly from different providers) with transparent network configuration [Zhou et al., 2016a].
4. The **deployment agent** installs application components onto provisioned infrastructure. The deployment agent is able to schedule based on network bottlenecks, and maximize the satisfaction of deployment deadlines [Hu et al., 2017].
5. The **infrastructure control agents** are a set of APIs that DRIP provides to applications to control the scaling containers or VMs and for adapting network flows. They provide access to the underlying programmability provided by the virtual infrastructures, e.g., horizontal and vertical scaling of virtual machines, by providing interfaces by which the infrastructure hosting an application can be dynamically manipulated at runtime.
6. The **DRIP manager** is implemented as a web service that allows DRIP functions to be invoked by outside clients as services. Each request is directed to the appropriate component by the manager, which is responsible for coordinating the individual components and scaling them if necessary. The manager also maintains a database containing user accounts.
7. The communication between the manager and the individual components is facilitated by a **message broker**. Message brokering is an architectural pattern for message validation, transformation and routing, helping compose asynchronous, loosely coupled applications by providing transparent communication to independent components.
8. Resource information, credentials, and application workflows are all internally managed via a **knowledge base**. It maintains the descriptions of the cloud providers, resource types, performance characteristics, and other relevant information. The knowledge base also provides an interface for these agents to look up providers, resources and runtime status data during the execution of an application.

The prototype of DRIP is based on industrial and community standards. The *infrastructure planner* is currently specified in YAML (formerly 'Yet Another Markup Language' but now 'YAML Ain't a Markup Language') in compliance with the Topology and Orchestration Specification for Cloud Applications (TOSCA)[2]. The *infrastructure provisioner* uses the Open Cloud Computing Interface (OCCI)[3] as its default provisioning interface, and currently supports the Amazon EC2[4], European Grid Initiative (EGI) FedCloud[5] and ExoGeni[6] Clouds. The *deployment agent* can deploy overlay Docker clusters using Docker Swarm or Kubernetes[7]. It may also deploy any type of customised distributed application based on Ansible playbooks[8]. The *infrastructure control agents* are set of API that DRIP provides to applications to control the infrastructure for scaling containers or VMs and adapting network flows. The *manager* provides a RESTful interface. DRIP uses the Advanced Message Queuing Protocol (AMQP) and RabbitMQ as its message broker where each process of each component is represented by a separate queue; this scalable architecture allows DRIP to be extended with additional components (e.g. planners) in order to handle larger workflows (e.g. in the case of a single DRIP service being provided to a large organisation for several applications). All DRIP software is open source; the essential characterisation of all of these components is that of independent micro-services that are able to perform their designated functions and their own reasoning autonomously, allowing for different individual implementations of components to be used by different and future configurations of the SWITCH workbench. This the DRIP components are made available as open source under the Apache License Version 2.0; the software has been containerised and can be provisioned and deployed on federated virtual infrastructures within minimal configuration. They can be obtained either via

[2] https://www.oasis-open.org/committees/tosca/
[3] http://occi-wg.org/
[4] https://aws.amazon.com/cn/ec2
[5] https://www.egi.eu/federation/egi-federated-cloud/
[6] http://www.exogeni.net/
[7] https://kubernetes.io/
[8] https://www.ansible.com/

the SWITCH release repository at https://github.com/switch-project or directly via the DRIP development repository at https://github.com/QCAPI-DRIP.

Figure 2-2 shows the basic process of using DRIP as a sequence diagram.

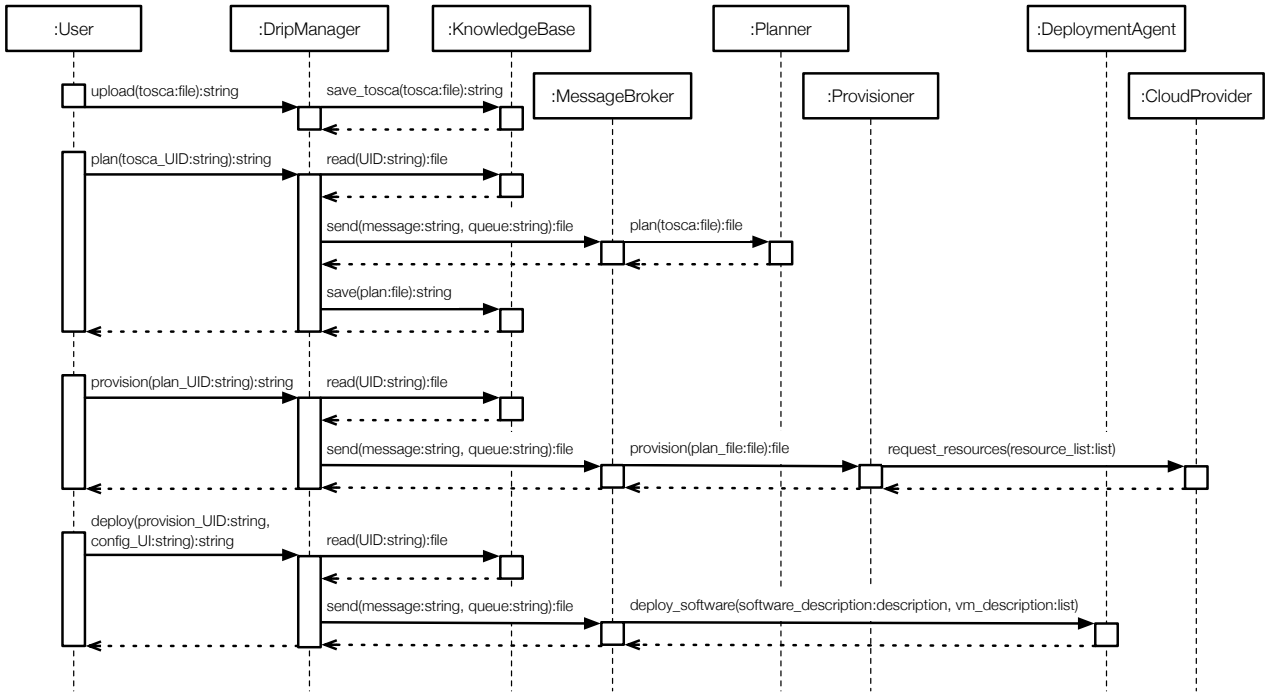

Figure 2-2 Sequence diagram describing how DRIP plans and provisions virtual infrastructure and how it deploys software.

The first step to obtain and manage a virtual infrastructure is to create an abstract definition of that infrastructure as a TOSCA description; this is a YAML structure text file describing the application workflow and characteristics, as provided for example by the SIDE subsystem of SWITCH (for more detail about TOSCA and its use by SWITCH, see Deliverable 2.4 "Concept description for application-infrastructure co-programming").

```
proxy_transcoder:
   type: switch.nodes.softwarecomponent.proxy_transcoder
   capability: proxy_transcoder
   properties:
      publish_ports: {get_input: proxy_trans_port}
   artifacts:
      docker_image:
         file: proxy_transcoder
         type: switch.artifacts.docker
         repo: SWTICH_MOG_Docker_Hub
   interfaces:
      standard:
         create:
            implementation: install.sh
         configuration:
            implementation: config.sh
```

```
            input:
               proxy_codec_profile: {get_intput: codec_profile}
      requirements:
            host:
               node_filter:
               capability:
                  host:
                     properties:
                        mem_size: 8GB
```

Figure 2-3 Sample of TOSCA-compliant YAML used by DRIP for planning/provisioning.

This description may contain network requirements such as desired bandwidth or network topology. For example a user may need a cluster with a private network. As soon as the user has specified the TOSCA description it can uploaded to DRIP via a POST request. At this point the TOSCA description is saved on the user's account under a unique ID. Next, the user may request a concrete plan from DRIP. This can be achieved by sending a GET request to DRIP containing the TOSCA description ID. The manager will direct the request to the planner which will generate a plan and return the plan's unique ID to the user. The generated plan will be further used by the provisioner to realise the virtual infrastructure. The provisioner will use the plan along with the necessary cloud credentials stored in the manager to request resources from one or more cloud providers. Finally, the deployment agent can use a description of the provisioned virtual infrastructure to deploy application components. The internal activities of the components follow the logical sequences already described in Deliverable 3.2.

The main interface by which external actors (such as the SIDE client) can invoke DRIP is provided by the DRIP manager. The control agent, of which multiple instances might be deployed alongside an application at runtime, provides an API for invoking operations on an application or its host infrastructure, which is not shown at this level of abstraction; however it can be seen that the control agent can itself invoke the DRIP manager when required to perform adaptations of a live application that require the use of other DRIP components, for example to deploy additional application components or to re-plan an infrastructure entirely (though the general idea in SWITCH is to minimise such drastic actions by ensuring that the original infrastructure planned and provisioned for an application has sufficient flexibility to handle adaptations *without* the need for this, it is nonetheless important to have such capability when it is unavoidable). A full specification of the API can be found at https://github.com/QCAPI-DRIP, with the current version also provided as Appendix A in this document.

In the following sections, we describe the research and innovation that has been carried out in the last year, focusing on each of the key components of the DRIP system and surveying their design, implementation and experimental evaluation.

3 Infrastructure planning for SWITCH applications

Executing time-critical applications within cloud environments while satisfying execution deadlines and response time requirements is challenging due to the difficulty of securing guaranteed performance from the underlying virtual infrastructure. Cost-effective solutions for hosting such applications in the Cloud require careful selection of cloud resources and efficient scheduling of tasks. Existing solutions for provisioning infrastructures for time constrained applications are typically based on a single global deadline. Many time-critical applications however have multiple internal time constraints when responding to new input. In this section we review the cloud infrastructure planning algorithm originally presented in D3.2 and further detailed by Wang et al. [2017a] that accounts for multiple overlapping internal deadlines on sets of tasks within an application workflow. In order to better compare with existing work, we adapted the IC-PCP algorithm of Abrishami et al. [2013] (see below) and then compared it with our own algorithm using a large set of workflows generated at different scales with different execution profiles and deadlines. Our results show that the proposed algorithm can satisfy all overlapping deadline constraints where possible given the resources available, and do so with consistently lower host cost in comparison with IC-PCP. Since D3.2, additional research and development has been carried out into accounting for and supporting software-defined networking (SDN), with the automatic selection and deployment of SDN controllers to augment data transfer between VMs provisioned for a given time-critical application. Thus we will provide an updated summary of the core algorithm originally presented in D3.2, and then provide a more detailed examination of the additional support prototyped for SDN controller placement.

The research and development results of this section have been published in international journal of Future Generation Computer System [Wang, 2017a] and IEEE IM [Wang, 2017b].

3.1 Infrastructure planning review

Deelman et al. [2009] provide a survey of the different kinds of workflow found in the e-science domain. Based on this, we can classify workflows deployed on virtual infrastructure into two basic categories: scientific workflows and service workflows. Scientific workflows are workflows in which each task is executed once and the virtual resource on which the task is deployed is released upon completion of the task and all following communication between the task and its successors. Service workflows are those with tasks that can be regarded as persistent services, where the tasks persist until the whole application is completed, and have to continue to respond to new inputs for the entire duration of the application. Workflows in both categories may exhibit multiple deadlines, but our concern is with the latter kind of workflow, which are often used for time-critical applications in (for example) environmental monitoring. UrbanFlood [Krzhizhanovskaya et al. 2011] is an example of an early warning system that tries to solve the problem of flood control, while Kosukhin et al. [2015] presents an architecture for performing extreme metocean event forecasting on cloud platforms. In the case of the UrbanFlood system, the workflow has multiple stages with separate modules for sensor monitoring, AI anomaly detection, reliability analysis, breach simulation, virtual dikes, and decision support. Such a system can have multiple internal deadlines in order to ensure timely responses, especially if individual modules must report to other external systems; however the quality of service is not addressed by Krzhizhanovskaya et al. when planning the infrastructure for the application.

Allocating and scheduling cloud resources for application workflows has become increasingly important for both the cloud provider and application developer, and so there are now many scheduling algorithms available to determine the amount and type of virtual machines needed to execute such workflows at minimal cost. To the best of our knowledge however, all this work addresses the problem of planning infrastructures for workflows that have a single global single deadline, rather than multiple internal dead-lines, which is our main concern within the context of SWITCH.

There exist a number of works that focus on optimal resource assignment on virtual infrastructure under different conditions and assumptions. Yu et al. [2005] propose a method to minimise the execution cost of a workflow to satisfy a global deadline. Their method first clusters the sequential tasks that have only one

parent and child together and assigns each task with a sub-deadline based on its minimum processing time and the sub-deadlines of its predecessor. Each task is then assigned to the least expensive virtual machine (VM) that can meet the deadline—however, the communication cost between tasks is not considered, nor the presence of multiple deadlines. The Infrastructure-as-a- Service Cloud Partial Critical Paths (IC-PCP) algorithm [Abrishami et al., 2013] calculates partial critical paths through the application workflow in order to schedule the deployment of tasks on the cloud in order to solve the same problem. IC-PCP can be combined with the approach taken by Yu et al.; after finding a partial critical path, each task in the path is assigned a sub-deadline with the execution time in proportion to the whole partial critical path length. The tasks in the workflow are then assigned to the cheapest VMs that still meet those deadlines. Though originally formulated to meet a global deadline, support for additional internal deadlines in IC-PCP can be added by overriding the generated sub-deadlines with pre-defined deadlines where the latter are more strict.

The planner we have developed is based on the IC-PCP algorithm, but we make a number of assumptions different from those made by Abrishami et al. For one, we assume that after one task transfers its results to all its successors, the VM where the task is deployed is not released, and instead the task will act as a persistent service waiting for more input—thus the deadline for a given task must be satisfied every time the task receives new input. We also make the assumption that every task in the workflow will be deployed on its own VM, both for simplicity, and because sharing VMs impacts the performance of tasks [Cai et al., 2016], and our focus is on time-critical applications. Most importantly, we assume that workflows can have multiple internal deadlines on different processes based on the requirements of users or downstream services.

Our **Multi-dEadline workflow Planning Algorithm (MEPA)** uses a ‘compress-relax’ method. VMs with best performance are assigned to tasks so that the ‘makespan’ (total execution time) is ‘compressed’ and all deadlines are met if possible; the assignment over the workflow is then ‘relaxed’ by re-assigning to tasks less powerful VMs albeit with lower cost while preserving deadline satisfaction. Initially MEPA assigns each task in the workflow with the best performing VM to guarantee a basic solution; if not all deadlines can be met this way, then an alternative infrastructure will be needed, or else the QoS requirements of the application will need to be relaxed. Based on the initial ‘compressed’ assignment, MEPA then calculates the earliest start time (EST), earliest finish time (EFT) and latest finish time (LFT) for each task based on the dependencies between tasks and necessary communication costs. MEPA then works backwards from the final tasks of the workflow to assign the internal deadlines; if the deadline on a task is stricter than the calculated LFT for that task, then the deadline simply replaces the LFT. If the EFT for a task exceeds its LFT, then the currently available resources cannot satisfy the time constraints on the workflow. Once the constraints on a critical path have been determined, it is then possible to determine the best assignment of VM type to each node on the path, as illustrated by Figure 3-1.

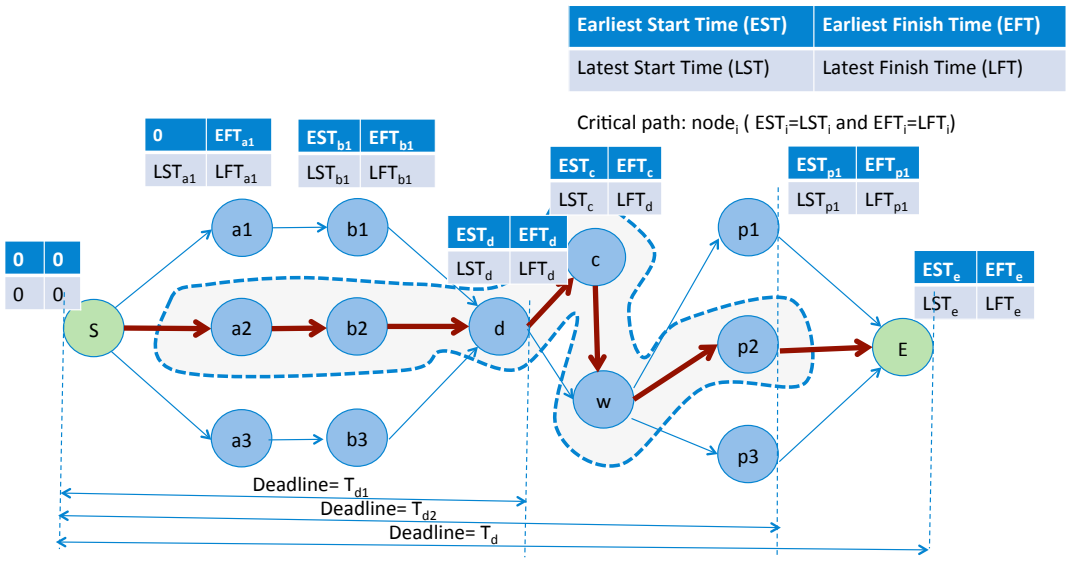

Figure 3-1 Example of deadline-aware planning by DRIP. The blue nodes represent the workflow, with the critical path outlined. For each parallel group of nodes, the earliest/latest start/finish times can be extracted.

Actual assignment of different kinds of VM to different nodes in the same workflow can be based on brute-force calculations, or based on the use of heuristics. Convolbo and Chou [2016] propose a heuristic approach which exploits the parallel properties of the workflow to minimise execution time. Rodriguez and Buyya [2014] apply particle swarm optimisation, encoding within each particle a task-resource mapping. Heterogeneous Earliest Finish Time (HEFT) has been proved to perform better than other heuristics in robustness and schedule length [Canon et al., 2008], and Multi-Objective HEFT extends HEFT to optimise the trade-off between monetary cost and the makespan of the workflow [Durillo and Prodan, 2014], though again the communication cost is not addressed. The critical path based iterative heuristic (CPI) [Cai et al., 2013] and multiple complete critical paths heuristic (CPIS) [Cai et al., 2016] are used in other algorithms for solving the infrastructure planning problem within the bounds of a single deadline. Based on the calculated earliest finish time and latest finish time of individual tasks, CPI identifies a complete critical path through the application workflow from start to finish and assigns the tasks in the critical path to VM services. In CPIS, a graph labelling method is applied to construct complete critical paths of the kind generated by CPI.

In our case, VM types are assigned to the constructed partial critical path using a genetic algorithm and a matrix of execution costs per task per VM type (which can be based on historical observation or extrapolation). Our Genetic Algorithm based Planning Algorithm (GAPA) runs for a set number of generations to find the best combination of assignments to nodes on a critical path that fulfil all deadlines. After assignment, the tasks in the critical path are tagged as assigned and the EST, EFT and LFT of the other tasks in the workflow are updated accordingly. Assignment of the remaining tasks will then continue until all the tasks in the workflow are assigned. A full description of the algorithms and logic involved can be found in [Wang et al., 2017a].

3.2 Infrastructure planner evaluation

In principle, IC-PCP can be adapted to plan the kind of service-based workflows for which DRIP is specialised for by using the sum cost of VMs per time unit as the metric for measuring whether one assignment is cheaper than the other and forbidding multiple tasks from being assigned to the same VM instance (deemed necessary to maintain good time-critical performance for application containers). By changing the calculation on the latest finishing time to take into account internal deadlines, a minimally modified variant of IC-PCP (which we will refer to as IC-PCP*) can plan for workflows with multiple deadlines. What we found however was that this approach still incurs unnecessary cost—it is possible to drive the cost down further than such a minimal adaptation of IC-PCP permits in many cases. In this section,

we describe how MEPA compares experimentally against IC-PCP* using a wide range of randomly generated workflows meant to represent the full variety of application workflows for which DRIP might be used.

Our implementation of MEPA for DRIP is based on Python 2.7.10. We use NetworkX[9] (version 1.10) to manage the workflow and PyDOT2[10] (version 1.0.33) to parse the graphs generated by GGen [Cordeiro et al., 2010]. NetworkX is a powerful Python library for manipulating complex networks. GGen is an open source random graph generator integrating several different random graph generating algorithms. The generated random DAGs (Directed Acyclic Graphs) are represented in DOT, which is a plain text graph description language. DEAP (Distributed Evolutionary Algorithms in Python) [Fortin et al., 2012] is a framework for experimenting with evolutionary algorithms such as genetic algorithms and particle swarm optimisation. For our experiments we use DEAP as the underlying framework for implementing GAPA. We conduct our experiment on the Distributed ASCI Supercomputer 5 (DAS-5)[11].

3.2.1 Workload generation

To investigate the behaviour of our algorithm, we use the graph generator GGen to generate random workflow typologies with different time constraints. Specifically, we apply 'fan-in/fan-out' methods to generate DAGs, which are widely used in random graph generation. This graph generation method takes three parameters: the number of vertices, the maximum in-degree of each node and the maximum out-degree of each node. This kind of graph generation method will generate a graph topology with all tasks' in-degrees and out-degrees within the chosen upper bounds. If the in-degree and out-degree is set to be one, then the DAG becomes a sequential graph. In order to test how our solutions perform on different scales of graph, we set the number of vertices in the workflows to range from 2^0 to 2^8. The in-degree and out-degree are used to generate DAGs with different shapes and we set the maximum in-degree and out-degree to range from 1 to 5 and 1 to 4 respectively.

For each DAG we need to generate an execution profile. The execution profile includes the performance of tasks on different VM services as well as the communication costs between tasks, which ranges from 1 to 200. In many time-critical applications, the performance of tasks on different services varies for each task. The response time of tasks may be less than or greater than the communication cost depending on the nature of computation being performed and the quality of the network. So in order to make the data more realistic, we should randomly generate response times for tasks running on different VM services that can both exceed and be significantly less than communication times. For each task we first generate the execution cost of the task on the 'best' VM service, randomly selecting a response time between 1 and half of the communication cost upper bound. The execution costs of the task on the other 'lesser' services are generated iteratively by increasing the previously generated cost by a randomised proportion. In order to simulate better different kinds of real world application, our performance generation method ensures that the performance of each task on different VM types can be substantially larger than the communication cost or much smaller, ensuring greater diversity in the workflows generated and removing any implicit assumption about the relative cost of computation versus communication.

The time constraints attached to a workflow are also randomly generated. The number of time constraints are set with a proportion to the scale of the workflow. Specifically, we set the number of time constraints per workflow to be $\lceil 0.1 \times |V| \rceil$, where $|V|$ is the number of tasks in the workflow. We then randomly select $\lceil 0.1 \times |V| \rceil$ tasks from the workflow (with the exception of the last task, which is always the final task in the workflow), and for each task we attach a random deadline based on the critical path calculation performed during workflow generation, limiting each deadline's range based on best and worst performing VM services so as to ensure no 'impossible' (or far too easy) deadlines are set. The final task will always receive a

[9] https://networkx.github.io/
[10] https://pypi.python.org/pypi/pydot2/1.0.33
[11] http://www.cs.vu.nl/das5/

deadline, which will serve as the global deadline for the entire application. All datasets generated for the experiments in this paper are available online[12].

3.2.2 Comparison of path assignment with IC-PCP and GAPA

The partial critical path can be seen as a sequential workflow, each task of which has only one predecessor and successor except for the entry and exit tasks. We therefore use a set of sequential workflows to test the performance of GAPA. We set the scale of the sequential workflow ranging from 10 to 100 and the proportion of deadlines is set to be 0.1. The performance matrix of the tasks in the critical path is generated randomly as described in Section 5. Considering the performance fluctuation, for the generated performance profile, we set the task performance fluctuation rate to be 10% and communication fluctuation rate to be 15%. The multiple time constraints of the workflow are generated as described in Section 5. We set the final generation in GAPA to be equal to the length of the partial critical path. In GAPA, we set the mutation rate to be 0.05 and population size to be 300. We assume the cloud offers three different types of VM services and set the price for each service as 5, 2 and 1, which is the same as used in [Abrishami et al., 2013].

In IC-PCP, the whole partial critical path is assigned with the same VM type. Thus, for each workflow, there are three path assigning choices, assigning all the tasks to s_1, s_2 or s_3. We compare the cost with GAPA and the cost with IC-PCP by assigning all the tasks to the VM service with best performance. We do this because other VM service selections usually violate one or more deadlines; based on our experiment involving 90 partial critical paths of incrementally increasing length, assigning the second best type of VM leads to a valid solution only 4 times in 90.

Experimental results are shown in Figure 3-2. The cost of assigning the best performing VM service to all tasks follows a linear incremental trend because the assigned VMs are of the same type. The result gotten from GAPA varies a lot because the deadline is randomly generated and the solution obtained from GAPA may not assign all the tasks to the same VM type at different path lengths. We can see that the assignment with GAPA appears to perform consistently better than when simply assigning all tasks on the VM with the best performance. We also find that GAPA can save up to almost two thirds of the IC-PCP cost in this experiment.

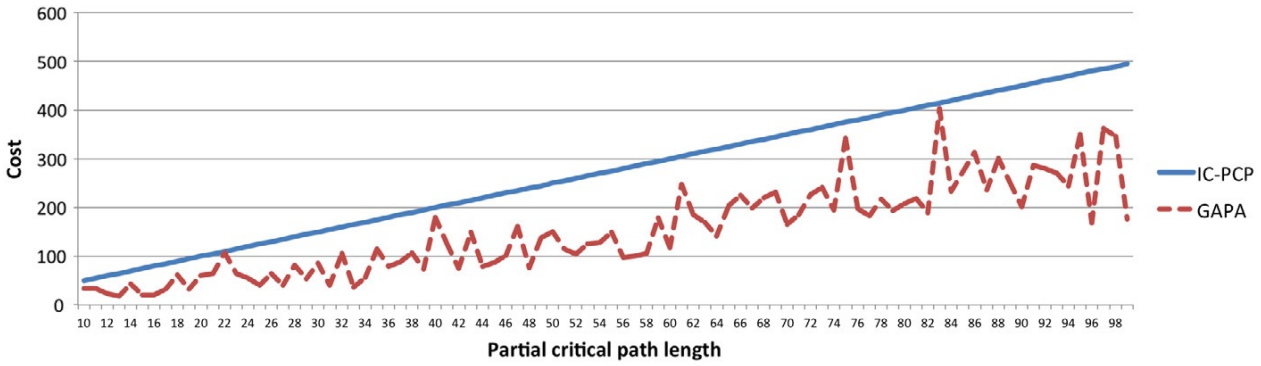

Figure 3-2 Cost comparison of path assignment in IC-PCP and GAPA.

3.2.3 Comparison of IC-PCP, CPI and MEPA with a single deadline

MEPA can plan virtual infrastructures for applications with multiple deadlines, but to compare it with existing planning approaches that only support a single global deadline, we have conducted experiments on the dataset described in Section 3.2.1, applying a single deadline and comparing the results of MEPA with IC-PCP and CPI. Figures 3-2, 3-3 and 3-4 show the planned virtual infrastructure cost of solutions produced by MEPA, IC-PCP and CPI with test workflows with 16, 32 and 64 tasks respectively, in each case varying both in-degree and out-degree (identified along the x-axis with the in-degree above the out-degree) to ascertain how connectivity influences results.

[12] https://github.com/WorkflowPlanning/workload

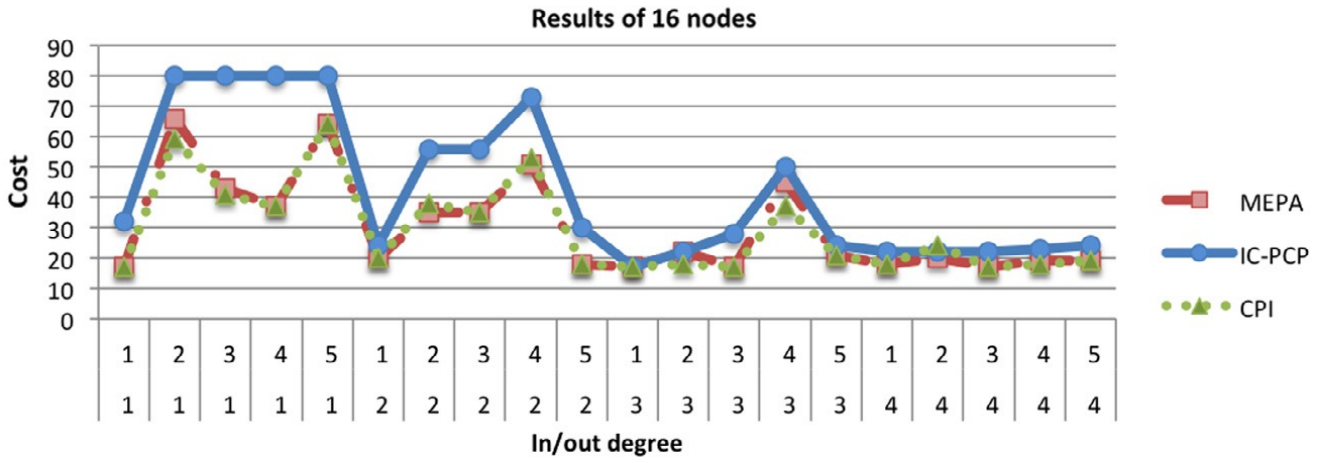

Figure 3-3 Results of 16 nodes with IC-PCP, CPI and MEPA of single deadline.

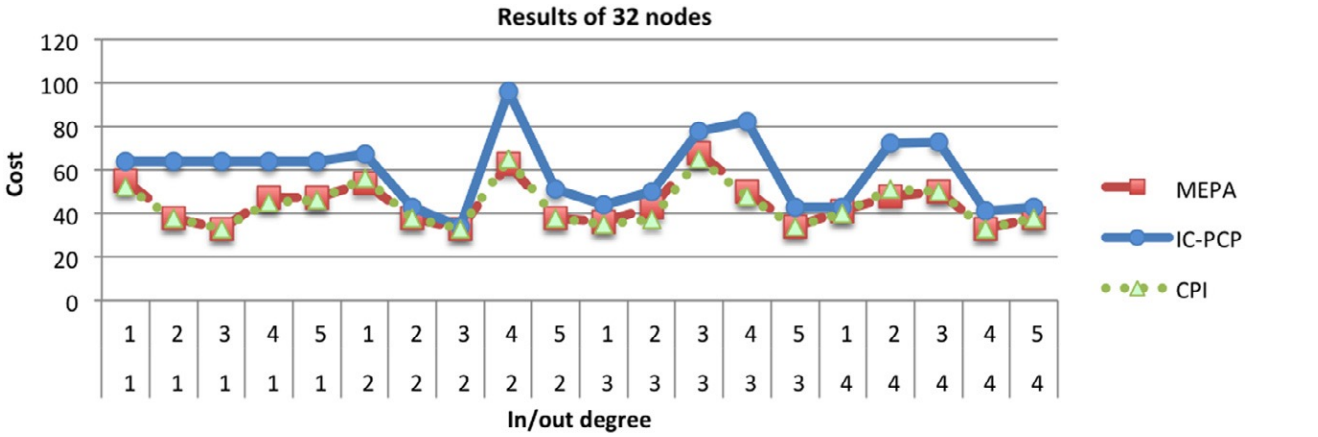

Figure 3-4 Results of 32 nodes with IC-PCP, CPI and MEPA of single deadline.

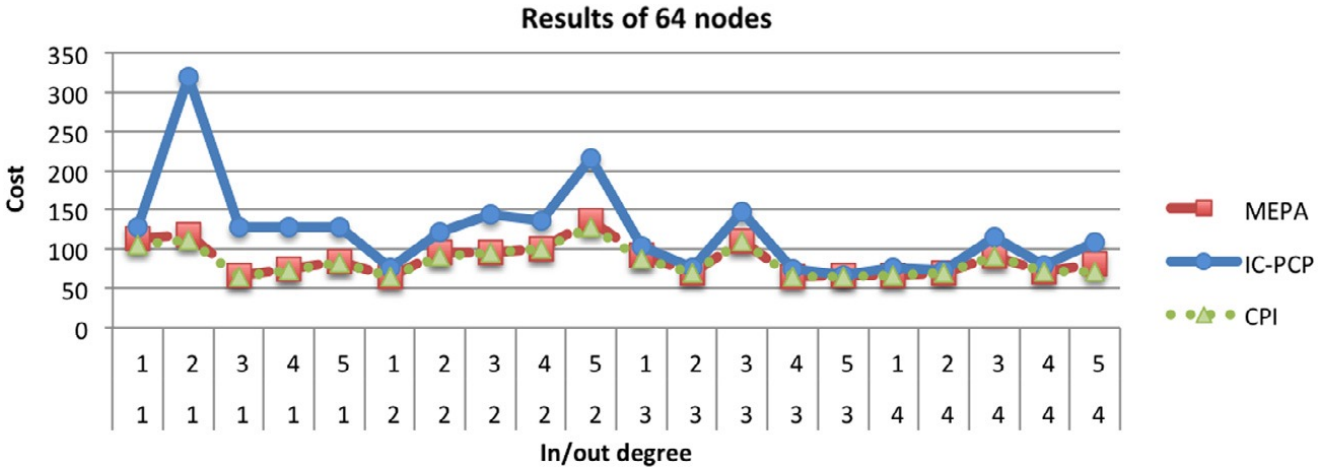

Figure 3-5 Results of 64 nodes with IC-PCP, CPI and MEPA of single deadline.

From the figures we can see that MEPA generally leads to less expensive VM assignments than IC-PCP. MEPA can even save around 66% of cost compared with IC-PCP in some cases. Although from the results we can see that MEPA and CPI give solutions with similar costs, the time complexity of the CPI is $O(N^3D^2M)$ due to the assignment of the path with dynamic programming to find the Pareto assignment [Cai et al., 2013], making it hard to scale when the deadline of the workflow is very large. N represents the number of nodes. M represents the number of services and D represents the global deadline. In our solution,

there is no such bottleneck with the scale of the deadline. Moreover, we can see that for a workflow with the same number of nodes, the cost of MEPA and IC-PCP, CPI are quite close. The reason for this is that when the in/out degree increase, the DAG will become "wider", making the length of the critical path become shorter. For a "loose" deadline, the path assignment of MEPA and IC-PCP can lead to similar solutions, leading to similar total cost. Moreover, it is not hard to see that when the scale of the workflow increases, the differentiation between MEPA and IC-PCP will become more significant.

3.2.4 Comparison of IC-PCP* and MEPA with multiple deadlines

In this part we take the workload described in Section 5 and feed the workload into both IC-PCP* (the minimal modification of IC-PCP for multiple deadline workflows described in Section 3) and MEPA. Figures 3-6, 3-7 and 3-8 compare the results of IC-PCP* and MEPA with workflows of size of 16, 32 and 64 respectively with different in-degrees and out-degrees (identified along the x-axis with the in-degree above the out-degree in all figures).

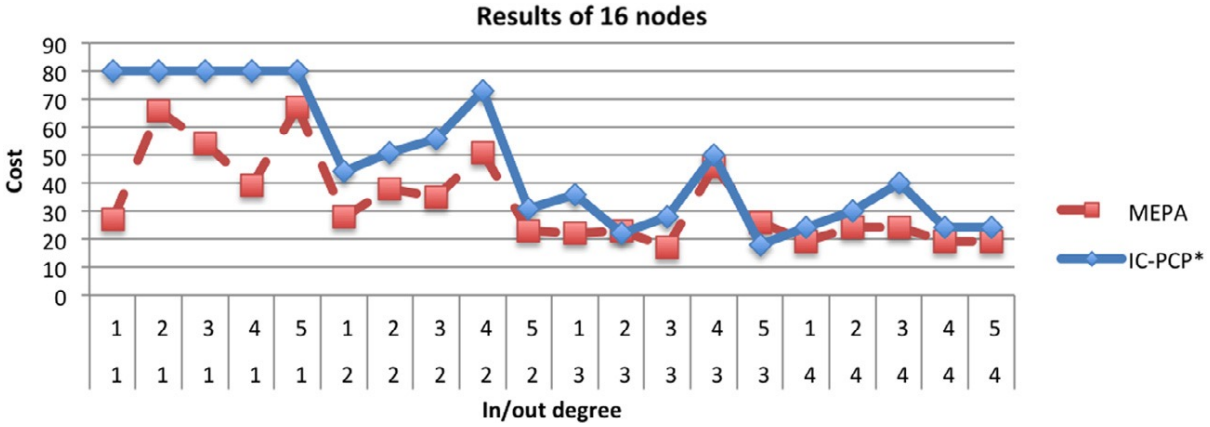

Figure 3-6 Results of 16 nodes with IC-PCP* and MEPA of multiple deadlines.

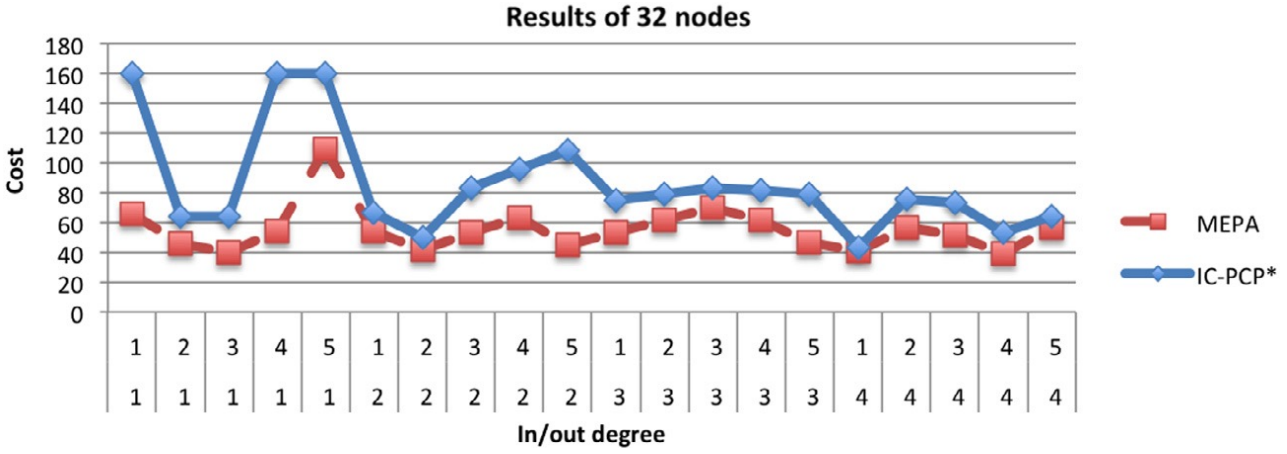

Figure 3-7 Results of 32 nodes with IC-PCP* and MEPA of multiple deadlines.

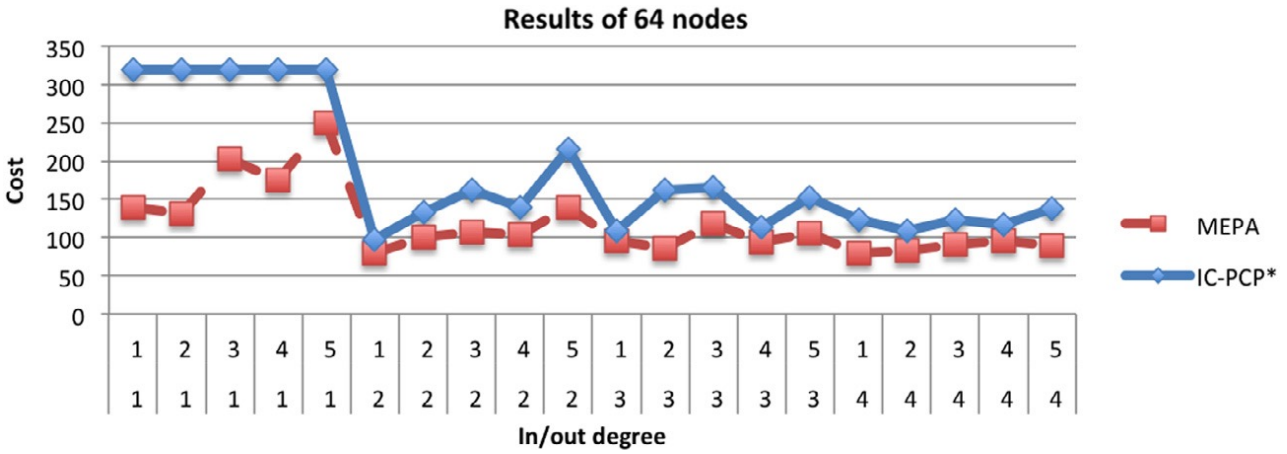

Figure 3-8 Results of 64 nodes with IC-PCP* and MEPA of multiple deadlines.

We can see from the results that MEPA is able to give cheaper solutions than IC-PCP*. When the scale of the workflow increases, the differentiation between the results of MEPA and IC-PCP* will become more significant. With the increase of in/out degree, the result of MEPA and IC-PCP* tend to become similar, but the similarity is reached for larger in/out degree ratios when the number of tasks in the workflow increases. This is because the length of the partial critical path will become shorter when the in/out degree increases. So the internal deadlines in each partial critical path can be less, making the planned results more similar.

The infrastructure planning problem has not been widely discussed from the networking perspective however, so we now intend to study SDN technologies to better enable network-aware workflow planning.

3.3 QoS-aware virtual SDN network planning review

By decoupling the control plane from the data plane, Software-Defined Networking (SDN) technologies allow administrators or applications to manipulate the underlying network behaviour via open interfaces [Kreutz et al., 2015; Ongaro et al., 2015]. SDN-based standards, e.g. Network Service Interface (NSI) and Openflow, have shown great impact on dynamic provisioning and reconfiguration in lightpaths and data centre networks in physical infrastructures [Kreutz et al., 2015]. In cloud environments, SDN-based virtual switches (e.g. implemented using Open vSwitch[13]) can be used together with networked virtual machines (VMs) to allow applications to dynamically adjust network flows via open interfaces in order to maintain the system-level performance [Jeong and Figueiredo, 2016].

At an abstract level, designing a topology of virtual network devices and placing suitable number of controllers are two key issues of designing a SDN network in a Cloud environment. When an application is distributed and has a high quality requirement such as on communication latency, designing a suitable SDN network can be difficult. Mapping application-level quality constraints onto network-level properties, e.g. topology, is not straightforward, in particular where the application has different requirements to be considered. The design of the virtual network should also take into account non-functional requirements, such as cost and reliability. To make the virtual network software-definable, one or more controllers are needed, and they can reside in the same VM with the virtual network devices or on a separate VM. Unnecessarily high numbers of controllers can not only make the resource cost high, but also increase the control complexity. Moreover, the placement of the controllers can also influence the control latency between controller and devices; when the application has critical time requirements, limiting such latency can also be crucial.

In recent years, network topology optimisation and SDN controller placement have attracted lots of research attention. The problem of network topology customisation is often studied using optimisation approaches. Gódor et al. [2005] proposed a heuristic algorithm which combines clustering and local optimisation

[13] http://openvswitch.org/

operators to optimise the cost of hierarchical network planning. The costs of the network are aggregated together from level to level with the degree constraints. Rosenberg [2005] tried to design a network topology with the minimum number of links under the constraints of network diameter, degree and survivability, conducting theoretical analysis on the problem and proposing a method with a mathematical model. However, detailed information about the algorithm is not given. For the DRIP planner, we consider similar QoS requirements as Rosenberg and propose a meta-heuristic approach. Kamiyama et al. [2009] targets at designing a network topology which can guarantee the connectivity and total link length. As the search space enumerating all the possibilities of links is too large, they applied binary partitioning and introduced extra constraints to reduce the search space. In [Tuba, 2010] the maximum entropy method (MEM) is applied to solve the problem of network design with the objective to minimise the cost under the constraint of link capacity and latency. Fencl et al. [2011], to solve the problem of network topology design with the objective of fault tolerance and capacity of traffic and delays, apply a genetic algorithm. In summary, only Rosenberg considered the QoS of network reliability and network diameter constraints and presented theoretical analysis when customising network topology. However, a detailed description of the solution is not given. The controller placement problem was first addressed in [Heller et al., 2012]. The placement metrics average-case latency and worst case latency are still widely used in current studies. Pareto-based Optimal COntroller placement(POCO) [Lange et al., 2015; Hock et al., 2014] is a framework for Pareto optimal controller placement in terms of different performance metrics. The controller-to-switch and controller-to-controller latency are considered to measure the network resilience. [Cheng et al., 2015] proposed three heuristic algorithms to solve the problem of QoS-guaranteed controller placement. The algorithms are more concerned with how to partition the network from the controller viewpoint. However, these existing works assume the number of controllers is assumed to be given. In the SDN network planning problem, this number is not known before. So we propose a solution that can determine the placement of controllers and the number of controllers needed.

From the existing work, we can see most of the topology customisation work study physical networks, without considering the SDN aspects; and the SDN placement studies mainly focus on the pre-defined physical networks. In cloud environments, combing these two perspectives are clearly needed, so here we formulate this problem as the *virtual SDN network planning problem*, and propose a Topology-Controller planner (TCPlanner) to solve the problem [Wang et al., 2017b], which can be seen as an extension of the MEPA planner described previously.

3.4 SDN network planning problem specification

The virtual SDN network planning problem is to customise a network topology and place the controllers that can meet the given QoS requirements. As discussed above, the VMs provided by the cloud can act as the switches. Network topology customisation determines how these virtual switches are connected. We assume that the users specify the number of virtual network devices (routers or switches) as N and QoS requirements (network diameter and reliability). The network diameter d is the communication cost of the longest path between all the pairs in the graph. It can reflect the worst end-to-end latency in the network [Rosenberg, 2005]. Therefore, we consider the network diameter specified by the user as the one of the QoS requirements. Due to the dynamics of cloud, the virtual links between VMs can fail or degrade occasionally [Hwang et al., 2016]. Therefore, the reliability in the virtual network topology customisation is another important issue which should be considered. In this paper we use single arc survivability to represent the reliability of the network. Single arc survivability means that when a single link in the network topology fails, the network is still connected. There are limited number of ports in network devices even though it is virtual instead of physical. The cloud provider may also limit the number of links that a VM is able to connect due to the limitations of physical infrastructures. More specifically, Δ is the maximum number of links from any given VM. The network topology customisation problem is to define a network topology that has single-arc survival with network diameter no greater than d and node degree no larger than Δ. The overall objective of the network topology customisation is to design a network topology with the minimum number of network links within the constraints described above.

The controller placement problem is to determine the number of controllers and places where controllers should be deployed. We assume that the controllers can manage the same number of virtual switches and the controllers can be placed in the same place as the virtual switch. The controller-to-controller and controller-to-switch communication are also enabled through the virtual network links in the network topology planning phase so that no extra links need to be re-planned. The controller-to-controller latency and controller-to-switch latency are two typical QoS requirements when placing controllers [Shah et al., 2013]. In this paper we use $\pi_C^{maxlatency}$ and $\pi_S^{maxlatency}$ to represent the maximum permitted controller-to-controller latency and controller-to-switch latency. We $\pi_S^{avglatency}$ to represent the average controller-to-switch latency. $\pi_S^{avglatency}$ is quite crucial to SDN because the controller needs to communicate frequently with the switches. Thus, in this paper we try to minimise the number of controllers and $\pi_S^{avglatency}$ within the constraints of $\pi_C^{maxlatency}$ and $\pi_S^{maxlatency}$.

An approach called Topology-Controller planner (TCPlanner) is proposed to solve the virtual SDN network planning problem. The TCPlanner first customises the network topology to meet the high level requirements, which can be given by the network developer or applications, and then places the optimal number of controllers within the planned topology. TCPlanner plans a topology to connect virtual network devices based on network diameter and reliability. We use d' to represent the maximum end-to-end latency tolerable for users. Thus, to guarantee the d' and $\pi_C^{maxlatency}$, we set $d = max\{d', \pi_C^{maxlatency}\}$.

Such a problem can be viewed as a transformation of the Minimum-Cardinality-Bounded-Diameter (MCBD) Edge Addition Problem which has been proved to be NP-hard [Li et al., 1992; Abd-El-Barr, 2009]. Theoretically, there exists a brute-force algorithm that solves the problem by iterating through all feasible solutions. In Cloud, the virtual links can be planned between any pair in the virtual network. Thus, there exist $(N \times (N - 1))/2$ links, so the scale of searching space will be $2^{N \times (N-1)}$. This is possible for small-scale graphs, but becomes computationally prohibitive when N is very large. A meta-heuristic approach based on evolutionary algorithms is adopted in TCPlanner, because evolutionary algorithms have been demonstrated as a feasible solution for several similar problems [Tsai and Rodriguez, 2014].

We model the network connectivity using a communication matrix and assume the links between nodes are not directed; the communication matrix is thus symmetric. We encode a solution to a chromosome with length of $(N \times (N - 1))/2$. Each element in the chromosome is 1 or 0, which indicates whether a link exists or not between vertices. Correspondingly we also design a decoding algorithm to decode the chromosome as a graph. The initial population can be seen as the 'seed' of the initial state which can have great effect on the performance of the genetic algorithm. Usually the initial population can be heuristically crafted or randomly generated. It is difficult to follow certain heuristics to create the initial population, so all the individuals in it are randomly generated. The assignment of each position has equal probability.

Due to the complex various situations in which the constraints described above can be violated, we add a penalty factor for each violation and aggregate them with the number of links into the fitness function. As the objective is to minimise the number of links, we use the reciprocal of the sum of the link number and penalties as the fitness function which is calculated as:

$$1/\left(LinkNum + \sum_{i=1}^{k} x_i \times penalty_i\right)$$

$$x_i = \begin{cases} 0 & The\ i^{th}\ constraint\ is\ not\ violated \\ 1 & The\ i^{th}\ constraint\ is\ violated \end{cases}$$

$LinkNum$ represents the number of links in the planned network topology. $penalty_i$ represents the extent of the violation of the i^{th} constraint. The penalties above are to avoid unfeasible solutions like unconnected graphs or graphs that violate the specified constraints. In our scenario, there are four possible situations where the constraints can be violated: diameter violation, connectivity violation, degree violation and survivability violation.

We use genetic operators crossover and mutation to produce new generations of individuals and introduce diversity. We set the probability of crossover between two parents as a static value p_1 for each generation so that in each iteration new chromosomes will be produced by intersecting the parent's chromosomes with a certain probability p_2. After the off-springs are generated, $p_3 \times PopSize$ individuals are mutated by switching certain places in their chromosomes from 1 to 0 or 0 to 1. $PopSize$ refers to the population size. p_3 represents the probability of mutation of individuals. Then the next generation is selected with the individuals of the best fitness value and the population remains the same size as last generation. After planning the topology, TCPlanner will determine the number and placement location of the SDN controllers. The objective is to minimise the number of controllers and average controller-to-switch latency under the constraints of the capacity of controllers and maximum latency of controller-to-switch [Shah et al., 2013].

In TCPlanner, we sort the degree of the nodes in the planned network topology in descending order and first choose the vertex v with the maximum degree as the centre of the first cluster. The higher degree a vertex has, the more chances the average latency can be reduced when looking at all its neighbours one step further away. At each level neighbours of the centre node, we first choose the node with the minimum degree so that it minimises the interference on other clusters. Vertex v tries to 'absorb' its neighbours in this way until the controller capacity or the maximum controller-to-switch latency is violated. Such process will continue until all the nodes are assigned to a cluster. In each cluster, its centre node is the place where controllers should be placed. The number of controllers is equal to the number of clusters. The results of this process can then be used by a provisioning service to actually place SDN controllers and configure the surrounding network topology.

3.5 Evaluation

To test the effectiveness of TCPlanner, we compared it with a K-Medoids based solution as the baseline [Park and Jun, 2009]. The K-Medoids algorithm is intended to classify a data set into several clusters based on the node distance. The basic process of K-Medoids is to randomly initialise K centres of clusters and add nodes to the clusters based on the distance between the centre node and non-clustered node. Then the algorithm will try to calculate some centres to reduce the inter-cluster and intra-cluster distance. The algorithm will converge when no better centres can be found. As the K-Medoids algorithm needs to specify the number of clusters before the execution of the algorithm, we use the square root of the number of nodes as the initial number of clusters. When a cluster in the solution given by K-Medoids algorithm exceeds the capacity of the controller or the maximum controller-to-switch latency is violated, we increase the number of clusters by 1. We conduct simulated experiments on different scales of networks to test the effectiveness of the proposed solution. Our solution is implemented in Python and depends on NetworkX and DEAP, just as for our earlier experiments in Section 3.2. We set the number of network devices N ranging from 6 to 25. The diameter of the network is set as $\lceil \sqrt{N} \rceil$. The maximum degree of the network devices is $d + 1$. d represents the diameter of the network. We set the maximum generation number of the genetic algorithm to be 250 to ensure that a feasible solution can be found.

From Figure 3-9 we can see that the number of links needed to guarantee the QoS increases with the number of network devices roughly linearly. In order to evaluate the performance of TCPlanner, we design a greedy algorithm which utilises all the degrees of each port. Therefore, the number of links that can meet the QoS requirements reaches $N \times \Delta$. From the result we can see that TCPlanner outperforms the greedy solution.

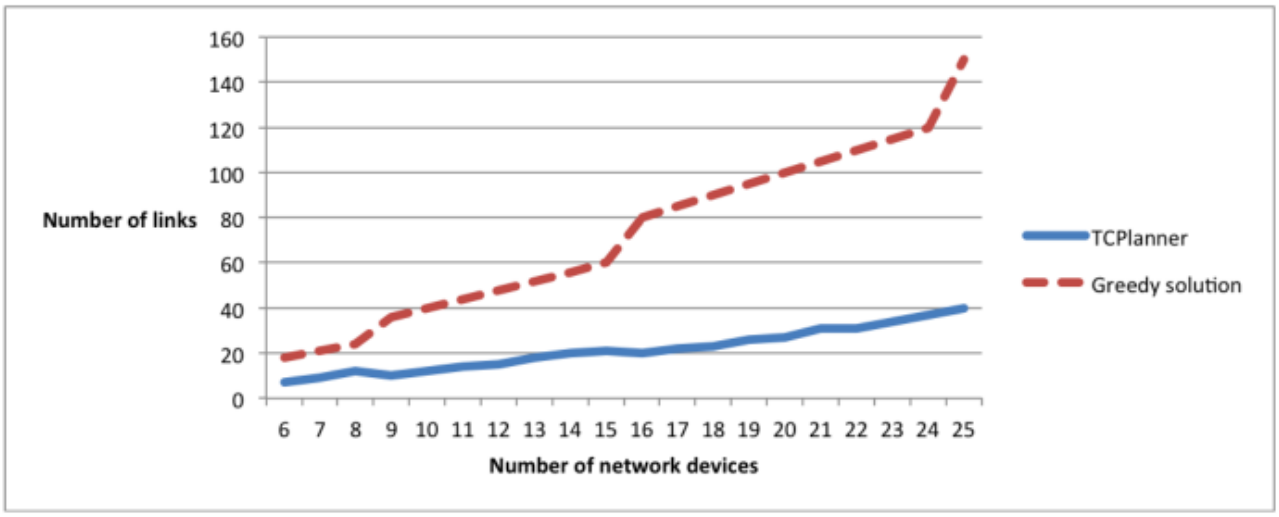

Figure 3-9 Number of links for a network topology with certain QoS requirements.

We take the network topology generated from the data plane planning phase and compare the results of K-Medoids and TCPlanner from the number of controllers and average controller-to-switch latency. The results are shown in Figures 3-10 and 3-11.

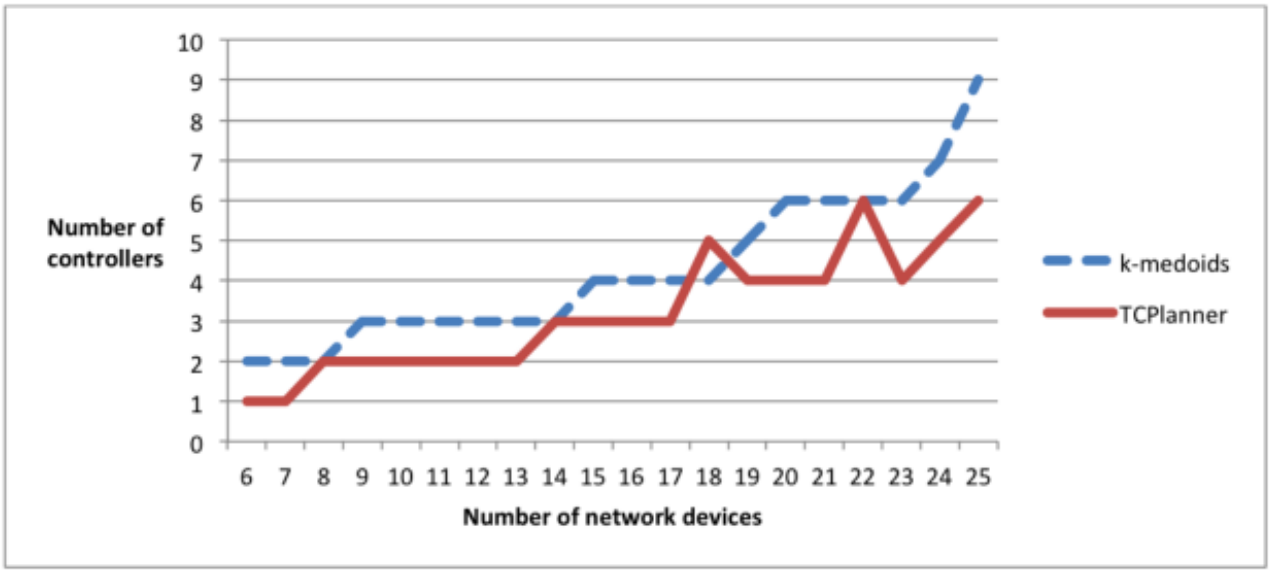

Figure 3-10 Number of controllers deployed by K-Medoids and TCPlanner.

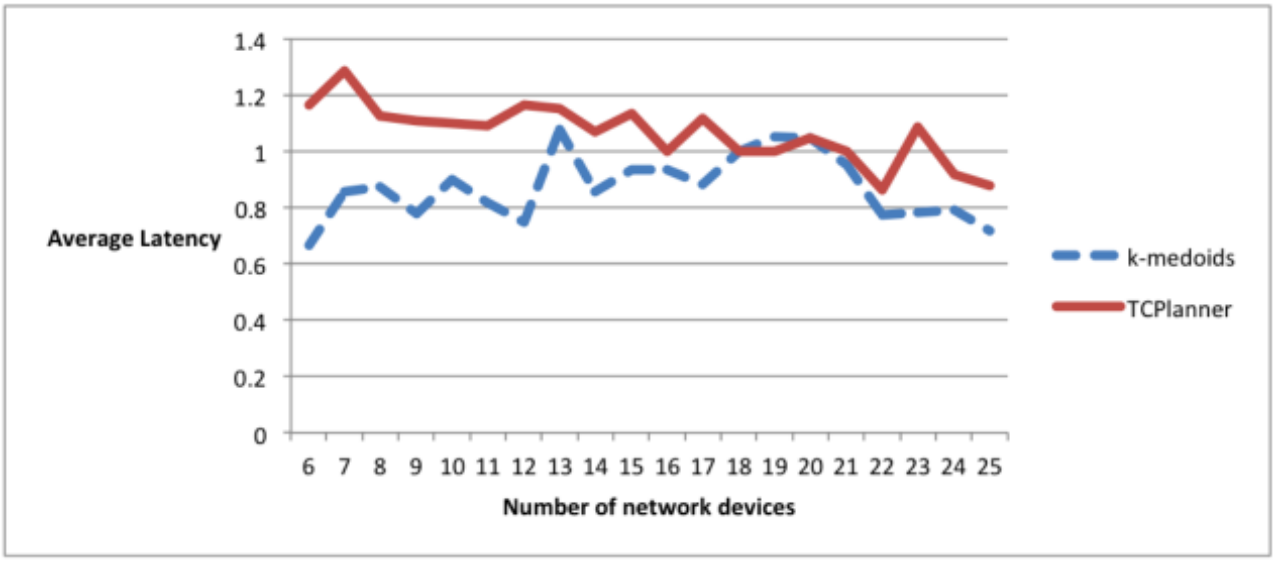

Figure 3-11 Average controller-to-switch latency for K-Medoids and TCPlanner.

From the results we can see that the K-Medoids based solution needs more controllers than TCPlanner but can reduce the average latency. The result is reasonable because the K-Medoids based solution tries to cluster the graph to minimise the intra-class distance and inter-class distance. With the increase of the number of network devices, the number of needed controllers can be reduced dramatically. When the scale of the topology reaches 25, TCPlanner can deploy three fewer controllers than the K-Medoids-based solution.

We only consider the latency constraints in the current prototype; other QoS attributes such as bandwidth can also have great impact on the performance of the network. Future work should thus include more network QoS constraints in the planning process. Moreover, the characteristics of application traffic patterns and the dynamic QoS control of SDN network can also be investigated in the planning algorithm, which indicates further development possibilities for the DRIP planner beyond the final SWITCH public release.

4 Dynamic cloud performance information

Over the last decade, the usage of cloud computing has become increasingly popular. With the increasing amount of available instances and cloud providers it is becoming increasingly difficult for application developers to select the right cloud provider for their application. Most cloud providers provide static information (e.g. CPU cores, memory size, disk size, and disk type) of different kinds of virtual machines (VMs). However, when an application developer wants to deploy a mission-critical application in the cloud, the static information provided by the cloud provider is often insufficient, because static information does not take into account the hardware and software that is being used or the policy that has been applied by the cloud provider. Therefore, more precise information about cloud resource types and provisioning constraints is crucial to successfully deploy an application within the cloud [Zhao et al., 2015; Zhao et al., 2016]. Over the last few years many automated benchmark tools are proposed in literature, all of which aim to help a single user to benchmark multiple instances and/or providers, so that the user is able to select the right instance according to their requirements. However, the performance of those instances may be different each time it is measured [Iosup et al., 2011; Leitner and Cito, 2016]. Thus, it would be helpful if users can obtain statistical information about the consistency and stability of cloud resources. For the DRIP performance modeller component, we need to look at how to test the performance consistency and stability of provisioned cloud resources. The systematic collection and sharing of such information will allow the DRIP planner to select the most suitable resources for mission-critical applications.

The research and development results of this section has been published in the IEEE Networking, Architecture and Storage (NAS) [Elzinga et al., 2017].

4.1 State of the art

Based on the challenges presented, we identify custom benchmarking as an important functional requirement in SWITCH. Implementing new and custom benchmarks is an important feature for a cloud performance evaluation tool. By using a well-defined way of specifying which application needs to be installed, configured and executed, it should be easy to implement any type of application to benchmark it on different cloud resources/providers. The requirements for selecting an automatic benchmark tool for DRIP are:

1. **Public availability**: most importantly, the application must be publicly available for usage.
2. **Open-source**: in order to make the community contribute to the tool as well, the application must be open-source. When applications are open-source and downloadable via for example GitHub, it helps to get the community involved with the project.
3. **Maintainability**: an important aspect of the application is that it must be maintained frequently. The cloud evolves rapidly and new features are presented regularly and therefore it is important that the automated benchmark tools follow the new trends of the Cloud and keep helping customers to select the right cloud resources.
4. **Support for IaaS providers**: in many cases users will compare multiple providers to select the best offer according to their wishes. Therefore, the application must support a large amount of public and private IaaS providers.

Over the last few years there are several automated cloud benchmarking tools proposed in literature. Chhetri et al. [2013] proposed Smart CloudBench, which is a platform that automates the performance benchmarking of cloud infrastructure, helping potential consumers quickly identify the cloud providers that can deliver the most appropriate price/performance levels to meet their specific requirements. They looked at benchmarking from the consumer's perspective and focused on benchmarking the entire application stack instead of looking at individual components. Cunha et al. [2017] proposed an automatic benchmark tool called the Cloud Crawler. The tool helps users to describe and automatically execute application performance test inside the cloud. New benchmarks are defined in a declarative domain-specific language called Crawl, which is based on YAML. Scheuner et al. [2014] presented Cloud WorkBench (CWB). CWB is designed and implemented to leverage the notion of Infrastructure as Code (IaC) for cloud benchmarking, and is used to

automate the benchmarking lifecycle from the definition to the execution of benchmarks. CWB uses Vagrant[14] to provision virtual resources and Opscode Chef[15] to install and configure the benchmark tools. CWB can run benchmarks directly or schedule benchmarks within various public Infrastructure as a Service (IaaS) clouds. Silva et al. [2013] presented CloudBench. CloudBench is an open-source framework that automates IaaS clouds to run controlled experiments, where complex applications are automatically deployed. The authors demonstrated CloudBench main characteristic through the evaluation of an OpenStack installation, including experiments with approximately 1200 simultaneous VMs at an arrival rate of up to 400 VMs/hour.

Table 4-1 compares the most important requirements of the tools proposed in literature described in the last section. None of the tools proposed in literature met our requirements, therefore, we decided to create our own tool for use in DRIP. We identify a number of technical gaps that we try to bridge in this research:

1. **Ability to add providers**: none of the tools have the ability to easily add providers to the tool. For example, the Cloud WorkBench uses Vagrant, which works well for the platforms Vagrant supports. However, when a provider is not supported by Vagrant one has to find an other way to add that provider. It would be helpful if a new provider could be implemented regardless of the type of software used to do so. Therefore, a well defined framework will be of great use to define a standard way of writing such a piece of code that controls the orchestration VMs.
2. **Possibility to add custom benchmarks**: most of the tools proposed in literature do not provide a way to add custom benchmarks in an easy way. Therefore, it's important that the installation, configuration and the benchmarking process are defined in a powerful way and in a common language (e.g. JSON, YAML) so that it's easy for users to benchmark their scenario.

Table 4-1 Comparison of proposed automated benchmark tools and our requirements.

	Publicly available	Open source	Custom benchmarks	Schedule	Provider support
Cloud WorkBench	Yes	Yes	Yes	Yes	Only public
Cloud Crawler	Yes	Yes	Yes	No	Only public
CloudBench	Yes	Yes	No	No	Public and private
Smart CloudBench	No	-	No	Yes	Only public

4.2 Cloud performance collector

The basic steps involved in the establishment of a testing framework for cloud resources are shown by the sequence diagram in Figure 4-1. In the first step, the experimenter (user) runs or schedules one or more benchmark scenarios. When a scenario is executed, the CPC will first provision the necessary resource via the cloud Application Programming Interface (API). As soon as the VM instance is reachable, the software can be installed and configured depending on the layout of the scenario. After the successful installation/configuration of the software, the benchmarks can be executed. When a benchmark is finished, the results will be collected by the CPC. After all benchmarks are finished and all the results are collected, the CPC will release the VM to keep the time the VM is used to a minimum. To make it easy for developers to implement new features, the design includes three modules: the provider module, the 'deploy and run' module, and the result module. The *provider module* makes it possible to provision and release VMs when the experiments are finished. The *'deploy and run' module* takes care of installing, configuring, and executing the benchmarks. The *results module* parses all the useful information out of the output of each benchmark application.

[14] http//www.vagrantup.com/
[15] http://www.getchef.com/

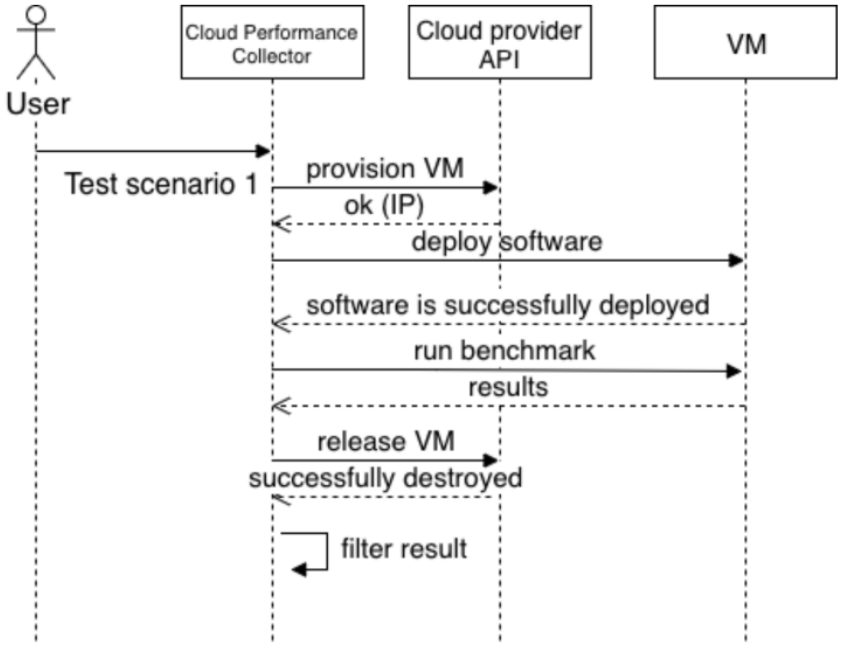

Figure 4-1 The CPC benchmark execution process.

To demonstrate the benefit of the design, we build a prototype to do experiments with. The prototype is a command-line interface (CLI) tool written in bash. During the experiments ExoGENI will be used as provider, therefore the *provider module* will make use of the python script omni[16] to communicate with the API of ExoGENI. The *deploy and run module* which takes care of installing, configuring and executing the benchmarks will be done via Ansible[17]. The scheduling of benchmarks is done via Linux cronjobs. The *results module* consists out of small bash scripts to filter the output.

4.3 Performance data collection experiments

In order to illustrate the benefits of the CPC we conduct several experiments using the ExoGENI test-bed. ExoGENI [18] is a distributed networked infrastructure-as-a-service (NIaaS) platform geared towards experimentation and computational tasks. During these experiments we aim to answer these questions:

1. **Performance consistency**: will VM instances with the same specifications perform consistently each time they are provisioned?
2. **Performance stability**: will the same VM instance with the same workload maintain the same performance levels over time?

The goal of the first question is, to measure if there is a difference between different provisioned VMs, using the same specifications and image from the same provider. The goal of the second question is, to find out whether a VM instance once provisioned performs the same over a longer period of time. By comparing the results of the first question with the results of the second question, we can analyse whether the performance variation is depending on the time of the day or the physical location of the VM instance or both. Moreover, we measure the performance of a real-world application, so we can demonstrate how the CPC can test any given application component. During each experiment, we will run the tool 24 times, scheduled each hour.

[16] http://trac.gpolab.bbn.com/gcf/wiki/Omni
[17] https://www.ansible.com/
[18] http://www.exogeni.net/

4.3.1 Experimental setup

All experiments were conducted on the ExoGENI test-bed using the racks of: The National ICT Australia (NICTA), Raytheon BBN Technologies (BBN), and the University of Amsterdam (UvA). The experiments were conducted on the "current types" offered by ExoGENI[19]. During the experiments, we used three different instance types: XOMedium, XOLarge, and XOXLarge. Table 4-2 shows the specification of the current resource types offered by ExoGENI. All instances are using the same Ubuntu 14.04 image.

Table 4-2 Resource types offered by ExoGENI.

Resource Type	Resource Name	Cores	RAM	Disk(s)
VM	XOMedium	1	3G	25G
VM	XOLarge	2	6G	50G
VM	XOXLarge	4	12G	75G

During the experiments four applications were used, three benchmark tools, and a real-world application. *sysbench*[20] is a modular, cross-platform and multi-threaded benchmark tool to quickly get an impression about system performance. During our experiments, we use sysbench to benchmark the CPU by verifying prime numbers of 100,000 natural numbers. sysbench measures the time it takes to calculate those number in seconds. The *STREAM*[21] benchmark is used to measure the performance of the main memory. STREAM is a benchmark which is designed to measure sustainable memory bandwidth using four vector-based operations: COPY $(a = b)$, SCALE $(a = q \times b)$, SUM $(a = b + c)$, and TRIAD $(a = b + q \times c)$. During our experiments we chose to use the TRIAD operation since it is the most complex operation with STREAM measuring the throughput in MB/s. *IOzone*[22] is a benchmark used to measure the read and write performance of the disk. To reduce the time it takes to complete both the read and write process, we make use of a file size of 2GB with a record size of 64Kb. IOzone measures the throughput in MB/s. To demonstrate that any type of application could be tested, we will use the application Montage[23] inside a Docker container. Montage is a toolkit for assembling Flexible Image Transport System (FITS) images into custom mosaics used in a variety of research contexts, notable for its flexibility and parallelisability in Grid and Cloud contexts. We will measure the time it takes for the application to create the image in seconds.

4.3.2 Experiment 1: Performance Consistency

In this experiment, we investigated if VM instances with the same specifications from the same provider performs similarly. During this experiment we used a different VM instance every two measurements. By benchmarking the same instance twice before a new one is used, we can see if the variation is caused by the fact that the instance is placed on a different physical server or by a lack of performance isolation on a single physical server (noisy neighbours). Figure 4-2 shows the results of running the sysbench CPU benchmark.

[19] https://wiki.exogeni.net/doku.php?id=public:experimenters:resource_types:start
[20] https://github.com/akopytov/sysbench
[21] http://www.cs.virginia.edu/stream/stream2
[22] http://www.iozone.org/
[23] http://montage.ipac.caltech.edu/

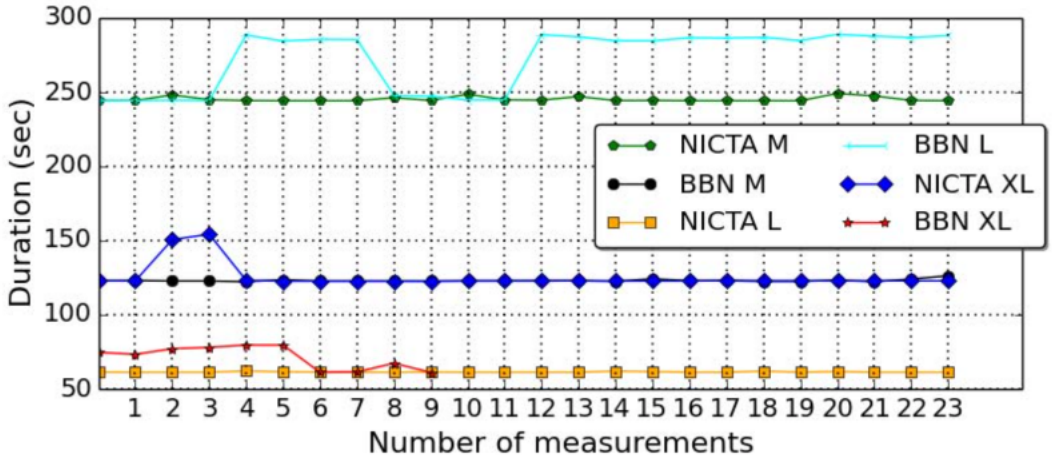

Figure 4-2 Variation in performance of different VM instances running sysbench.

The instances running on the rack of NICTA perform quite stably and have little to no performance variation when a different instances is used. In contrast to the instances running on the rack of NICTA, we observed large performance variations when a new instance is provisioned on the rack of BBN. However, when we run the same benchmark for the second time on the same VM instance, we see a similar level of performance. It is likely that the instance is placed on a different physical server within the rack. After ten measurements it was not possible to provision the BBN XL instances again. Therefore, during this experiment the data available of the BBN XL instance is limited. Similar problems occurred on provisioning instances on the rack of the UvA. Therefore, we decided to not include the instance of the UvA during this experiment. Figure 4-3 shows the memory throughput measured by STREAM.

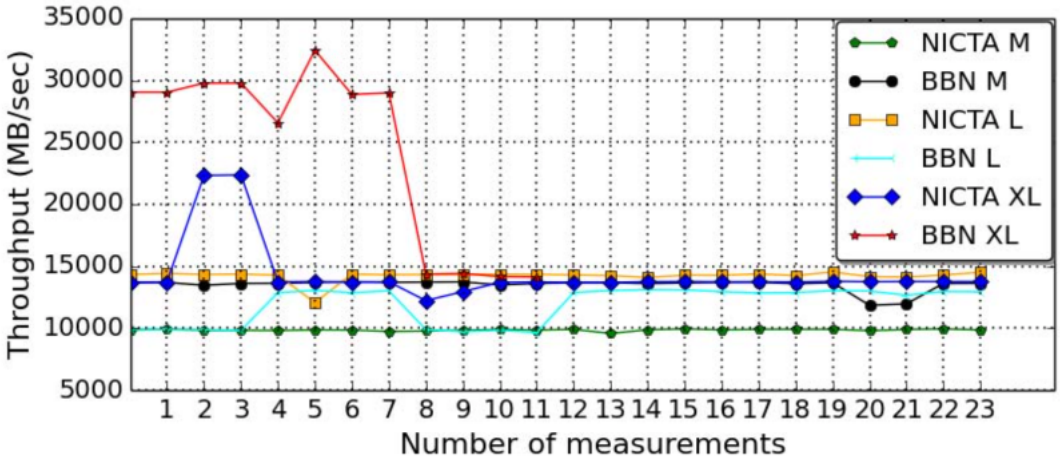

Figure 4-3 Variation in performance of different VM instances running STREAM.

In general, the results of the RAM benchmark shows the same behaviour compared to the results of the CPU benchmark. The Large and XL instance of NICTA show slightly decrease in performance in some measurements. The instance on BBN shows the similar pattern compared to the CPU benchmark. Interesting is that during a measurement during which the memory throughput is higher, the time it takes for sysbench to finish is longer. For example, the first four measurements on both the NICTA Medium and the BBN Medium instance show more or less the same result. During the fifth measurement of the BBN Medium instance, we see an increase in execution time during the CPU benchmark but also an increase in memory throughput. The performance disk I/O has the tendency to vary more compared to CPU and memory, during this experiment we can see this behaviour on the instances of NICTA as well. However, we did not see this

behaviour on the instances of BBN were the CPU and memory in some cases vary more compared to the disk I/O. Figure 4-4 shows the read performance of the different instances, all instances perform similar, whereas the instances of BBN perform slightly higher.

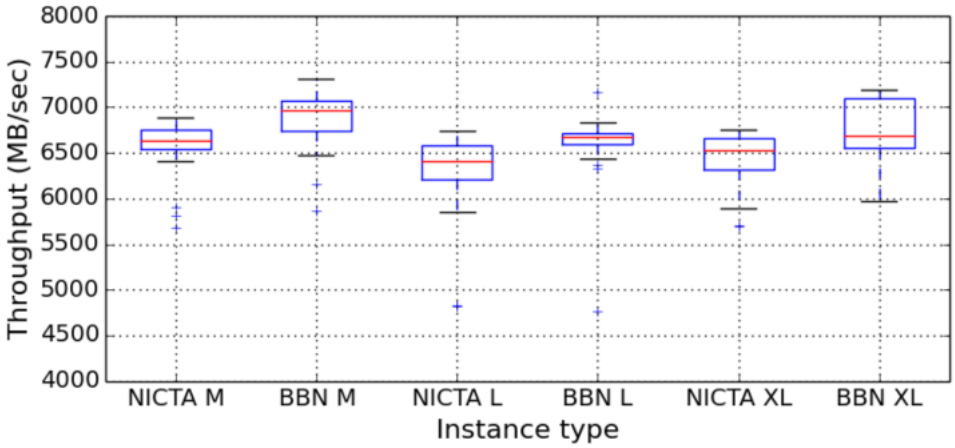

Figure 4-4 Variation in performance of different VM instances running IOzone (to read).

The write performance is shown in Figure 4-5.

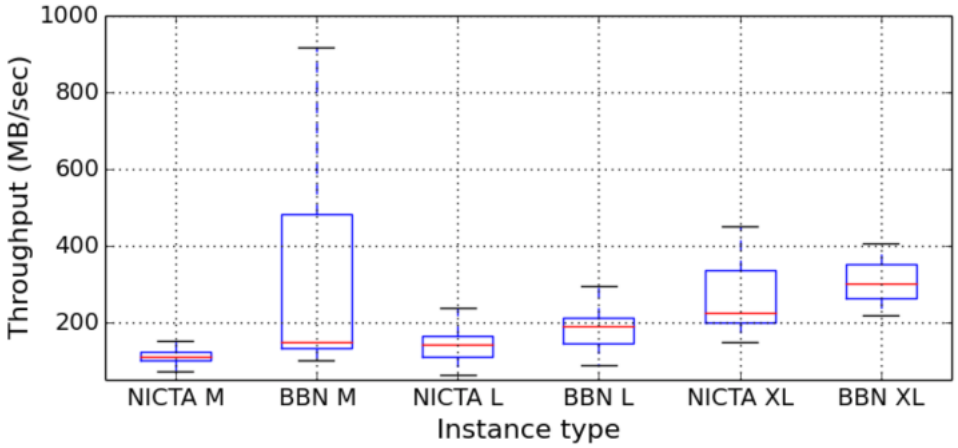

Figure 4-5: Variation in performance of different VM instances running IOzone (to write).

Compared to the read performance, the larger instance tends to perform a little bit better compared to the smaller instances. Interesting is that the BBN Medium instance shows a big variation in performance.

4.3.3 Experiment 2: Performance Stability

During the second experiment, we investigated if the same VM instance with the same workload provide a stable level of performance over time. For this experiment we provisioned a VM instance for each provider/resource type and we will use that same VM instance for all measurements. Figure 4-6 shows the results of running the sysbench CPU benchmark.

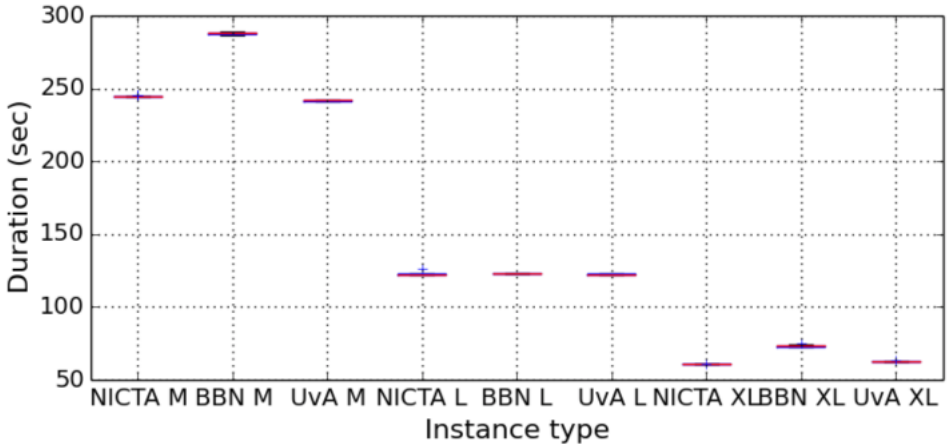

Figure 4-6 Variation in performance on the same VM instance running sysbench.

All the measured instances show almost no variance in performance. The Large instance of all three racks perform on the same level. However, the Medium and XL instance of BBN performed less compared to the same instance of the other racks. Figure 4-7 the variety in performance of STREAM.

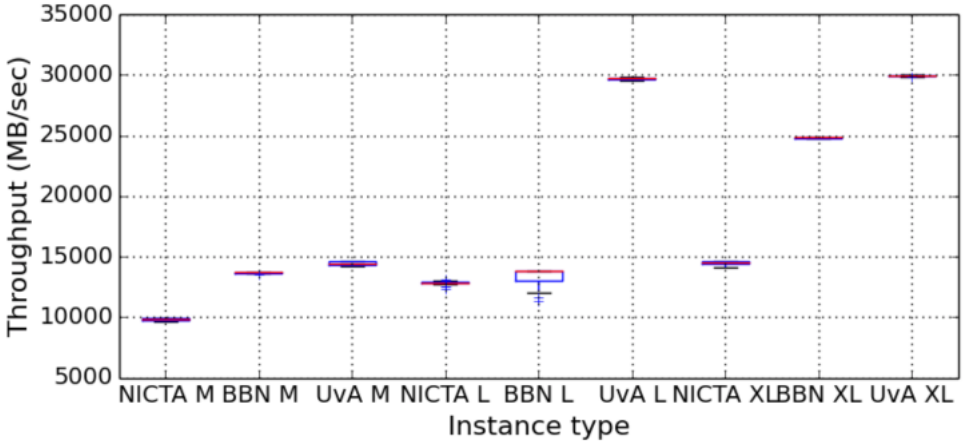

Figure 4-7 Variation in performance on the same VM instance running STREAM.

Whereas, the results CPU shows almost no variation in performance, the results of the memory show some small differences. The BBN Large instance shows significant difference and the NICTA Large and NICTA XL show some difference as well. The results of the disk I/O are similar to the first experiment. Still there is a lot of performance variation measured on all instances. However, the performance variation on some instances seem to be less compared to the first experiment. Figure 4-8 shows the read performance of the different instances.

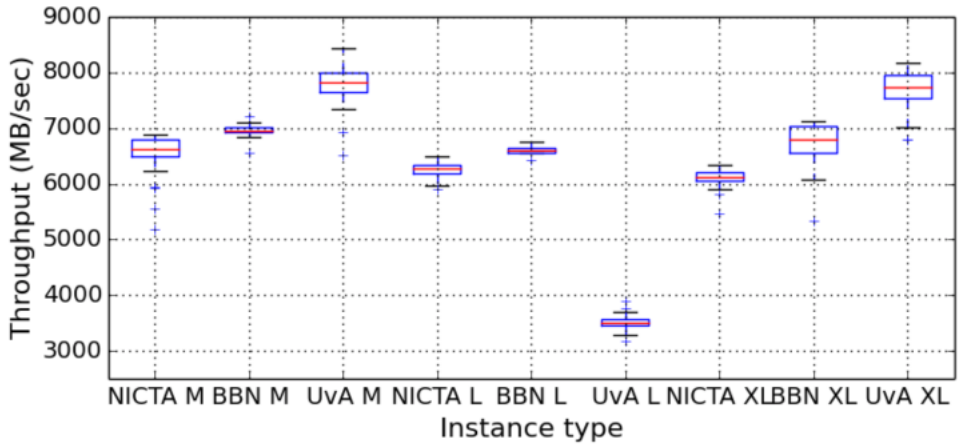

Figure 4-8 Variation in performance on the same VM instance running IOzone (to read).

In almost all cases (except for UvA Large) the instance running on the UvA have the highest performance followed by BBN. A possible explanation for the results of the UvA Large instance is the fact that the UvA rack is heavily used (which resulted in provisioning problems during the first experiment). Figure 4-9 shows the write performance of the different instances.

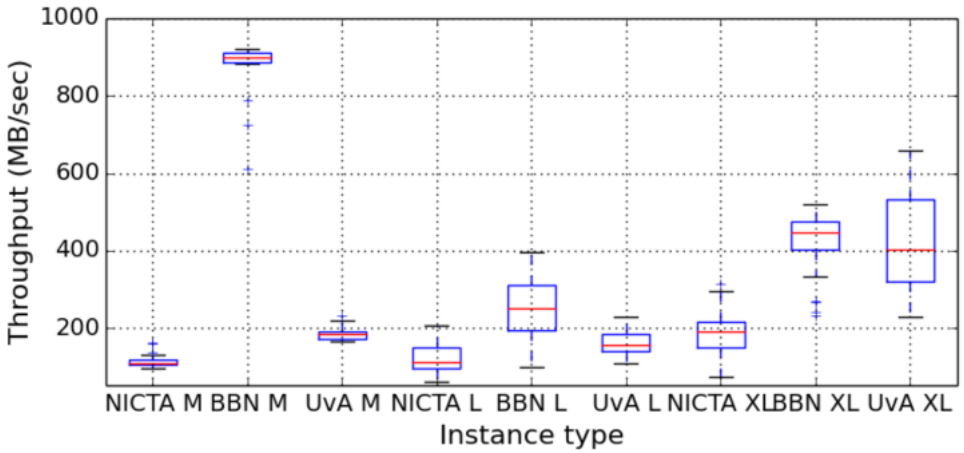

Figure 4-9 Variation in performance on the same VM instance running IOzone (to write).

In general, the larger instance tends to perform a little bit better compared to the smaller ones. Just like the first experiment, the write throughput of the BBN Medium is much higher compared to the write throughput of the other instances. However, this time the variation is much lower than during the first experiment.

4.3.4 Experiment 3: Performance variation of a real-world application

Figure 4-10 shows results for the Montage application running inside a Docker container.

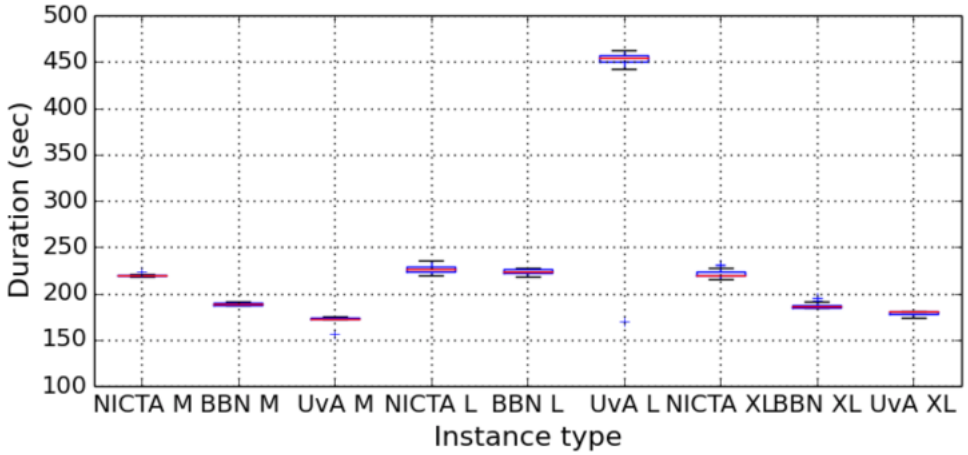

Figure 4-10 Results of running Montage inside a Docker container.

During this experiment, we used the same VM instance that is used during the second experiments. Similar to the results of the other experiments, during this experiment we can see that all instances show little performance variation over time. Interesting is that all medium instances are performing similar or better compared to their same provider counterpart.

4.4 Discussion

During our experiments, we have looked at performance consistency and have seen that when a new VM instance is provisioned the performance can be similar but in some cases can differ. When we looked at performance stability, we have seen that with regards to CPU and memory the performance provided by all tested providers is constant over time. With regards to the variation of disk performance we have seen that there is much more variation during both experiments. We have seen that the read performance of the UvA Large instance was significantly lower compared to all instances, a possible explanation being the fact that the UvA rack is heavily used. However, to really understand why the performance was significantly lower more testing is needed to identify if it has something to do with the VM type or the physical server the VM is hosted on. During our experiments, we also looked at the performance variation of a real-world application. The Montage application read a large amount of images from disk and created one big image out of those images. Therefore, read performance is an important aspect of the application. When we compare the results of this experiment with the result of disk read performance of the second experiments, we can see that Figures 4-8 and 4-10 have a lot of similarities. Both figures show that the UvA is the faster except for the UvA Large instance. Both figures show that the Large instance of each rack is performing less compared the Medium and XL instance of that rack. Hence, we can conclude that it should be possible to use synthetic benchmarks to show which instance is best for a specific application. However, it is important to understand the most important components of the application to compare its results to various results of synthetic benchmarks.

These results demonstrate the feasibility of collecting some basic performance information for use in systems such as DRIP, though it should be noted that the kind of characteristics that can be accurately gathered are limited by the type of application component for which the best VM is to be selected (dependence on processing, on disk or network I/O, general variability of performance, etc.). Nevertheless, such a performance modelling tool can be used to gather data for the DRIP knowledge base, which can then be accessed by the planner, provisioner and in principle any other component that may benefit from the information therein.

5 Inter-locale virtual cloud provisioning

In DRIP we have designed and implement a flexible inter-locale Cloud engine for quality critical applications to help satisfy time-critical requirements for highly distributed big data processing. This cloud engine is able to provision a networked infrastructure, recover from sudden failures quickly, and scale across data centres or Clouds automatically. The key technologies used include transparent network connection and standardised multi-level infrastructure description. This work was originally presented in D3.2, but additional details about how the TOSCA-based plans generated by DRIP are handled are provided here, as well as a report on further experiments conducted.

The research and development results of this section have been published in IEEE International Symposium on Real time distributed computing (ISORC) [Zhou et al., 2016a], IEEE Cloud [Zhou et al., 2016b], and workshop IT4RIs in IEEE RTSS [Zhou et al., 2016c]

5.1 Challenges and gaps

According to the current state of the Cloud in industry, we infer the following challenges and gaps when migrating this kind of quality critical application onto Cloud, focusing mainly on infrastructure provisioning:

1. **Networked infrastructure.** The applications workflow becomes more complex with a lot of components that need to communicate with each other. Separated instances cannot complete the whole job. For instance, the components in different parts of a big data infrastructure need to communicate with each other and transfer data. The virtual infrastructure must therefore realise a particular network topology. Most current cloud providers cannot support this however; for example, Amazon EC2 can only allow users to describe private subnets, making it hard to build a complete topology.
2. **Nearly real-time constraints.** Nearly real-time applications require that most task deadlines be met over the lifetime of the application. Missing one deadline does not lead to immediate failure of the application, but continued failure to meet deadlines is unacceptable. We identify two particular types of nearly real-time constraints in this section. The first type is of static constraints on network transmission time as data is processed, which restrict task scheduling before provisioning. The second type is of runtime constraints restricting the time the application has to recover from sudden failures—because the application is running all the time and some failures cannot be avoided, especially where the Cloud is remote and not totally reliable. Currently developers generally put all components in one data centre. If that data centre is not accessible, then we have to re-provision the whole infrastructure within another data centre, which is a costly operation.
3. **Geography.** Not all the components of an application are on the cloud. Data collectors such as (for example) cameras providing video of a live event are not on the Cloud themselves, but provide data to be ingested into the Cloud. The geographic location of any virtual infrastructure therefore has to be considered to satisfy the nearly real-time constraints on data delivery [Alamri et al., 2013].
4. **Auto provisioning and federated cloud.** Since these applications are complex, we need a way to provision the whole infrastructure and deploy applications automatically. Currently, some tools can only provision automatically at instance level, for example Chef[24]. On the other hand, we may need more resources from other Clouds to provision a large scale infrastructure [Zhang et al., 2016]. It is a problem to combine these resources across multiple locales however.

5.2 Methodology and use

To address the challenges of SWITCH, we designed and developed a Cloud engine to set up the virtual Cloud. This virtual Cloud is an encapsulation of different data centres or other Clouds. With the help of this Cloud engine, the Cloud user can provision networked virtual infrastructure and manage all virtual resources together on the one virtual Cloud. This engine relies on transparent network connection methods and

[24] https://www.chef.io/chef/

standardised multi-level infrastructure descriptions. The engine applies two different methods to settle the problem of connectivity between partitioned topologies in different locales, which is a key step for provisioning across multiple data centres or Clouds. These two connection methods have also been discussed in detail in D3.2 and [Zhou et al., 2016].

Figure 5-1 illustrates the first connection method. It shows how one packet gets through the public network between two sub-topologies. It is mainly based on NAT. The proxy node works as a mirror of the node in another topology and is not made visible to the Cloud user. At the same time, VM1 and VM2 can communicate via private IP addresses, which are selected by the application developer.

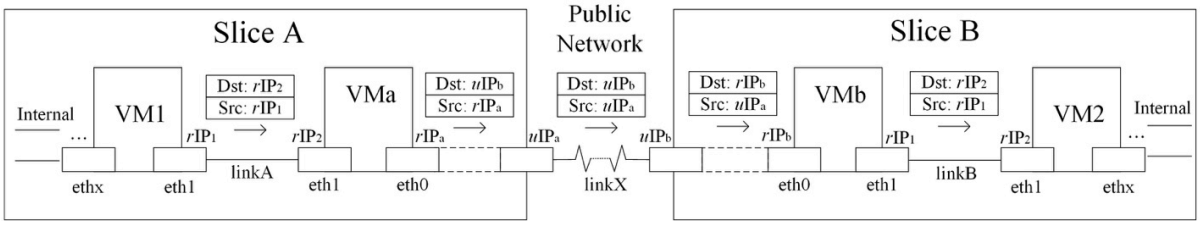

Figure 5-1 Connection technique with proxy nodes.

The second method to connect these sub-topologies is using IP tunnelling. This method is shown in Figure 5-2. With the IP tunnelling technique, the original packet, which uses the internal private network addresses provided by the application developer, can be wrapped in another packet which allows the original packet to be delivered through the public network.

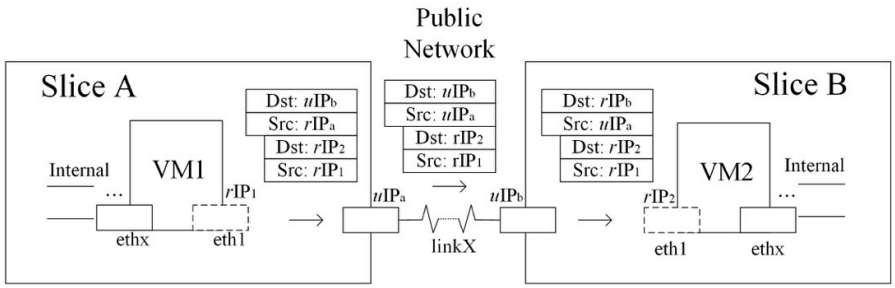

Figure 5-2 Connection method with IP tunnelling.

The advantage of the second method is that it does not add the extra overhead of proxy nodes for every link that crosses sub-topologies, in contrast with the first method. However, only some versions of Linux support IP tunnelling by default. If the customer adopts (for example) Windows for the virtual machines to run on, then the second method cannot be easily made to work. Another disadvantage of the second method is that we need to re-configure the original nodes provided by the developer. It is therefore not totally transparent when compared with the proxy node method. We therefore adopt both methods and choose which one to apply depending on the specific situation. Meanwhile, we have tested to confirm that the network performance will not significantly drop with use of either of these methods.

Another key part of our solution lies with infrastructure description. The infrastructure specification used by our engine adopts the TOSCA standard, expressed in YAML. The multi-level description is used to provision infrastructure provided by different data centres or even different Clouds. Figure 5-3 shows an example of the files used.

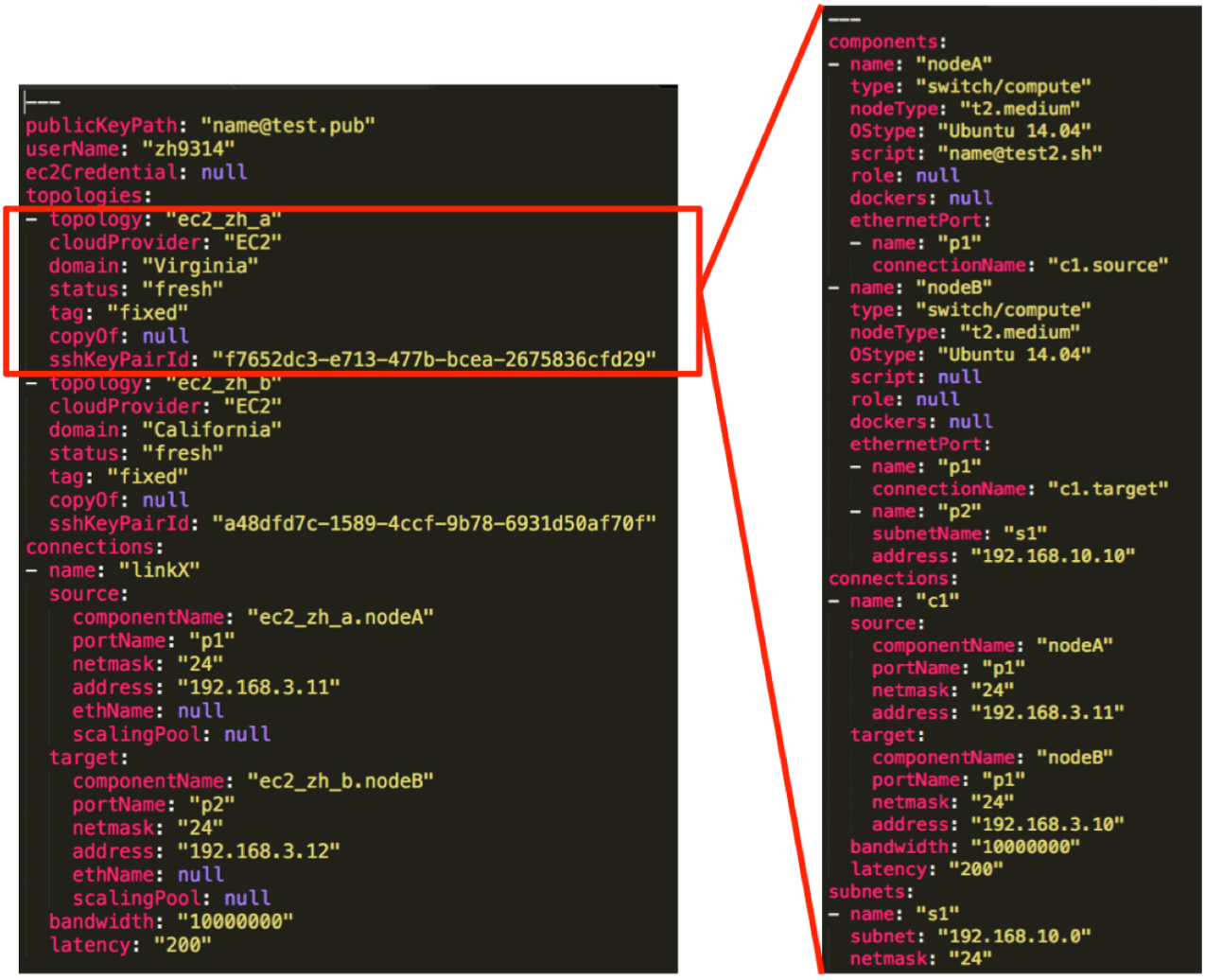

Figure 5-3 Example of an infrastructure description (zh_all.yml on the left, ec2_zh_a.yml on the right).

In Figure 5-3 the file **zh_all.yml** provides a top-level infrastructure description. It specifies different sub-topologies and their providers. The field 'topologies' defines the whole topology. The subfield 'topology' of this field defines the name of the sub-topology. It is also the name of the low-level description file, which describes the infrastructure in more detail. The user should also define which cloud provider this sub-topology belongs to. The field 'connections' describes how the sub-topologies are connected. Besides these, the fields 'publicKeyPath' and 'userName' are important to set up the virtual Cloud. The user generates a RSA key pair. He keeps the private key and publishes the public key within the field 'publicKeyPath'. After the virtual resources are provisioned, the user can then login to every instance with the corresponding private key and the user name defined in the configuration file. Otherwise, the user would need different private keys to access resources from different cloud providers. The default user-name would also be different.

File **ec2_zh_a.yml** is an example of the low-level infrastructure description. The infrastructure resources described in one file are all in one data centre. The field 'components' describes the computing resources of VM nodes. The fields 'subnets' and 'connections' describe the network resources. Among them, the field "subnets" is used to describe several nodes in one subnet. The field 'connections' defines a specific link between two nodes. This field makes it easy to describe the network topology. It is worth mentioning that the user can specify the installation file and installation script path in each node description. With these fields, the applications developed by developers can be automatically deployed after provisioning.

These files are human readable and standardised, and are generated by the DRIP planner. The provisioning engine can then be used in a number of scenarios to satisfy the static and runtime requirements of big data applications:

1. **Provisioning networked infrastructure**. While the user can describe network topologies using networked infrastructure providers such as ExoGENI, the user cannot get network topology on other providers such as EC2 or EGI FedCloud[25], as shown in Figure 5-4. EC2 and EGI FedCloud represent the current state of most cloud providers whether private or public. With our Cloud engine, the user can describe his own network topology even on these Clouds by defining the field 'connections' in infrastructure descriptions. In addition, it is transparent to the provider, which means that the cloud provider does not need to do anything to support this feature. Thus our Cloud engine is able to set up a networked virtual Cloud across even public Clouds which do not explicitly support network topology configuration.

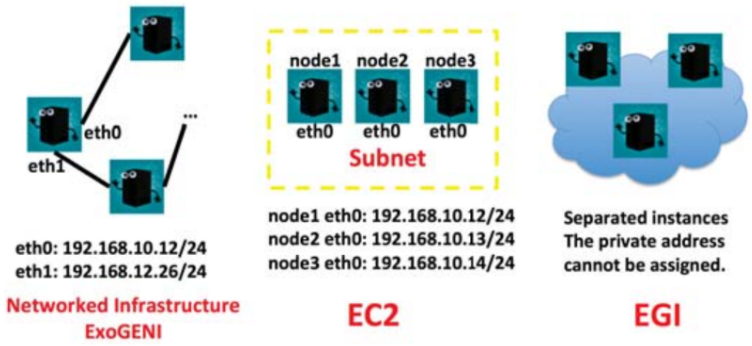

Figure 5-4 Provisioning networked infrastructure.

2. **Fast failure recovery**. Figure 5-5 describes the process of failure recovery with our Cloud engine. There are two key components of the Cloud engine that are relevant to this scenario: the provisioning agent and the monitoring agent. When some data centre is down or inaccessible, a probe previously installed on the node can detect this. The monitoring agent can then invoke the provisioning agent to perform recovery. The provisioning agent then just needs to provision the specific part of the application hosted on the failed infrastructure. As the infrastructure description is already partitioned, it is easy for the agent to provision the same topology in another data centre. Meanwhile, the connection method will keep the topology identical to the previous one. From the application point of view, the topology is the same and the application does not need to be changed. Avoiding the re-provisioning of the whole infrastructure can save a lot of time and make the overall infrastructure more reliable.

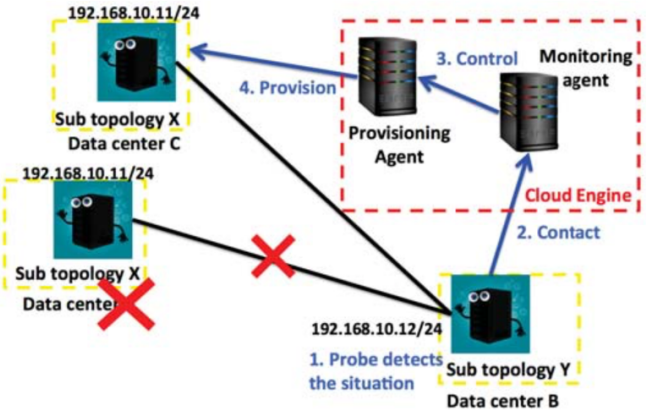

Figure 5-5 Fast failure recovery.

[25] https://www.egi.eu/services/cloud-compute/

3. **Auto scaling among data centres or Clouds**. Currently, the user can only define an auto-scaling group in one data centre as in the example of Amazon EC2. Moreover, most cloud providers do not even afford this function. With our Cloud engine, the user just needs to define an address pool for auto-scaling. Figure 5-6 shows the process. The scaling part can then be provisioned from another data centre or Cloud at runtime. More importantly, the address pool can be defined in the range of private IP addresses. The application can then be configured to know where the scaling part is before execution. Otherwise, the application needs to be configured manually at runtime. This is also useful for large-scale applications; when the resources are exhausted or limited in one data centre or Cloud currently in use, the Cloud engine can make the infrastructure scale-out to use resources from other locations.

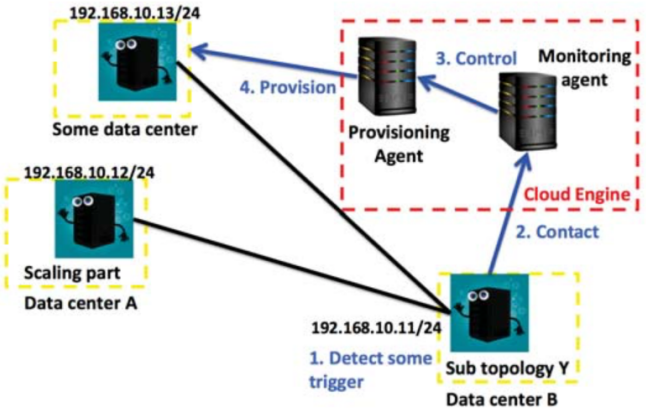

Figure 5-6 Auto-scaling among data centres or Clouds.

5.3 Evaluating new developments

We set up experiments to test the feasibility of the solution provided by the DRIP provisioner, supplementing the network experiments of [Zhou et al., 2016]. Specifically, the feasibility of transferring data between different provisioned sites (i.e. data centres) and data sources (e.g. sensors deployed in the field) under different conditions. In order to simulate a realistic scenario, we create four objects in the experiment. The detailed properties of these objects are listed in Table 5-1.

Table 5-1 Properties of objects in the experiment.

Number	Subject	Computing Properties			Access Network Properties			Geography Properties	
		CPU Core	OS	Memory	Mode	Upload	Download	Cloud Provider	Location
①	Laptop	1 GB	MacOS	8	WIFI[1]	0.94 Mbps	8.59 Mbps	-[2]	Amsterdam
②	Laptop	1 GB	MacOS	8	WIFI[3]	193 Mbps	305 Mbps	-	UvA
③	VM	1 GB	Ubuntu 14.04	3	Ethernet	-	-	ExoGENI	UvA
④	VM	1 GB	Ubuntu 14.04	3	Ethernet	-	-	ExoGENI	CA, US

[1] It is connected with the home network.
[2] '-' means unknown.
[3] It is connected with Eduroam.

We use a laptop to act in the role of data collector and put it in different network environments. For object 1, the laptop is connected to its home network via WIFI. This object is designed to simulate the situation where the data collector is far from the regional data centre and does not have a particularly good network connection. Object 2 is deployed within the campus network of UvA (University of Amsterdam) to simulate the situation where the data collector is close to the regional data centre and does have a very good network connection. Objects 3 and 4 are two VM nodes provisioned by our Cloud engine within different locales provided by the ExoGENI infrastructure platform. They are connected via private IP addresses far from each other geographically. We adopt the second connection method described in Section 2 (IP tunnelling). There are two main scenarios we need to compare. The first is the deployment of all the components in one data centre without use of our engine. The second is the adoption of our solution, which is to distribute the components on the virtual Cloud set up by our engine.

We design the first experiment to test the latency in these two scenarios. The results are shown in Figure 5-7. We start sixty ping requests one by one between different objects of Table 5-1. From the legend in the figure, we can tell which link between two objects each plot belongs to. In addition, "S1" preceding the legend indicates that it refers to the first scenario (without engine) described above and "S2" for the second scenario (with engine). It is clear that the latency is lower when the data collector is closer to the server. In the first scenario, despite the fact that the data collector has good network connectivity, the average latencies are nearly ten times higher than those in the second scenario. Moreover, the latencies in scenario 1 are not stable, especially when network access is bad, which is common for real data collectors.

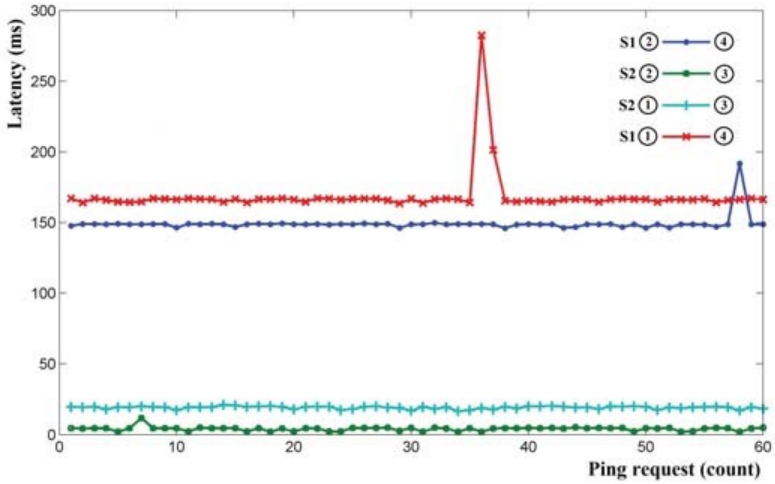

Figure 5-7 Latency comparison.

The second experiment tests the bandwidth in both scenarios. Figure 5-8 shows the results. We measure the bandwidth continuously over 200 seconds. The corresponding y-axis of all blue lines in this figure is on the left, measured in Mbps. The corresponding y-axis of the green line is on the right, measured in Kbps. Figure 5-8 shows that the quality of the cloud-based network is better. The link between the two VMs (objects 3 and 4) provisioned by our Cloud engine use a cloud-based network which exhibits superior bandwidth. If we deploy the application without our solution, data collectors are needed to directly connect to the remote server. Two lines in Figure 5-8 with "S1" denote the performance. Although object 2 is in a good network environment, the average bandwidth is 26 Mbps less when it is directly connected to the remote server. Moreover, it is obvious that the bandwidth of the cloud-based network is more stable. In addition, the green line shows that when data collectors do not have good network access, the bandwidth is much worse.

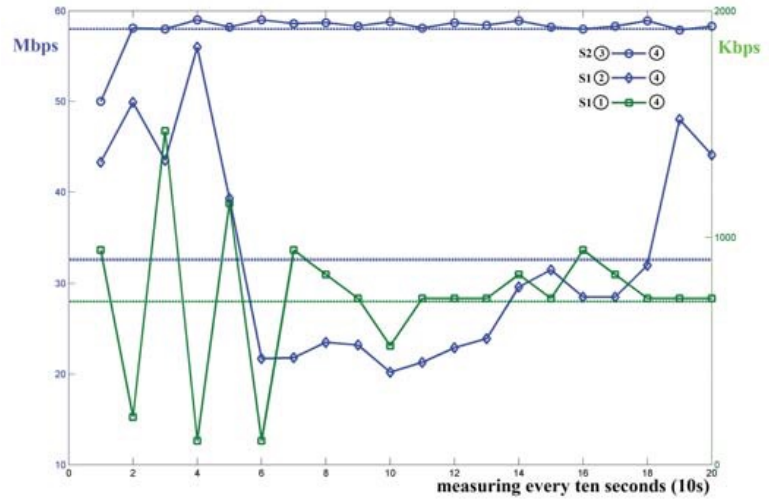

Figure 5-8 Bandwidth comparison.

The transmission time for data collectors can therefore be reduced using our solution. Our engine can set up a virtual Cloud that considers the underlying network in order to better satisfy the nearly real-time requirements of the application to the extent that it is possible. This kind of consideration is essential for data collectors to work more efficiently as part of a larger distributed system.

5.4 Summary

There are several innovations demonstrated by the DRIP provisioner. These innovations can help satisfy the requirements of time-critical applications generated using SWITCH.

1. **Fast and flexible**. Multiple smaller infrastructures can be provisioned with less overhead. If some part of the infrastructure crashes, we just need to re-provision the smaller sub-infrastructure containing the failed component, not the whole aggregate infrastructure. This property can minimise violations of the real-time constraints of some quality-critical applications. Flexibility in where parts of the application are provisioned can also help satisfy any geographic requirements of the application.
2. **Flexible scaling**. As cloud providers often have limitations on the scale of infrastructure provisioned for a particular application, our mechanism puts forward a way to provision large-scale infrastructure across multiple domains. The infrastructure can then even scale across cloud providers.
3. **Transparency**. Our mechanism is not only transparent to cloud providers but also to application developers. From the providers' point of view, there is nothing required of them to support this kind of provisioning. From the point of view of developers, the infrastructure is provisioned as designed, including selected IP addresses, the precise locations of components hidden in the network configuration. It is also transparent to use with tools like Apache Hadoop[26] or Spark[27], as long as they are configured with the proper private IP addresses.
4. **Standardised infrastructure level auto-provisioning**. The Cloud engine only takes as input description files like those described in Section 5.2 The files are human readable and can be written compatible with the emerging TOSCA standard. Hence, they are easy to standardise. Compared with other automatic provisioning tools, it not only provisions the separate instances but also the network as defined by the user. Moreover, the application can be installed and run automatically after the infrastructure is provisioned.

With this provisioning engine, application developers can design and deploy their applications on an inter-locale virtual Cloud. The results of the simple experiments we have so far conducted demonstrate the feasibility and potential efficiency of our solution, though still many challenges must be tackled in order to truly support multi-cloud environments.

[26] http://hadoop.apache.org/
[27] https://spark.apache.org/

6 Deadline-aware deployment for SWITCH applications

For DRIP, we propose a Deadline-aware Deployment System (DDS) for time-critical applications in clouds which accounts for deadlines on the actual deployment time of application components. DDS enables users to automatically deploy time-critical applications and provide scheduling mechanisms to guarantee deployment deadlines. First, DDS helps users to create a local repository for application components instead of using a remote repository, providing a guarantee of bandwidth for transmitting application packages where the transmission rate directly from the remote repository is widely varying. To be deadline-aware, DDS schedules deployment requests based on Earliest Deadline First (EDF) [Liu and Layland, 1973] which is a classical scheduling technique to minimise the number of deployments that miss deadlines. Furthermore, we design bandwidth-aware EDF to facilitate DDS to satisfy a greater number of deadline requirements and achieve sufficient utilisation of bandwidth. In the evaluation, we demonstrate that DDS significantly reduces the number of deployments that miss deadlines, and leverages bandwidth sufficiently.

The research and development results of this section have been published in International Conference Euro-Par [Hu et al., 2017]

We summarise our contributions as follows:

- We designed and implemented DDS, a deadline-aware deployment system which can support automatic deployments of time-critical applications in clouds.
- We built on DDS to implement deployment scheduling algorithms that minimise the number of deployments that miss deadlines and maximize the utilisation of bandwidth.
- We experimentally evaluated the benefits of DDS on the ExoGENI test-bed and large-scale simulations by comparing it with three different scheduling techniques.

6.1 Problem specification

A typical scenario for deploying distributed applications in Clouds involves two basic steps: transmitting necessary application packages or software components from remote repositories to virtual machines (VMs) in the provisioned infrastructure; and installing the software once runnable. Containers, e.g. built using Docker [Merkel, 2014], are the default way to wrap application components in SWITCH.

For a distributed application, the deployment service has to know the location of application components, and the location to deploy (VMs) for each component. Those container images are often stored in a repository, e.g. Docker hub, that is not a part of the provisioned virtual infrastructure. The deployment service should schedule the sequence of each component based on the application description for transmitting and installing each individual component. The time for deploying a single container (T_d) typically contains time cost for transmitting the component from its repository T_f) and installing (extracting files from the Docker image) the component T_i). The total time of the deployment of the whole application starts from the first component transmission until the last component finishes its installation. When an application contains more components, careless scheduling of the deployment sequence might lead to a high time cost, which can eventually influence the execution of the application if key application components are delayed during deployment.

T_f depends on the size of the container and the network bandwidth between repository and target. T_i mainly depends on the performance of the VM and the complexity of the container itself. In many cases, T_f is much bigger than T_i. Table 6-1 shows some observations in the ExoGENI Cloud environment [Baldin et al., 2016]. We created VMs which are 'xo.medium' configuration in three different locations: Boston, Washington and Houston. We found that T_f is widely varying because the internet connection between VMs and Docker hub is different between different locations, and T_i is stable for the same VM configurations. For meeting the deployment time constraints of time-critical applications in provisioned virtual infrastructure, the key challenge is how to minimise the transmission time T_f and predict the installation time T_i. Installation time prediction is not the focus of this section—we assume that existing predictors [Smith et al., 1998] can

achieve good estimations of installation time. Instead, we focus on the transmission process (T_f) of deployment.

Table 6-1 Comparison of transmission time and installation time in different locations.

Docker Image	Image Size	Boston Rack	Washington Rack	Houston Rack
ubuntu	400Mb	$T_f : 40.8s(\pm 2.2s)$ $T_i : 6.3s(\pm 0.5s)$	$T_f : 27.0s(\pm 1.5s)$ $T_i : 6.4s(\pm 0.4s)$	$T_f : 20.3s(\pm 1.5s)$ $T_i : 6.3s(\pm 0.6s)$
nginx	576Mb	$T_f : 58.7s(\pm 2.5s)$ $T_i : 9.3s(\pm 0.7s)$	$T_f : 38.9s(\pm 2.6s)$ $T_i : 9.1s(\pm 0.5s)$	$T_f : 29.2s(\pm 1.8s)$ $T_i : 9.3s(\pm 0.6s)$
mongodb	1200Mb	$T_f : 122.4s(\pm 3.0s)$ $T_i : 15.4s(\pm 0.5s)$	$T_f : 81.0s(\pm 3.4s)$ $T_i : 15.7s(\pm 0.8s)$	$T_f : 60.9s(\pm 1.9s)$ $T_i : 15.5s(\pm 0.8s)$
cassandra	1296Mb	$T_f : 132.2s(\pm 3.1s)$ $T_i : 17.1s(\pm 0.9s)$	$T_f : 87.5s(\pm 3.4s)$ $T_i : 17.3s(\pm 0.7s)$	$T_f : 65.7s(\pm 2.3s)$ $T_i : 17.4s(\pm 0.6s)$

The deployment model in this work is a set of deployment requests. The deployment service has to optimise the time cost by scheduling component transmissions carefully, and parallelise the data transfer based on the time constraint obtained from the application. We model the deployment request as a tuple $R_i = (v_i, s_i, d_i)$, where v_i is the target virtual machine to deploy request R_i, s_i is the application size (e.g. in Mb), and d_i is its deadline. As we concentrate on transmission, we model bandwidth information for provisioned VMs as sets $B = \{b_1, b_2, b_3, \dots, b_n\}$, where b_i denotes the bandwidth of virtual machine i. This means that the *throughput* of virtual machine i cannot exceed b_i during the transmission process, and the bandwidth is stable based on the SLA provisioning mechanisms [Casalicchio and Silvestri, 2013] in this context. We denote the bandwidth of the target machine v_i as b_j, so that the transmission time of request R_i can be represented as $T_f = (s_i/b_j)$. Similarly, the deployment time can be represented as $T_d = (s_i/b_j) + T_i$. The problem of this paper is thus to investigate the scheduling mechanisms needed to meet the deployment deadlines (i.e. ensure that $T_d \leq d_i$) of time-critical applications in clouds.

6.2 Methodology and implementation

DDS aims to provide a deadline-aware, efficient and automatic deployment system that supports time-critical applications on infrastructure as a service on cloud systems, focusing on the network of the underlying distributed system to provide the best guarantee for deployment within deadlines. We follow a number of design principles:

1. **Repository location**. The repository for the application is a shared storage from which application packages can be fetched to be installed on another machine. The repository can be located in a remote server or in the cloud already. The location of the repository can directly impact the deployment time because the network bandwidth between cloud VMs and between a VM and a remote repository in a different location can be very different. Compared to a remote repository, a local repository within a cloud has some obvious advantages. First, the local repository has greater transmission capacity than the remote repository. Second, the bandwidth of the local repository inside a cloud is more stable, which provides a guarantee regarding the transmission time. Third, the local repository is more flexible due to the possibility of personalized configuration. Thus, DDS would help users to create a local repository first if there is only a remote repository from which to fetch application packages.

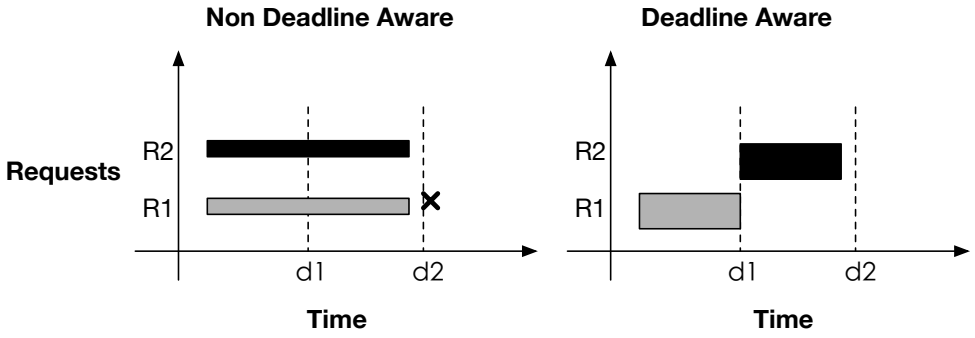

Figure 6-2 Awareness of deadlines can be used to meet two deadlines.

2. **Deadline-aware mechanism**. As the goal of DDS to meet the deadline of requests, whether the system is aware of the deadline is important for deployment. Consider a common time-critical application scenario involving two deployment requests sent to the same application component provider simultaneously, where one request has a tighter deadline than the other. The resulting requests share a bottleneck via which to transmit application packages. As shown in Figure 6-2, with today's setup, the transport protocol (e.g. TCP) strives for fairness and the transmission finishes for both requests almost simultaneously. However, only one of the requests meets its deadline which makes the another request useless or degrades its value. Alternatively, given explicit information about deployment deadlines, the system can arrange the transmission order to better meet the deployment deadline.

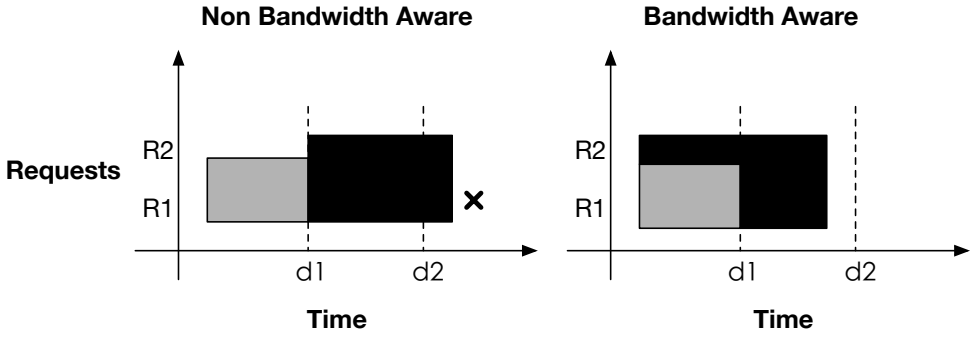

Figure 6-3 Awareness of bandwidth can be used to meet two deadlines.

3. **Bandwidth-aware mechanism**. In addition to deadline-aware scheduling, to be aware of bandwidth is another significant attribute for deployment. Consider another scenario with two deployment requests, where the second request pulls a larger application package. The resulting requests also share a link to transmit their respective packages. As shown in Figure 6-3, the deployment system has information about the deadlines and schedules the transmission based on those deadlines. However only one request meets its deadline. Because the transmission bottleneck is the bandwidth of the target machine, there is some spare bandwidth on the server which is not used. Thus, given explicit information about the bandwidth capacity of each machine in the cloud, the system could schedule more deployment requests and leverage the bandwidth more efficiently.

The main goal of our algorithms is to minimise the deadline miss rate: the application packages should be transmitted to the target machine within the deadline wherever possible. In addition to minimising miss rate, we should maximize the bandwidth utilisation to reduce the total transmission time. To achieve both these goals, we employ EDF to prioritise requests and design bandwidth-aware EDF to support parallel transmission and realise dynamic rate control.

1. **EDF scheduling**. The key insight guiding the design of deadline-aware scheduling is derived from the classic real-time scheduling algorithm Earliest Deadline First (EDF) [Liu and Layland, 1973], which prioritises tasks based on their deadline. EDF is an optimal scheduling algorithm in that if a set of deadlines can be satisfied under some schedule, then EDF can satisfy them too. We adopt EDF to schedule deployment requests. When a deployment request comes, DDS compares the deadline of new request with previous requests and then sets the corresponding priority relative to the other deadlines. DDS then puts the new request into the request queue where the requests are sorted by priority. The algorithm is described in Algorithm 1. Consequently, DDS obtains the request from the queue and starts to transmit application packages to the target machine.

Algorithm 1 EDF scheduling

Input: The new deployment request Ri
Output: The request queue RQ where requests sorted by the deadline

```
for each Rj ∈ RQ do
    if Ri.deadline < Rj.deadline then
        RQ.insert(Ri)
        return RQ
    end if
end for
RQ.append(Ri)
return RQ
```

2. **Bandwidth-aware EDF scheduling**. In addition to EDF scheduling, we design bandwidth-aware scheduling in cooperation with EDF scheduling. The key idea of bandwidth-aware scheduling is to make use of the spare bandwidth available between the local repository and the target as much as possible for parallelising multiple requests. Thus, DDS needs the bandwidth information for each machine in the cloud. DDS would collect the bandwidth information before the whole deployment procedure begins.

Algorithm 2 Bandwidth-aware EDF scheduling

Input: $throughput$ and $bandwidth$ of the local repository

```
while throughput < bandwidth do
    if RQ ∉ ∅ then
        R_i = RQ.pop()
        b_j = GetBandwidth(v_i)
        if throughput + b_j < bandwidth then
            throughput = throughput + b_j
        else
            SetTransmissionRate(R_i, bandwidth − throughput)
            throughput = bandwidth
        end if
        StartTransmission(R_i)
    end if
end while
return
```

EDF is optimal when the deadlines can be satisfied. However, without bandwidth information, EDF would schedule requests in a sequential way which leads to insufficient utilisation of bandwidth or even missed deployment deadlines. However if we directly schedule requests in a parallel way, the bandwidth contention among different requests can also cause deployment deadlines to be missed. Therefore, the challenge of bandwidth-aware scheduling is how to dynamically allocate transmission rates for deployment requests in

order to avoid unnecessary contention. For this purpose, we design bandwidth-aware EDF algorithm as described in Algorithm 2.

As per the description of bandwidth-aware EDF, if there is spare bandwidth in the local repository, DDS will continue to obtain requests from the request queue until the required bandwidth is equal or greater than the local repository bandwidth. DDS then sets the specific rate for the last deployment request to make sure the total required bandwidth is equal to the bandwidth of local repository. Consequently, it avoids bandwidth contention with previous deployment requests and makes full use of spare bandwidth to transmit. Once a new deployment request arrives, DDS performs bandwidth-aware EDF scheduling after putting the request in the request queue. When one deployment request finishes, DDS will allocate the released bandwidth for the running requests first, and then perform bandwidth-aware EDF scheduling again.

6.3 Evaluation

In this section, we describe experiments for quantitative evaluation of the deadline-aware deployment system. We perform three kinds of experiments. First, we evaluate the transmission time using a DDS local repository versus a remote repository. Second, we evaluate DDS in comparison with three typical scheduling algorithms by running experiments on our cloud test-bed. Third, we evaluate DDS in larger-scale simulations.

6.3.1 Repository Evaluation

In this section, we compare the transmission time to a target machine from a DDS local repository and a remote repository based on Docker. In most common cases, the application provider only has the repository outside cloud. Thus, DDS would help users to create local repository within their cloud first. We provision two virtual machines with 50Mbps bandwidth in the ExoGENI Boston rack and create a local repository in one of them. Then, we use the other machine to fetch the image from the local repository and also the original remote repository (Docker Hub). The comparative results are shown in the Table 6-2. Note that the transmission time (T_f) from the local repository is much less than from the remote repository, the reason being that the bandwidth inside Cloud is much better than outside.

Table 6-2 Comparison of transmission times from different repositories.

Docker Image	Image Size	Local Repository	Remote Repository
ubuntu	400Mb	$T_f : 8.1s(\pm 1.1s)$	$T_f : 40.8s(\pm 2.2s)$
nginx	576Mb	$T_f : 11.7s(\pm 1.3s)$	$T_f : 58.7s(\pm 2.5s)$
mongodb	1200Mb	$T_f : 24.4s(\pm 1.2)$	$T_f : 122.4s(\pm 3.0s)$
cassandra	1296Mb	$T_f : 26.4s(\pm 1.5)$	$T_f : 132.2s(\pm 3.1s)$

6.3.2 Test-bed experiments

In this section, we evaluate DDS alongside three typical scheduling algorithms in the ExoGENI test-bed. ExoGENI is a networked infrastructure-as-a-service (NIaaS) platform where researchers can define the network topology and bandwidth of virtual infrastructures. In our experimental setup, we chose the "xo.xlarge" type of machine as our local repository, and all other application nodes we chose "xo.medium" type machines. The guest OS in VMs which are provisioned for evaluation is Ubuntu 14.04. In the experiment, we use *iPerf* [Tirumala et al., 2005] to simulate the application package transmission, therefore the size of application package can be customised via *iPerf* in the evaluation. For transmission rate control, we leverage Linux Traffic Control (TC) to perform deployment request rate limiting. We use two-level Hierarchical Token Bucket (HTB) in TC: the root node classifies requests to their corresponding leaf nodes based on IP address and the leaf nodes enforce each request rate. We compare the following schemes with DDS:

- **FIFO:** All the deployment requests are scheduled by the arrival time of the request in a sequential way.
- **EDF:** All the deployment requests are scheduled by the EDF algorithm in a sequential way.
- **PARALLEL:** All the deployment requests are scheduled immediately after arrival in a parallel way.

Through comparison with these three schemes, we can inspect the benefits from DDS for different aspects. FIFO is the most common scheduling algorithm in distribution. EDF is optimal in sequential scheduling when the deadline can be satisfied, but it is not bandwidth-aware. PARALLEL can make high utilisation of the bandwidth, but it is not deadline-aware. We compare the number of schedulable requests (requests that meet the deadline) and the total deployment time among different schemes. The number of schedulable requests can indicate the satisfaction of deadline requirements. The total deployment time can indicate the utilisation of network bandwidth.

In this experiment, we provision two kinds of bandwidth configuration to evaluate DDS as described by Table 6-3. We instantiate four nodes to deploy time-critical applications in ExoGENI. For these four nodes, we generate six deployment requests which include the target machine, application size, arrival time and the deadline. To understand the scheduling mechanisms in DDS better, we assume that the installation time T_i of each application is 1s in this experiment.

Table 6-3 Bandwidth configuration (left) and deployment requests (right).

(a) Configuration A (Mbps)

Repository	Node1	Node2	Node3	Node4
100	20	50	70	100

(b) Configuration B (Mbps)

Repository	Node1	Node2	Node3	Node4
100	70	70	70	70

Machine	Size	Deadline	Arrival Time
Node1	200Mb	14s	0s
Node1	160Mb	20s	10s
Node2	320Mb	9s	11s
Node2	560Mb	15s	30s
Node3	960Mb	20s	30s
Node4	640Mb	25s	30s

In Figure 6-4 (left), we inspect the number of schedulable requests on different schemes. We observe that DDS can schedule more requests in two different bandwidth configurations, because sequential scheduling (EDF, FIFO) can not meet all the deadlines when multiple requests emerge simultaneously, and direct parallel scheduling suffers from bandwidth contention. Figure 6-4 (right) shows the total deployment time of various schemes. We note that the total deployment time of DDS is less than EDF and FIFO, and similar to PARALLEL. This indicates that DDS makes full use of network bandwidth.

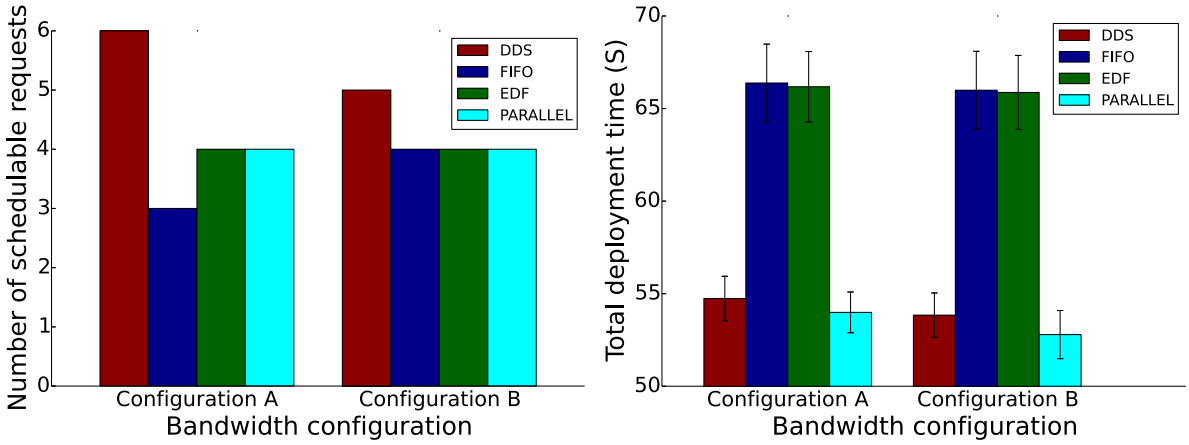

Figure 6-4 Comparison of the number of schedulable requests in various schemes (left) and the total deployment time in various schemes (right).

6.3.3 Large-scale simulations

Our simulations evaluate DDS considering the common public cloud providers (EC2, Azure). We evaluate the deployment schedulable ratio which is the percentage of schedulable requests in different schemes.

- **VM configuration.** We equip the deployment server with 10Gbps bandwidth connection and application node with 1Gbps bandwidth connection which are typical configuration in public cloud. In the simulation, the number of application nodes range over 10, 20, 40 and 80 nodes which are sufficient to account for most distributed cloud applications.
- **Deployment requests.** We simulate the deployment service running 10 days ($T_{running}$) in the experiment. During this period, we generate deployment requests in different densities to simulate deploying various applications on each node. We denote S^i_{total} as the total application size of all deployment requests on node i. The request density of node i is equal to $S^i_{total}/(T_{running} \times 10\ Gigabit)$, and the request density of whole system is the average for each node. The overall request density varies from 0.1 to 0.9. In the experiment, the deadline (d_i) of each request ranges from 10s to 100s, and the application size is equal to $d_i \times 1\ Gigabit$. We assume the installation time (T_i) is 1s in the simulation.

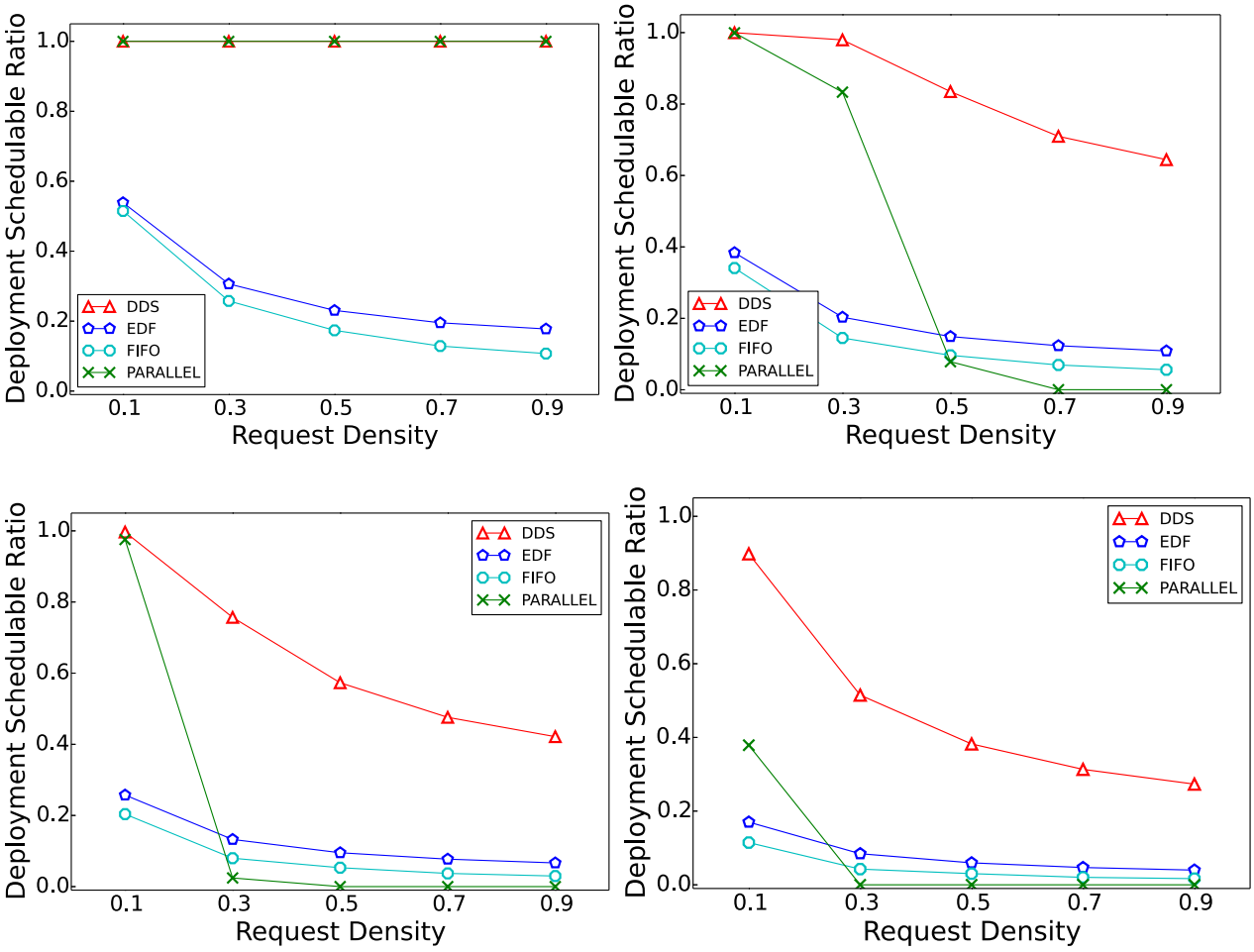

Figure 6-5 Comparison of the deployment schedulable ratio for 10 nodes (top left), 20 nodes (top right), 40 nodes (bottom left) and 80 nodes (bottom right).

Figure 6-5 shows the deployment schedulable ratio in different scenarios. We observe that DDS can reduce from 24% to 83% of the deployment deadline miss ratio compared to EDF, from 26% to 89% compared to FIFO, and up to 86% compared to PARALLEL. Because EDF and FIFO schedule deployment requests in

sequential way, DDS can take advantage of parallelised deployments. The PARALLEL scheme parallelises deployments but suffers severe bandwidth contention as request density increases. In contrast, DDS is bandwidth-aware and provides dynamic transmission rate control to avoid bandwidth contention for different deployment requests. In summary, DDS significantly reduces the number of deadline missing requests for deploying cloud applications.

6.4 Summary

In recent years, deployment has been an important topic in distributed environment, service-oriented systems and cloud computing, as well as in some of SWITCH's contemporary projects such as ENTICE[28]. The techniques in DDS are related to the following areas of research:

- **Automatic cloud application deployment.** To enable automatic deployment has been the focus of several recent works. SO-MVDS [Gao et al., 2012] allows users to design and create virtual machines with specific services running in them and define a service deployment request to enhance the efficiency of service deployment. Li et al. [Li et al., 2012] propose a general approach to application deployment. They adopt contextualisation process which is to embed various scripts in VM images to initiate applications. DDS, on the other hand, is compatible with Docker containers, achieving automatic deployment more easily.
- **On-demand image distribution.** The idea of distributing images in clouds efficiently has been explored in recent works. [Vaquero et al. 2015] proposes a solution based on combining hierarchical and Peer to Peer (P2P) data distribution techniques. VDN [Peng at al., 2012], a new VM image distribution network on the top of chunk-level, enables collaborate sharing in cloud data centres. These approaches focus on fast transmission. In contrast, DDS is not only transmitting images efficiently but is also aware of deadlines via scheduling mechanisms.
- **Deadline-aware scheduling techniques.** D^3 [Wilson et al., 2011] and D^2 TCP [Vamanan et al., 2012] are transport protocols designed for deadline-aware transmission inside data centres. These protocols add the deadline information to TCP and provide control mechanisms based on the deadline information. Techniques like Karuna [Chen et al., 2016] and pFabric [Alizadeh et al., 2013] prioritise network flows to transmit. All these approaches schedule transmission at flow level. In contrast, DDS exploits the information of bandwidth to schedule transmission in application level which is more relevant to users requirements.

It is challenging to deploy time-critical applications into clouds while meeting the time constraints of deployment. This is an important and practical problem, but has been neglected by prior work in this field. For DRIP we use DDS for the deployment agent component in order to help users to create local repositories and automatically deploy applications into Clouds. We have investigated the scheduling mechanisms in cloud deployment systems and implemented a bandwidth-aware EDF scheduling algorithm in DDS. DDS schedules deployment requests based on deadline and bandwidth information to make better scheduling decision. In the evaluation, we showed that DDS leverages network resources sufficiently and significantly reduces the number of missed deployment deadlines.

[28] http://www.entice-project.eu/

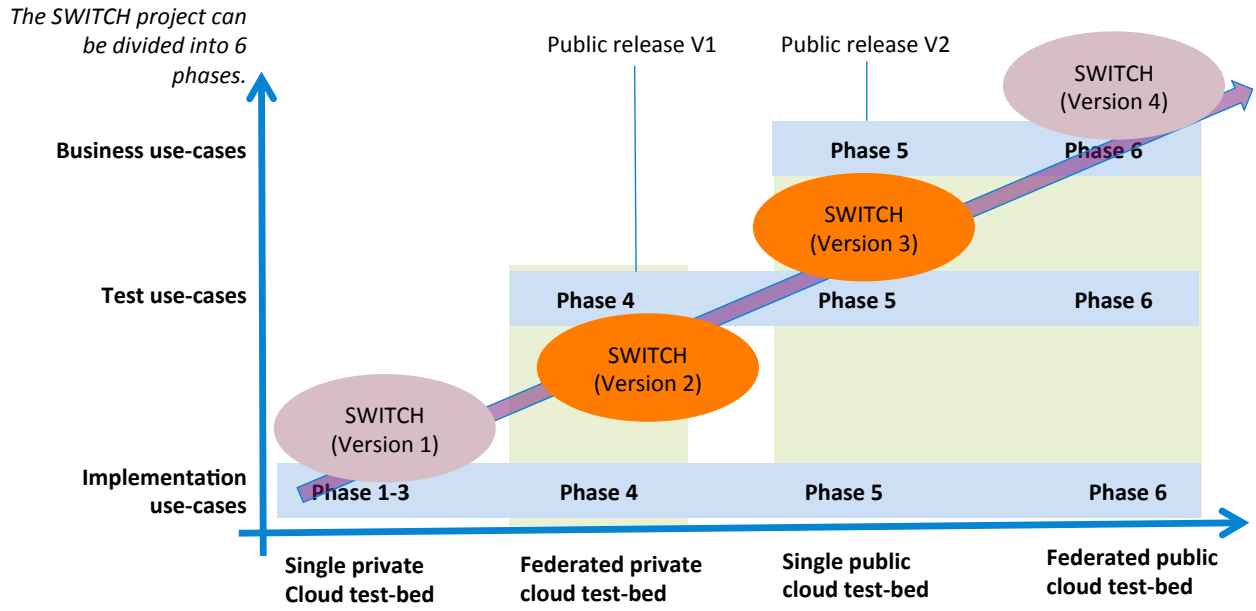

Figure 7-1 Overview of SWITCH development phases.

7 Summary

The development of DRIP has been conducted in accordance with the basic plan illustrated by Figure 7-1. At the time of publication, the project is entering the fifth phase of development, with the integration of the second release of the SWITCH workbench, providing tools for provisioning and controlling time-critical applications on both private and public clouds, and indeed offering the basic support for inter-cloud provisioning needed for the final 'federated public cloud test-bed' objective.

7.1 Software functionality in public releases

With regards to DRIP, the second release of the SWITCH workbench will contain a fully functional and integrated DRIP service suite, managed via a single online manager component. In particular, within this deliverable we have reviewed four main topics of research and innovation within the SWITCH project over the prior twelve months that serve to directly contribute to the development of DRIP:

- The extension of the DRIP planner algorithm MEPA to support planning of the placement of SDN controllers in software-defined networks.
- The prototyping of a performance modelling service for collecting information about the performance of cloud resources for different kinds of application component, important for helping DRIP determine the best selection of resources for a range of different applications with different time-critical constraints.
- The further refinement and experimental evaluation of the DRIP provisioner, supporting multi-site provisioning across multiple data centres.
- The development of a deployment agent for DRIP that can maximise use of bandwidth to expedite the retrieval and installation of application components from remote repositories.

Architecture components (defined in D2.2)	Functionality in V1	Functionality in V2	Key Performance Indicators (KPI)	Current status
DRIP manager	Yes	Yes	Scalability and reliability	Achieved KPI
Application interpreter	Yes (part of DRIP manager)	Yes (part of DRIP manager)	Functionality	Achieved KPI
Infrastructure Planner	Yes	Yes	Support for wide range planning constraints	Partial: planner supported mainly time related constraints, e.g., performance and deadline-based.
Infrastructure interpreter	Yes (inside DRIP manager)	Yes (inside DRIP manager)	Functionality	Achieved KPI
Infrastructure evaluator	No.	Yes	Accuracy	Partial: current version can do correctness evaluation. Performance fitness is still under research.
Discovery service	Yes (Inside DRIP knowledge base)	Yes (Inside DRIP knowledge base)	Supported providers.	Implemented support for ExoGENI, Amazon EC2 and FedCloud. More will be included
Infrastructure Provisioner	Yes	Yes	Provisioning time, failure recovery time.	Achieved: analysis of current provisioner provisioning time in [Zhou et al., 2016].
Resource selector	Yes (In planner)	Yes (Part in planner, and part as performance modeller)	Supported cloud providers.	Support for ExoGENI, Amazon EC2 and EGI FedCloud.
Cloud broker	Yes (Inside infrastructure provisioner)	Yes (Inside infrastructure provisioner)	Supported providers.	Support for ExoGENI, Amazon EC2 and EGI FedCloud.

SLA Negotiation	Partially in performance modeller; however, the negotiation part will be our research topic and not in software. We explained the situation during review meeting.		N/A	On our research agenda, to be finished soon.
SWITCH executor	Yes (deployment agent, and execution for container)	Yes (Control agents for container and VM, deployment agent with real-time).	Deployment time, repository support, deadline support and liveness.	Achieved: analysis of deployment agent capabilities in [Hu et al., 2017]. The liveness (fault tolerance) is jointly with provisioner.

7.2 Innovation

The innovation of the DRIP system lies in its support for time-critical concerns in the planning and provisioning of virtual infrastructure and the deployment and execution of application components on such infrastructure. For each major component of DRIP, we can identify a specific key innovation over the existing state of the art.

Component (in release)	**Current state of the Art**	**Innovation**
DRIP manager	Many suites provide integrated facilities for Cloud planning and provisioning.	Support for scalable services via use of **message queuing to distribute workloads automatically**.
Infrastructure planner	Support for single deadlines for complete application workflows based on critical path analysis.	Support for **multiple deadlines** on application workflows.
DRIP performance modeller	Clouds publish the attributes of the resources they offer, which clients must evaluate against their requirements.	A framework for **automatic testing** of resources against different kinds of application component; ability to aggregate information within DRIP knowledge base in order to improve planning.
Infrastructure provisioner	Ability to provision a given application in a single cloud environment, or for multiple clouds with manual network configuration.	Support for **multi-locale provisioning** with a single seamless network topology handled automatically by the provisioner **without intervention by client or cloud provider**.
DRIP deployment agent	Support for application component retrieval from remote repositories.	Support for **optimal use of network when retrieving remote components to meet deployment deadlines**.

The DRIP subsystem is used in the project together with the other two to implement the industrial pilot cases. Besides which DRIP has also been exploited in EU H2020 ENVRIPLUS project for optimising data services in e-Infrastructures. A finished use case is to enhance the data subscription service of European EURO-ARGO[29] research infrastructure for generating data products for distributed partners. EGI FedCloud is used as a test-bed. The demo of the use case is available on YouTube[30]. A research paper has also been submitted to the IEEE e-Science conference. In addition there are a number of other on-going use cases, e.g., for optimising data processing workflows in EISCAT_3D[31] and EPOS[32].

In the last phase of the project, exploiting DRIP within the integrated SWITCH will be highlighted. A detailed exploitation plan and report will be presented in D6.4 “Report on dissemination, communication, collaboration, exploitation and standardization V3”.

[29] http://www.euro-argo.eu
[30] https://www.youtube.com/watch?v=PKU_JcmSskw&t=19s
[31] http://www.eiscat.se
[32] http://www.epos-ip.eu

Bibliography

[Abd-El-Barr, 2009] M. Abd-El-Barr, “Topological network design: A survey,” Journal of Network and Computer Applications, vol. 32, no. 3, pp. 501–509, 2009.

[Abrishami et al., 2013] S. Abrishami, M. Naghibzadeh, and D. Epema. “Deadline-constrained work- flow scheduling algorithms for infrastructure as a service clouds”. Future Generation Computer Systems, 29(1):158–169, 2013.

[Alamri et al., 2013] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain, “A survey on sensor-cloud: architecture, applications, and approaches,” International Journal of Distributed Sensor Networks, vol. 2013, 2013.

[Alizadeh et al., 2013] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, S. Shenker. “pfabric: Minimal near-optimal datacenter transport”. In: ACM SIGCOMM Com- puter Communication Review. vol. 43, pp. 435–446. ACM (2013)

[Baldin et al, 2016] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, J. Mills. “ExoGENI: A multi-domain infrastructure-as-a-service testbed”. In: The GENI Book, pp. 279–315. Springer (2016)

[Canon et al., 2008] L-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. “Comparative evaluation of the robustness of DAG scheduling heuristics”. In Grid Computing, pages 73–84. Springer, 2008.

[Cai et al., 2013] Z. Cai, X. Li, and J. N.D. Gupta. “Critical path-based iterative heuristic for workflow scheduling in utility and cloud computing”. In International Conference on Service- Oriented Computing, pages 207–221. Springer, 2013.

[Cai et al., 2016] Z. Cai, X. Li, and J. N. D. Gupta. “Heuristics for provisioning services to workflows in XaaS clouds”. IEEE Transactions on Services Computing, 9(2):250–263, 2016.

[Casalicchio and Silvestri, 2013] E. Casalicchio, L Silvestri. “Mechanisms for SLA provisioning in cloud-based service providers”. Computer Networks 57(3), 795–810 (2013).

[Chen et al., 2016] L. Chen, K. Chen, W. Bai, M. Alizadeh. “Scheduling mix-flows in commodity data centers with Karuna”. In: Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference. pp. 174–187. ACM (2016)

[Cheng et al., 2015] T.Y. Cheng, M. Wang, and X. Jia, “QoS-guaranteed controller placement in SDN,” in 2015 IEEE Global Communications Conference (GLOBE-COM), pp. 1–6, IEEE, 2015.

[Chhetri et al., 2013] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, “Smart cloudbench–automated performance benchmarking of the cloud,” in 2013 IEEE Sixth International Conference on Cloud Computing. IEEE, 2013, pp. 414–421.

[Convolbo and Chou, 2016] M. Convolbo and J. Chou. “Cost-aware DAG scheduling algorithms for minimizing execution cost on cloud resources”. The Journal of Supercomputing, 72(3):985–1012, 2016.

[Cordeiro et al., 2010] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J. M. Vincent, and F. Wagner. (2010). “Random graph generation for scheduling simulations”. In Proceedings of the 3rd international ICST conference on simulation tools and techniques (p. 60). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Cunha et al., 2017] M. Cunha, N. Mendonça, and A. Sampaio, “Cloud crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds,” Concurrency and Computation: Practice and Experience, vol. 29, no. 1, 2017.

[Deelman et al., 2009] E. Deelman, D. Gannon, M. Shields, I. Taylor. “Workflows and e-Science: An overview of workflow system features and capabilities”, Future Gener. Comput. Syst. 25 (5) (2009) 528–540.

[Durillo and Prodan, 2014] J. Durillo and R. Prodan. “Multi-objective workflow scheduling in Amazon EC2”. Cluster Computing, 17(2):169–189, 2014.

[Evans et al., 2015] K. Evans, A. Jones, A. Preece, F. Quevedo, D. Rogers, I. Spasić, I. Taylor, V. Stankovski, S. Taherizadeh, J. Trnkoczy, G. Suciu, V. Suciu, P. Martin, J. Wang and Z. Zhao. (2015). “Dynamically reconfigurable workflows for time-critical applications”. In Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science (p. 7). ACM.

[Fencl et al., 2011] T. Fencl, P. Burget, and J. Bilek, “Network topology design,” Control Engineering Practice, vol. 19, no. 11, pp. 1287–1296, 2011.

[Fortin et al., 2012] F. A. Fortin, F. M. D. Rainville, M. A. Gardner, M. Parizeau, and C. Gagné. (2012). “DEAP: Evolutionary algorithms made easy”. Journal of Machine Learning Research, 13(Jul), 2171-2175.

[Gao et al., 2012] W. Gao, H. Jin, S. Wu, X. Shi, J. Yuan. “Effectively deploying services on virtualization infrastructure”. Frontiers of Computer Science 6(4), 398–408 (2012)

[Gódor and Magyar, 2005] I. Gódor and G. Magyar, “Cost-optimal topology planning of hierarchical access networks,” Computers & operations research, vol. 32, no. 1, pp. 59–86, 2005.

[Heller et al., 2012] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” in Proceedings of the first workshop on Hot topics in software defined networks, pp. 7–12, ACM, 2012.

[Hock et al., 2014] D. Hock, S. Gebert, M. Hartmann, T. Zinner, and P. Tran-Gia, “Poco-framework for pareto-optimal resilient controller placement in SDN-based core networks,” in Network Operations and Management Symposium (NOMS), 2014 IEEE, pp. 1–2, IEEE, 2014.

[Hu et al., 2017] Y. Hu, J. Wang, H. Zhou, P. Martin, T. Arie, C. De Laat, and Z. Zhao, “Deadline-aware deployment for time critical applications in clouds,” in 2017 International European Conference on Parallel and Distributed Computing (Euro-Par 2017), 2017.

[Hwang et al., 2016] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu, “Cloud performance modeling with benchmark evaluation of elastic scaling strategies,” IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 1, pp. 130–143, 2016.

[Iosup et al., 2011] A. Iosup, N. Yigitbasi, and D. Epema, “On the performance variability of production cloud services,” in Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on. IEEE, 2011, pp. 104–113.

[Jeong and Figueiredo, 2016] K. Jeong and R. Figueiredo, “Self-configuring software-defined over-lay bypass for seamless inter-and intra-cloud virtual networking,” in Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pp. 153–164, ACM, 2016.

[Kamiyama, 2009] N. Kamiyama, “Efficiently constructing candidate set for network topology design,” in Communications, 2009. ICC’09. IEEE International Conference on, pp. 1–6, IEEE, 2009.

[Kosukhin et al., 2015] S.S. Kosukhin, S.V. Kovalchuk, A.V. Boukhanovsky. “Cloud technology for fore-casting accuracy evaluation of extreme metocean events”, Procedia Comput. Sci. 51 (2015) 2933–2937.

[Kreutz et al., 2015] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015.

[Krzhizhanovskaya et al., 2011] V.V. Krzhizhanovskaya, G. Shirshov, N. Melnikova, R.G. Belleman, F. Rusadi, B. Broekhuijsen, B. Gouldby, J. Lhomme, B. Balis, M. Bubak, et al., “Flood early warning system: design, implementation and computational modules”, Procedia Comput. Sci. 4 (2011) 106–115.

[Lange et al., 2015] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale SDN networks," Network and Service Management, IEEE Transactions on, vol. 12, no. 1, pp. 4–17, 2015.

[Laplane and Ovaska, 2011] P. A. Laplante, and S. J. Ovaska. (2011). "Real-time systems design and analysis: tools for the practitioner". John Wiley and Sons.

[Leitner and Cito, 2016] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public IaaS clouds," ACM Transactions on Internet Technology (TOIT), vol. 16, no. 3, p. 15, 2016.

[Li et al., 1992] C.-L. Li, S. T. Mccormick, and D. Simchi-Levi, "On the minimum-cardinality-bounded-diameter and the bounded-cardinality-minimum-diameter edge addition problems," Operations Research Letters, vol. 11, no. 5, pp. 303–308, 1992.

[Li et al., 2012] W. Li, P. Svärd, J. Tordsson, E. Elmroth. "A general approach to service deployment in cloud environments". In: Cloud and Green Computing (CGC), 2012 Second International Conference on. pp. 17–24. IEEE (2012)

[Liu and Layland, 1973] C. L. Liu, J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". Journal of the ACM (JACM) 20(1), 46–61 (1973)

[Merkel, 2014] D. Merkel. "Docker: Lightweight linux containers for consistent development and deployment". Linux Journal 2014(239), 2 (2014)

[Elzinga et al., 2017] Elzinga, O., Koulouzis, S., Hu, Y., Wang, J., Zhou, H., Martin, P., Taal, A., de Laat, C., and Zhao, Z (2017), Automatic collector for dynamic cloud performance Information, IEEE Networking, Architecture and Storage (NAS), Shenzheng, China, Auguest 7-8, 2017

[Ongaro et al., 2015] F. Ongaro, E. Cerqueira, L. Foschini, A. Corradi, and M. Gerla, "Enhancing the quality level support for real-time multimedia applications in software-defined networks," in Computing, Networking and Communications (ICNC), 2015 International Conference on, pp. 505– 509, IEEE, 2015.

[Park and Jun, 2009] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," Expert Systems with Applications, vol. 36, no. 2, pp. 3336–3341, 2009.

[Peng et al, 2012] C. Peng, M. Kim, Z. Zhang, H. Lei. "VDN: Virtual machine image distribution network for cloud data centers". In: INFOCOM, 2012 Proceedings IEEE. pp. 181–189. IEEE (2012)

[Poslad et al., 2015] S. Poslad, S. E. Middleton, F. Chaves, R. Tao, O. Necmioglu, and U. Bügel. (2015). "A semantic IoT early warning system for natural environment crisis management". IEEE Transactions on Emerging Topics in Computing, 3(2), 246-257.

[Rodriguez and Buyya, 2014] M. A. Rodriguez and R. Buyya. "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds". IEEE Transactions on Cloud Computing, 2(2):222–235, 2014.

[Rosenburg, 2005] E. Rosenberg, "Hierarchical topological network design," IEEE/ACM Transactions on Networking (TON), vol. 13, no. 6, pp. 1402–1409, 2005.

[Scheuner et al., 2014] J. Scheuner, P. Leitner, J. Cito, and H. Gall, "Cloud work bench– infrastructure-as-code based cloud benchmarking," in Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on. IEEE, 2014, pp. 246–253.

[Shah et al., 2013] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi, "An architectural evaluation of sdn controllers," in Communications (ICC), 2013 IEEE International Conference on, pp. 3504–3508, IEEE, 2013.

[Silva et al., 2013] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. Da Silva, "Cloudbench: experiment automation for cloud environments," in Cloud Engineering (IC2E), 2013 IEEE International Conference on. IEEE, 2013, pp. 302–311.

[Smith et al., 1998] W. Smith, I. Foster, V. Taylor. "Predicting application run times using historical information". In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 122–142. Springer (1998)

[Tirumala et al., 2005] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, K. Gibbs. "Iperf: The TCP/UDP bandwidth measurement tool". http://dast.nlanr.net/Projects (2005)

[Tsai and Rodrigues, 2014] C.-W. Tsai and J. J. Rodrigues, "Metaheuristic scheduling for cloud: A survey," Systems Journal, IEEE, vol. 8, no. 1, pp. 279–291, 2014.

[Tuba, 2010] M. Tuba, "An algorithm for the network design problem based on the maximum entropy method," in Proceedings of the American Conference on Applied Mathematics, Cambridge, USA, pp. 206–211, 2010.

[Vamanan et al., 2012] B. Vamanan, J. Hasan, T. Vijaykumar. "Deadline-aware datacenter TCP (D2TCP)". ACM SIGCOMM Computer Communication Review 42(4), 115–126 (2012)

[Vaquero et al., 2015] L.M. Vaquero, A. Celorio, F. Cuadrado, R. Cuevas. "Deploying large-scale datasets on-demand in the cloud: treats and tricks on data distribution". IEEE Transactions on Cloud Computing 3(2), 132–144 (2015)

[Wang et al. 2017a] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao, "Planning virtual infrastructures for time critical applications with multiple deadline constraints," Future Generation Computer Systems, 2017.

[Wang et al. 2017b] J. Wang, C. de Laat, and Z. Zhao, "Qos-aware virtual sdn network planning," in Proceedings of IFIP/IEEE International Symposium on Integrated Network Management. IEEE, 2017.

[Wilson et al. 2011] C. Wilson, H. Ballani, T. Karagiannis, A. Rowtron. "Better never than late: Meeting deadlines in datacenter networks". In: ACM SIGCOMM Computer Communication Review. vol. 41, pp. 50–61. ACM (2011).

[Yu et al., 2005] J. Yu, R. Buyya, and C.K. Tham. "Cost-based scheduling of scientific workflow applications on utility grids". In First International Conference on e-Science and Grid Computing (e-Science'05). IEEE, 2005.

[Zhao et al., 2011] Z. Zhao, P. Grosso, J. van der Ham, R. Koning, and C. de Laat. (2011). "An agent based network resource planner for workflow applications". Multiagent and Grid Systems, 7(6), 187-202.

[Zhao et al., 2015] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses et al., "Developing and operating time critical applications in clouds: The state of the art and the switch approach," Procedia Computer Science, vol. 68, pp. 17–28, 2015.

[Zhao et al., 2016] Z. Zhao, P. Martin, C. de Laat, K. Jeffery, A. Jones, I. Taylor, A. Hardisty, M. Atkinson, A. Zuiderwijk-van Eijk, Y. Yin, Y. Chen, "Time critical requirements and technical considerations for advanced support environments for data-intensive research," in 2nd International workshop on Interoperable infrastructures for interdisciplinary big data sciences (IT4RIs 16), in the context of IEEE Real-time System Symposium (RTSS), Porto, Portugal, 2016.

[Zhang et al., 2016] Z. Zhang, D. Li, and K. Wu, "Large-scale virtual machines provisioning in clouds: challenges and approaches," Frontiers of Computer Science, vol. 10, no. 1, pp. 2–18, 2016.

[Zhou et al., 2016a] H. Zhou, Y. Hu, J. Wang, P. Martin, C. de Laat, and Z. Zhao, "Fast and dynamic resource provisioning for quality critical cloud applications," in 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2016, pp. 92–99.

[Zhou et al., 2016b] Zhou, H., Hu Y., Wang, J., Martin, P., Su, J., de Laat, C. and Zhao, Z., (2016) Fast Resource Co-provisioning for Time Critical Application Based on Networked Infrastructure, IEEE International Conference on CLOUD (CLOUD) 2016, San Francisco US.

[Zhou et al., 2016c] Zhou, H., Martin, P., Su, J., de Laat, C. and Zhao, Z. (2016) A Flexible Inter-locale Virtual Cloud For Nearly Real-time Big Data Applications, Proceedings of the 2nd International workshop on Interoperable infrastructures for interdisciplinary big data sciences (IT4RIs 16), in the context of IEEE Real-time System Symposium (RTSS), Porto, Portugal, November 29-December 2, 2016.

A Resource API

name	path	methods	description
AnsibleOutputController	/user/v1.0/deployer/ansible/ /user/v1.0/deployer/ansible/all /user/v1.0/deployer/ansible/commands /user/v1.0/deployer/ansible/ids /user/v1.0/deployer/ansible/{id}	GET DELETE GET GET DELETE GET	This controller is responsible for showing the output from ansible executions
BenchmarkController	/user/v1.0/benchmark/ /user/v1.0/benchmark/all /user/v1.0/benchmark/ids /user/v1.0/benchmark/{id}	GET DELETE GET DELETE GET	This controller is responsible for handling cloud benchmark tests like sysbench
CloudConfigurationController0	/user/v0.0/switch/account/configure/ec2 /user/v0.0/switch/account/configure/geni	POST POST	This controller is responsible for handling cloud credentials used by the provisoner to request for resources (VMs).
CloudCredentialsController	/user/v1.0/credentials/cloud/ /user/v1.0/credentials/cloud/all /user/v1.0/credentials/cloud/ids /user/v1.0/credentials/cloud/sample /user/v1.0/credentials/cloud/{id} /user/v1.0/credentials/cloud/upload/{id}	POST DELETE GET GET DELETE GET POST	This controller is responsible for handling CloudCredentials. CloudCredentials are a represntation of the credentials that are used by the provisoner to request for resources (VMs)
ConfigurationController	/user/v1.0/deployer/configuration/all /user/v1.0/deployer/configuration/ids /user/v1.0/deployer/configuration/post /user/v1.0/deployer/configuration/upload /user/v1.0/deployer/configuration/{id}	DELETE GET POST POST DELETE GET	This controller is responsible for storing PlayBook descriptions that can be used by the planner.
DeployController	/user/v1.0/deployer/all /user/v1.0/deployer/deploy /user/v1.0/deployer/ids /user/v1.0/deployer/sampl	DELETE POST GET GET	This controller is responsible for deploying a cluster on provisoned resources.

	e /user/v1.0/deployer/{id}	DELETE GET	
DeployController0	/user/v0.0/switch/deploy/kubernetes /user/v0.0/switch/deploy/swarm	POST POST	This controller is responsible for deploying a cluster on provisoned resources.
KeyPairController	/user/v1.0/keys/ /user/v1.0/keys/all /user/v1.0/keys/ids /user/v1.0/keys/sample /user/v1.0/keys/{id}	POST DELETE GET GET DELETE GET	This controller is responsible for handling user public keys. These keys can be used by the provisoner to allow the user to login to the VMs from the machine the keys correspond to.
PlannerController	/user/v1.0/planner/all /user/v1.0/planner/ids /user/v1.0/planner/plan/ /user/v1.0/planner/vereify_plan /user/v1.0/planner/{id} /user/v1.0/planner/plan/{tosca_id} /user/v1.0/planner/post/{name} /user/v1.0/planner/tosca/{id} /user/v1.0/planner/post/{level}/{name}/{id}	DELETE GET POST POST DELETE GET GET POST GET POST	This controller is responsible for planing the type of resources to be provisopned based on a TOSCA description.
PlannerController0	/user/v0.0/switch/plan/planning	POST	This controller is responsible for planing the type of resources to be provisopned based on a TOSCA description.
ProvisionController	/user/v1.0/provisioner/all /user/v1.0/provisioner/ids /user/v1.0/provisioner/provision /user/v1.0/provisioner/sample /user/v1.0/provisioner/{id}	DELETE GET POST GET DELETE GET	This controller is responsible for obtaining resources from cloud providers based the plan generated by the planner
ProvisionController0	/user/v0.0/switch/provision/execute /user/v0.0/switch/provision/upload	POST POST	This controller is responsible for obtaining resources from cloud providers based the

			plan generated by the planner and uploaded by the user
ScriptController	/user/v1.0/script/ /user/v1.0/script/all /user/v1.0/script/ids /user/v1.0/script/sample /user/v1.0/script/upload /user/v1.0/script/{id}	POST DELETE GET GET POST DELETE GET	This controller is responsible for handling user scripts. These user can be used by the provisoner to run on the created VMs.
ToscaController	/user/v1.0/tosca/all /user/v1.0/tosca/ids /user/v1.0/tosca/post /user/v1.0/tosca/upload /user/v1.0/tosca/{id}	DELETE GET POST POST DELETE GET	This controller is responsible for storing TOSCA descriptions that can be used by the planner.
UserController	/manager/v1.0/user/all /manager/v1.0/user/ids /manager/v1.0/user/modify /manager/v1.0/user/register /manager/v1.0/user/{id}	GET GET POST POST DELETE GET	This controller is responsible for handling user accounts
UserController0	/manager/v0.0/switch/account/register	POST	This controller is responsible for handling user accounts
UserPublicKeysController0	/user/v0.0/switch/provision/confuserkey	POST	This controller is responsible for handling user public keys. These keys can be used by the provisoner to allow the user to login to the VMs from the machine the keys correspond to.
UserScriptController0	/user/v0.0/switch/provision/confscript	POST	This controller is responsible for handling user scripts. These user can be used by the provisoner to run on the created VMs.

B Data Types

type	description
AnsibleOutput	This class represents the the ansible out put for a specific VM. This can be used as a archive / log of ansible executions

AnsibleResult	This class represents an ansible execution result. This can be used as a archive / log of ansible executions for example how much time it took for execution, errors etc.
BenchmarkResult	This is the base class for users to own resources. Many classes extend this class
CloudCredentials	This class represents the cloud credentials. They are used by the provisoner to request for resources.
DeployParameter	This class is used by the deployer to deploy software (swarm,kubernetes,ansible). It is generated by the provisioner to contain VM information.
DeployRequest	This class holds the necessary POJO IDs to request the deployment of a software
DeployResponse	This class represents the response of a deploy request. It may hold a key pair used for logging in and managing a docker cluster. Currently they key pair is only used by kubernetes
Key	This class represents a key. This key can be used to either login to a VM created by the provisiner or by the VM to allow the user to login to the VMs from the machine the keys correspond to.
KeyPair	This class hold the pair of public private keys. The keys may be used for logging in VMs.
KeyType	This enu specifies if a key is private or public
KeyValueHolder	This is a generic class that hold key-value pairs. It's main usage is to hold abstract types such as TOSCA.
OwnedObject	This is the base class for users to own resources. Many classes extend this class
PlanRequest	This class represents a plan request sent to the planner.
ProvisionRequest	This class is a holder for the the object IDs that are required by the provisioner to request for cloud resources.
ProvisionResponse	This class represents a description of provisioned resources
Script	This class represents a simple script that can run on a provisioned VM.
User	This class represents a user.