Using Docker Swarm with a User-Centric Decision-Making Framework for Cloud Application Migration

Barlaskar, E., Kilpatrick, P., Spence, I., & Nikolopoulos, D. S. (2018). Using Docker Swarm with a User-Centric Decision-Making Framework for Cloud Application Migration. In Cloud Computing and Service Science - 7th International Conference, CLOSER 2017, Revised Selected Papers (1 ed., Vol. 864, pp. 81-101). (Communications in Computer and Information Science; Vol. 864). Springer International Publishing. https://doi.org/10.1007/978-3-319-94959-8_5

Published in:
Cloud Computing and Service Science - 7th International Conference, CLOSER 2017, Revised Selected Papers

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Using Docker Swarm with a user-centric decision-making framework for cloud application migration

Esha Barlaskar, Peter Kilpatrick, Ivor Spence, and Dimitrios S. Nikolopoulos

The School of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast,
BT7 1NN, Belfast, United Kingdom
{ebarlaskar01,p.kilpatrick,i.Spence,
d.nikolopoulos}@qub.ac.uk

Abstract. *Vendor lock-in is a major obstacle for cloud users in performing multi-cloud deployment or inter-cloud migration, due to the lack of standardization. Current research efforts tackling the inter-cloud migration problem are commonly technology-oriented with significant performance overheads. Moreover, current studies do not provide adequate support for decision making such as why and when inter-cloud migration should take place. We propose the architecture and the problem formulation of a Multi-objective dYnamic MIgratioN Decision makER (MyMinder) framework that assists cloud users in achieving a stable QoS performance in the post-deployment phase by helping decide on actions to be taken as well as providing support to achieve such actions. Additionally, we demonstrate the migration capability of MyMinder by proposing an Automated Triggering Algorithm (ATA), which uses existing Docker Swarm technology for application migration.*

Keywords: Cloud computing, Dynamic decision making, QoS monitoring, Inter-cloud migration, Docker Swarm

1 Introduction

With the expansion of the range of Cloud Infrastructure-as-a-Service (IaaS) providers, efficient and accurate cloud provider (CP) selection based on user-specific requirements has become a significant challenge for cloud IaaS users. Cloud users have to engage in a number of complex decision-making processes which mainly stem from performance variability amongst the CPs and also from diversified pricing policies offered by different CPs. The reason for such variability is the heterogeneity prevailing amongst the CPs. In addition to this initial challenge in CP selection, there exist further challenges after the deployment of user applications in the form of monitoring the health of the acquired virtual machines (VMs) to verify whether the applications are performing in a stable manner with minimum or acceptable variations. In the post-deployment phase

the major cause for performance variability is muti-tenancy problems which arise because most of the computing resources (network and disk I/O) except for CPU cores are shared amongst several users' instances (from here on we will use the terms VM and instance interchangeably) running in a physical server [15] [19] [18]. Such variations due to performance degradation can be a serious problem for latency-sensitive and I/O-bound applications. Therefore, accurate monitoring and detection methods are required. Although cloud service monitoring tools are provided by CPs and third party companies [30] [33] [4], these monitoring tools do not provide any decision support on what steps a cloud user should follow if he/she realises that even their minimum performance requirements are not met by the selected instances in the current CP.

To meet the desired performance requirements cloud users may require to migrate their applications to new instance type with higher configuration from the same provider or with a similar configuration from a different provider . Apart from performance, cost can also be an important factor for certain budget-constrained users who may be interested to migrate to different instances if the price for the current cloud service rises or other providers offer a better price. Taking decisions on whether to migrate applications for better performance/cost poses further decision making and technical challenges for cloud users.

Although some researchers have tried to address the vendor lock-in issues by designing inter-cloud migration techniques, they have not provided any decision-making support. Others have focussed mainly on pre-deployment decision-making and there has been very limited work on post-deployment phase support, and these latter do not consider realistic migration overheads in the evaluation of their decision making framework. Therefore, naive cloud users should have an efficient dynamic decision making framework, which can help to provide guidance on the following:

1. How to detect if the current provider is not performing as required by user's application?
2. How to decide that the user's application needs to be migrated from the current provider?
3. Which alternative CP should be chosen to migrate the VM?
4. What instance type(s) will provide the best trade-off between cost and performance?
5. Whether the migration overhead will be more significant compared to the performance degradation in the current CP?

We envisage a system which can handle inter-cloud migration automatically along with a decision making framework, thus delivering the best of both the worlds. In previous work [1] we introduced a Multi-objective dYnamic MIgratioN Decision makER (MyMinder) framework designed to address the above issues. MyMinder offers a catalogue of metrics based on performance, cost and type of resources, from which cloud users can choose their requirement metrics depending on their application. Also, while choosing these metrics users can set some internal performance requirements and their maximum budget. MyMinder takes these requirements as inputs to carry out the monitoring and computes user satisfaction values based on their applications' performance requirements. In the event of any QoS violation or performance degradation MyMinder supports the user in finding alternative cloud services which can provide near-optimal

performance, and efficiently migrate the application to that service provided by either the same or different CP. Our work in [1] presented the MyMinder architecture and problem formulation for selecting the most suitable CP to migrate the VM along with some initial experimental results that motivate the need for live VM migration from one CP to another. Although we presented the performance variability results from the bursting instance types of the selected public CPs, authors in [15] and [14] have experimentally proved that even dedicated instances show performance variability.

In this study we present MyMinder's migration module and demonstrate its use for performing user application migration across VMs within the same CP or across different CPs. We deploy user applications using Docker container technology [5]. Docker container is a lightweight virtualisation technology that relies on operating system virtualisation. Using operating system virtualisation, containers can be easily ported across multiple providers and can run smoothly on top of public cloud providers' virtual machines. These capabilities have made Docker container technology highly prevalent in the DevOps community [5]. All these features make containers a suitable lightweight option for cloud user applications that easily supports transferability/portability and interoperability across different CPs. Considering the benefits of Docker containers and Docker Swarm management facilities[8] (detailed discussion is in Section 2), the proposed MyMinder prototype adopts the widely accepted Docker Swarm technology in order to perform the multi-cloud deployment of user applications. However, Docker Swarm does not provide a facility for resource provisioning policies that are required by MyMinder. Therefore, we introduce an Automated Triggering Algorithm (ATA) that automates VM allocation and de-allocation in the Docker Swarm cluster and integrates the Swarm orchestration features as guided by the output generated by MyMinder's Decision-making process. We evaluate the performance of MyMinder's migration process by deploying it in an OpenStack testbed, where a cluster of Docker Swarm nodes are created using the VMs and application containers are transferred among these Swarm nodes. This evaluation is an attempt to verify the feasibility of MyMinder's migration process and does not include performance results from inter-cloud migration across public clouds.

The remainder of the paper is organised as follows. Section 2 presents background and related work in user centric live VM migration and decision making. Section 3 and Section 4 provide detail of the problem formulation and the MyMinder architecture, respectively. Migration using Docker Swarm and ATA is explained in Section 5. MyMinder migration module prototype set-up and performance evaluation of ATA are discussed in Section 6. Section 7 concludes the paper and discusses future work.

2 Background and Related Work

With the proliferation of CPs it has become very difficult for cloud users to select the one that best meets their needs. Once they select the perceived optimal cloud service from a CP, cloud users encounter further challenges as they need to verify whether their applications are performing in a stable manner with minimum or acceptable variations after being deployed in the CP's instances. If the user realises that their desired QoS requirements are not met by the selected instances then they may require to migrate

their applications to a new instance type from the same provider or to an instance with a similar configuration from a new provider. Apart from performance, cost can also be an important factor for certain budget-constrained users who may be interested in migrating to different instances if the price for the current cloud service rises or other providers offer better price. Taking decisions on whether to migrate applications for better QoS/cost poses further decision-making and technical challenges for the user. We discuss how current work in the literature addresses these challenges in the following sections.

2.1 Post-deployment decision making

Although researchers have proposed different decision making methods in the pre-deployment phase [16], [3], [11], [32], [26], [24] decision making in the post-deployment phase has not received much attention, other than the works in [25] and [17].

The authors in [25] address decision making in the post-deployment phase by proposing a multi-stage decision-making approach. In the first stage, the available CP instances are shortlisted on the basis of the user's minimum QoS and cost criteria, and in the second stage, migration cost and time are evaluated. After completing these stages, they use the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) [2] and ELimination Et Choix Traduisant la REalit (ELimination and Choice Expressing REality), commonly known as ELECTRE [28], to find the most appropriate migration suggestion. They demonstrate their approach using a case study example. However, in their evaluation, they consider the overhead of a manual migration process where they assume that the network throughput between the source and the destination hosts remains constant during the migration process, which is unlikely to be true in real scenarios.

In [17] a linear integer programming model for dynamic cloud scheduling via migration of VMs across multiple clouds is proposed in the context of a cloud brokerage system. The migration is triggered if a CP either offers a special discount or introduces a new instance type, and also if the user needs to increase the infrastructure capacity. They do not consider QoS violation or degradation in their migration decision. Moreover, they performed their experiments in a simulation based environment and the metrics that they considered for measuring migration overhead may not be feasible to obtain in real world scenarios.

2.2 User-centric inter-cloud migration

Although cloud users should not be worried about the complexities involved in VM migration - which is the essence of the 'cloud philosophy', experienced cloud users may wish to have the flexibility that migration brings in the form of inter-cloud migration. However, there are complexities in migrating VMs from one CP to another CP due to vendor lock-in issues. Vendor lock-in makes a cloud customer dependent on a specific CP due to inherent dependencies on underlying cloud infrastructures. This makes it very difficult for the customers to transfer their applications to another CP without substantial migration costs. These dependencies are often subject to CPs specific (non-standardized) service APIs. For users to avail of the benefits of application

migration independent of the CP's permission, recent studies proposed different inter-cloud migration techniques which use a second layer of hardware virtualisation called nested virtualisation [34], [12], [23]. Nested VMs are usually migrated by using an NFS-based solution or an iSCSI-based solution. In some cases such as that of [22] the focus is not on providing storage and network support for wide-area network (WAN) application but rather on providing an enclosed environment for distributed application development and debugging. In an NFS-based and iSCSI-based solution the WAN VM migration experiences increased latencies, low bandwidth, and high internet cost in accessing a shared disk image if the shared storage is located in a different data centre or region. To address this issue [31] proposed Supercloud using nested virtualisation with a geo-replicated image file storage that maintains the trade-off between performance and cost. They designed an image storage that tries to propagate only data which is frequently accessed and it proactively transmits data before migration is triggered. However, Supercloud have some performance overhead due to that fact that nested virtualization imposes additional performance overhead, I/O overhead and CPU scheduling delay and also they do not provide any decision-making framework.

Other state-of-the-art techniques which allow multi-cloud deployment are Docker [5] and Multibox containers [10]. Nowadays containers are widely used as an alternative solution to more traditional Virtual Machines (VMs) allowing the deployment of virtualised resources with comparatively limited performance impact. Unlike VMs which run a full OS on virtual hardware, containers provide operating system level virtualisation where the associated deployments are much smaller in size because the container-based applications share their underlying OS. Containers can easily package an application into a single file which makes the process of application delivery and orchestration very flexible for the developers. Docker offers an elastic container platform called Docker Swarm which integrates container hosts (also referred to as Docker nodes or Docker Engines) into one single and higher level cluster. The Docker SwarmKit performs the Docker Engine's cluster management and builds the orchestration features for the cloud user applications. These features include deployment, scale up/down, termination, and migration/transfer across Docker nodes. The author in [13] proposed a control loop which is able to scale and transfer elastic container platforms (i.e. Docker Swarm and Kubernates etc.) across different public and private cloud-service providers. However, this control loop is just one phase of a self-adaptive auto-scaling MAPE loops (monitoring, analysis, planning, execution) and does not include the monitoring, analysis and planning phases.

In an extensive discussion the author in [13] points out the four main benefits of using the elastic container platforms (like Docker Swarm, Google's Kubernetes, etc.), which are summarised below:

1. One logical cluster can be formed by integrating single container nodes (hosts), where the hosts are within a single CP in order to help in complexity management of the deployed application.
2. This logical cluster can be extended across different CPs.
3. Different CP container nodes can be accessed as one single cluster which will solve the vendor lock-in problem.

4. These elastic container platforms have self-healing capabilities as they are designed with failover mechanisms. Their auto-restart, auto-replication, and auto scaling features help in the event of node failure or any process failure.

Considering the benefits of lightweight virtualization and evaluating the complexities/performance overhead of the existing inter-cloud migration techniques like nested virtualisation[12], the proposed MyMinder prototype adopts the widely accepted Docker Swarm container technology [8] in order to perform the multi-cloud deployment of user applications. However, Docker Swarm does not provide the facility for resource provisioning policies that are required by MyMinder. Therefore, we introduce an Automated Triggering Algorithm (ATA) that automates Docker Swarm cluster management and orchestration features based on the output generated by MyMinder's Decision-making process.

3 Problem Formulation

As presented in [1] the MyMinder framework (Figure 1) can assist cloud users in achieving a stable QoS performance in the post-deployment phase by helping decide on actions to be taken as well as providing support to achieve such actions. MyMinder can monitor the performance of the deployed users' applications and provide the required measurements to determine the satisfaction level of the user's requirements described in their requests. In the event of QoS violation or degradation in the current CP's service, MyMinder can trigger a migration decision after identifying a suitable CP to which the overhead of migration and the chances of QoS violation are the least. For performing these actions MyMinder needs to evaluate the satisfaction values based on the QoS/performance requirements specified in the user's requests. In the following subsections we illustrate user requirements, details of the CP instance type model, and the related measures [1].

3.1 User Requirements

A user sends a request describing his/her resource requirements and QoS/performance requirements. This request is represented by a requirement vector : $r = [r_1, r_2, ..., r_j]$ where r_j specifies the $jth(j = 1, 2, ...J)$ requirement of the user that has to be satisfied by the selected CP and these requirements may include the following information criteria [1]: 1) Resource criteria: amount of resources required for running user's application (e.g. memory, storage, CPU etc.). 2)Budget constraint: prices of the instances should be within the cost limit of the user. 3)QoS/performance criteria: Quality of service or performance requirements of user's application that has to be fulfilled (e.g. desired and maximum execution time, response time, throughput etc.) 4)Migration overhead constraint: cost of migration and performance overhead of migration should be acceptable.

Here, criteria 1 and 2 will be evaluated before deploying the application and only if these criteria are met then the application will be deployed and after deploying the application criteria 3 will be measured using a satisfaction value. The criteria 4 depends on the type of inter-cloud migration technology being used. The details of migration overhead measurement is discussed in section 5

3.2 CP Instance Types Model

Instances of different CPs differ in performance depending on their characteristics such as VM instance size, hardware infrastructure, VM placement policies used for load balancing or power optimisation etc. Factors affecting the QoS obtained from a particular instance type of a CP are typically not known by the user and so the QoS data of a given CP are not available in advance. It is possible to measure the QoS parameters only after the instance is deployed and these measurements may be evaluated against the requirements specified in the user request by determining the runtime performances such as execution time of applications, instructions committed per second (IPS), throughput etc. These measurements constitute the evaluation of the extent to which the QoS/ performance requirements specified in the user's request r_j are satisfied. The satisfaction level of user requirement r_j is denoted by $s_j \in [0,1]$, where $s_j = 1$ if the requirement r_j is fully satisfied, otherwise $0 \leqslant s_j < 1$ [1].

If a user provides the requirement vector r_i along with the desired QoS requirement and acceptable maximum variability in the QoS, then standard deviation (*SD*) is used as a measure of QoS performance variability. The closer the *SD* is to 0, the greater is the uniformity of performance data to the desired value ($r_{Qd(r_j)}$) and greater is the satisfaction value. The closer the *SD* is to 1, the greater is the variability of performance data to the desired value and smaller is the satisfaction value. Hence, the satisfaction value is given as follows [1]:

$$s_j = 1 - SD \tag{1}$$

$$SD = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(Qa(r_j) - \overline{M(r_j)})^2} \tag{2}$$

$$M(r_j) = \frac{1}{N-1}\sum_{i=1}^{N}(Qa(r_j)) \tag{3}$$

where,
$Qa(r_j)$=Actual QoS value obtained after deploying the user's application (e.g. actual execution time, response time, etc.). These values are in normalised form.
$M(r_j)$= The arithmetic mean of $Qa(r_j)$.
$r_{Qd(r_j)}$= Desired QoS requirements of the user's applications (e.g. desired execution time, response time, etc.) for the QoS requirement r_j. This value is used as a standard value against which QoS variability is compared.
N= total number of measurements.

3.3 Utility Function

The utility function $f(r)$ for each user request r_j is a linear combination of the satisfaction value s_j and the associated weights w_j multiplied by an indicator function $\phi(r)$. The weight for each of the user requests indicates its importance to the user and the indicator function sets the satisfaction level to zero when the request is not satisfied. In the case of satisfied requests the value of the indicator function is selected such that:

$\phi(r) = (\sum_j w_j)^{-1}$ normalizes the weight vector and limit the maximum possible value of $f(r)$ to 1 [1].

Thus, the utility function is defined as [1]:

$$f(r) = \phi(r) \sum_{n=1}^{J} w_j s_j \tag{4}$$

where

$$\phi(r) = \begin{cases} 0, & \text{if } QoS \text{ not met}. \\ (\sum_j w_j)^{-1}, & \text{otherwise}. \end{cases} \tag{5}$$

If all the requirements of a user are fully satisfied then $f(r) = 1$; otherwise if the requirements are partially satisfied then the value of $f(r)$ will vary with the amount of requirements being satisfied by a particular instance type of a CP. To demonstrate this lets consider one simple example [1]:

Let $r = [r_1, r_2, ..., r_j]$ be the user's requirement vector while making his/her initial request. The request contains the user's requirements constraints and the type of the requirement attributes are presented below:

1) r_R: Requested amount of resources required for running the user's application (e.g. memory, storage, CPU etc.) where $r_R \in micro, small, medium, large, xlarge$.

2) r_B: Prices of the instances specified in the user's budget where $r_B \in Max_{price}$

3) r_{Qd}: Desired QoS requirements of the user's applications (e.g. desired execution time, response time, IPS, etc.) where $r_{Qd} \in D_{val}$.

4) r_{Mo}: Maximum migration overhead a user can accept where $r_M \in Overhead\ of\ migration$.

The value of the satisfaction vector is calculated with the help of monitoring and detection modules (see Section 4) which is given by S_i^T (Equation 1). We assume that for a user's request with a requirement vector r_i=[micro, 200s, 400s, £5/hr, 30%], the satisfaction vector is calculated as [1]:

$$S^T = [1, 0, 1, 1] \tag{6}$$

For simplifying the example we did not consider partial satisfaction values, and here 0 denotes fully satisfied and 1 denotes not satisfied. Therefore the utility value is calculated as follows if the weight vector is $W^T = [0.1, 0.1, 0.1, 0.3]$ [1]:

$$f(r) = \begin{cases} 0, & \text{if } QoS \text{ not met}. \\ \phi(r) W^T S^T = 0.5, & \text{otherwise}. \end{cases} \tag{7}$$

The induction function's value is considered to be 1 in this case and also the utility function's value did not exceed 0.5 even though more than half of the requirements were fully satisfied.

These utility values will be used to predict the QoS for each CP's instance types model [1].

4 MyMinder Architecture

In this section we describe the architecture of MyMinder [1] that will implement the problem formulation. Figure 1 [1] depicts the MyMinder architecture, which includes modules for: monitoring, detection, prediction, decision making and migration. We describe each of these modules in the following subsections:

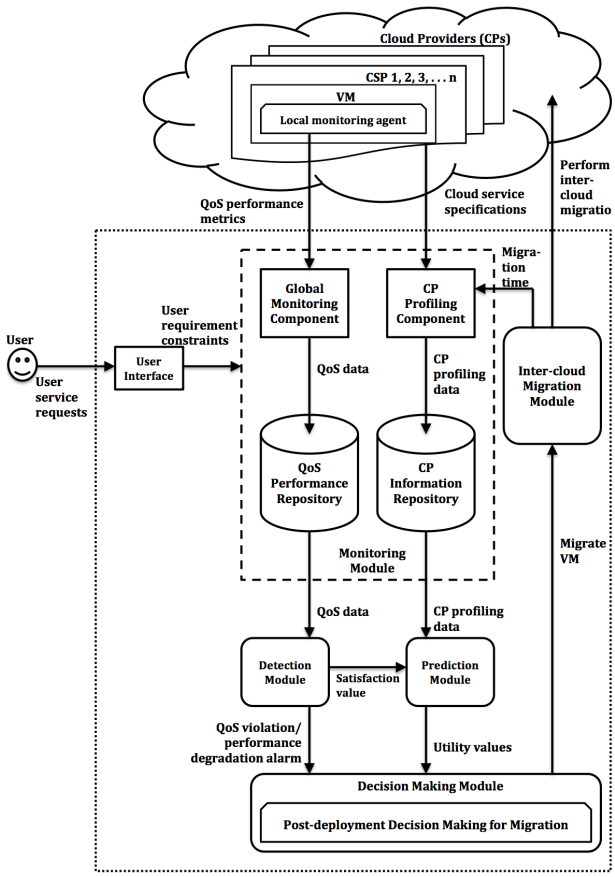

Fig. 1: MyMinder Architecture

4.1 Monitoring Module

The monitoring module is designed for monitoring the QoS performance of the user's application containers deployed in the VM. The performance data are collected by local monitoring agents deployed in each user's VM. The local monitoring agents send

the collected data periodically to the global monitoring component in the monitoring module and then finally the data are stored in the QoS performance repository. Also, the monitoring module maintains another repository, which stores information regarding the list of available VMs from different CPs and their prices. This information is collected by CP profiling components.

4.2 Detection Module

The detection module is responsible for detecting any QoS violation or degradation in the performance. The performance data are retrieved from the QoS performance repository. It uses a window-based violation detection technique [20] to generate QoS violation or performance degradation alarms based on the user's QoS requirement constraints and the user can decide the size of the window. This module generates QoS violation alarms if the current performance value falls outside the acceptable range as defined by the QoS statement. It also can be tuned to generate a degradation alarm if the performance moves to and stays within a defined distance of the QoS limits throughout a defined period. Degradation alarms may be used to predict likely breach of QoS and so may contribute to preventative migration. The module reports QoS violation and degradation alarms on a continuous basis by sending them to the decision making module.

4.3 Prediction Module

The objective of the prediction module is to help identify a suitable CP instance to which the user's application may be migrated. Based on the user's QoS satisfaction values (measured by the detection module) and the user's requirements, the prediction module calculates the utility function (see Equation 4) for each of the available CP instances. The satisfaction values for the current as well as previously deployed CP instances by the same user or different users are stored with their corresponding utility values. These perceived utility values are used to train the prediction models for each of the CP instances using machine learning techniques. Thus, the prediction models are capable of predicting the QoS satisfaction values in the destination CP for the new user's instance which needs migration. Further, the prediction module predicts application migration/transfer time based on the analysis of the historical migration time data (stored in the monitoring module) factoring in similar type of applications and also the pair of CPs involved in the migration. The measurements of the migration time are previously obtained by ATA (see Section 5) in the migration module.

4.4 Decision-making Module

The decision making module receives alarms from the detection module if any QoS violation or degradation is detected, and also it takes utility function values as input from the prediction module. It then checks with user requirement constraints to know whether the user wants to be informed before reaching the minimum requirement levels, i.e. performance degradation alert or to be informed if the minimum requirements are not met, i.e. QoS violation alert. After confirming user requirements, this module

verifies whether the instances with different utility values provided by the prediction module are currently available for selection. If the instances are available then it evaluates the migration overhead of each of the instances and finally ranks the instances based on their utility value and migration overhead values. The instance with highest utility value and lowest migration overhead is chosen for migration. The migration overhead will depend on the type of inter-cloud migration technique being used. The migration overhead can be defined either in terms of monetary loss or performance loss and it usually denotes the service downtime penalty per time unit.

4.5 Migration Module

The migration module (Figure 2) takes the decision generated by the decision-making module as its input for transferring/migrating user applications to the selected VM of the same CP or different CP. Once the migration module completes the task of application transfer/migration, it sends the migration overhead measurements to the monitoring module, which stores them as historical data for later use by the predication module. We adopt Docker containers to deploy user applications in the VMs of the selected CPs and Docker Swarm technology [8] to enable the transferability and portability features of Docker containers. We introduce an Automated Triggering Algorithm (ATA) that takes the output generated by the Decision-making module and makes use of the Docker Swarm cluster management and orchestration features (e.g. auto-placement, auto-restart, auto-replication and auto-scaling) in order to meet MyMinder's migration requirements. For example, if the decision making module takes a decision to migrate the user application from CP1 to CP2, ATA firstly adds the selected VMs of CP2 to the Swarm cluster and secondly, ATA de-allocates the VMs of CP1. The Swarm cluster immediately identifies this de-allocation of VMs as node failures and then using its self-healing mechanism it auto-restarts the lost application containers in CP2. Although this action is just a rescheduling mechanism of the Docker Swarm platform due to the node failures, this appears to be a migration from CP1 to CP2 at real-time from a cloud user perspective. The following section explains the transferability/migration operations in details.

5 Migration using Docker Swarm

As stated earlier, by proposing MyMinder we do not aim to design a new inter-cloud migration technique, but rather we aim to design a framework for cloud users which can take correct decisions on when and where to migrate their applications in case of QoS violations and degradations. In order to demonstrate the capability of MyMinder in performing migrations, we prototype the migration using the existing Docker Swarm technology in a lab-based OpenStack cloud.

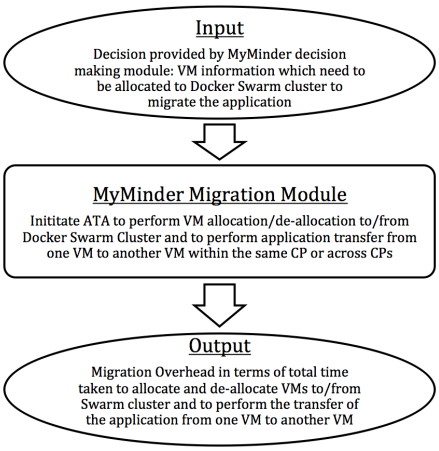

Fig. 2: MyMinder Migration Module

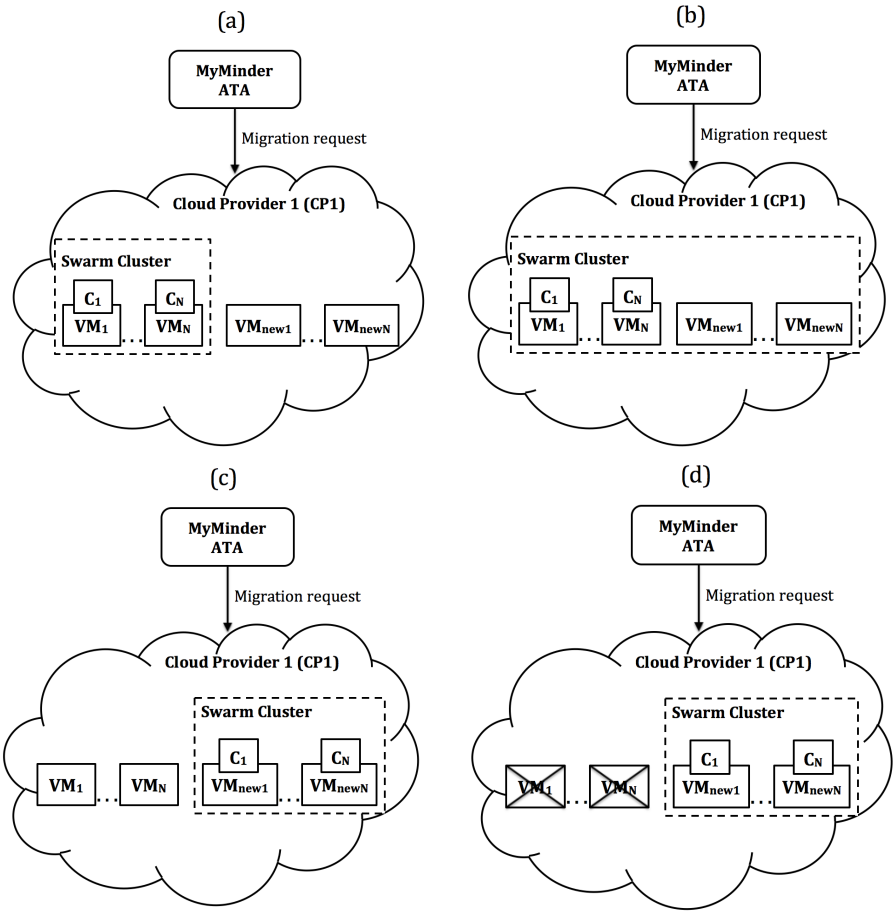

Fig. 3: MyMinder migration scenario 1 (migration within a single cloud provider (CP1)): (a) start new VMs, (b) allocate new VMs to Swarm Cluster, (c) de-allocate old VMs from Swarm cluster, (d) terminate the old VMs

In addition to using the Docker Swarm orchestration features we need to perform allocation and de-allocation of resources (VMs) to/from the Docker Swarm cluster in order to complete the migration operation. Specifically, we need to allocate new VMs (to which the applications need to be migrated) from the same CP or a different CP in the Swarm cluster and de-allocate current VMs (which failed to meet users's satisfaction) from the Swarm cluster. Figure 3 shows the migration scenario within a single CP, whereas Figure 4 depicts inter-cloud migration scenario. However, adding VMs to a Swarm cluster from multiple CPs is challenging due to the different approaches followed by different CPs in providing access to their virtual resources. As reported in [13] the public CPs organize their IaaS by using mainly two approaches which are identified as project-based and region-based service deliveries. Any multi-cloud deployment must consider the occurrence of both approaches in parallel. Therefore, we introduce an Automated Triggering Algorithm (ATA) to merge different approaches of different CPs and allocate/de-allocate VMs to Swarm cluster. To connect the Swarm cluster with a specific public CP it is required to have a configuration file for storing details used to communicate with the CPs. This can include authentication credentials and driver-specific configuration options. In this paper, the focus is on multiple VMs from a single private OpenStack cloud as depicted in figure 3, so the discussions on authentication credentials and driver-specific configurations are not included. Deployment in multiple public CPs will be considered in our future work once the decision-making module is functional; readers interested in details of multi-cloud deployment can refer to [13]. We describe MyMinder's migration operation using ATA in the following subsection.

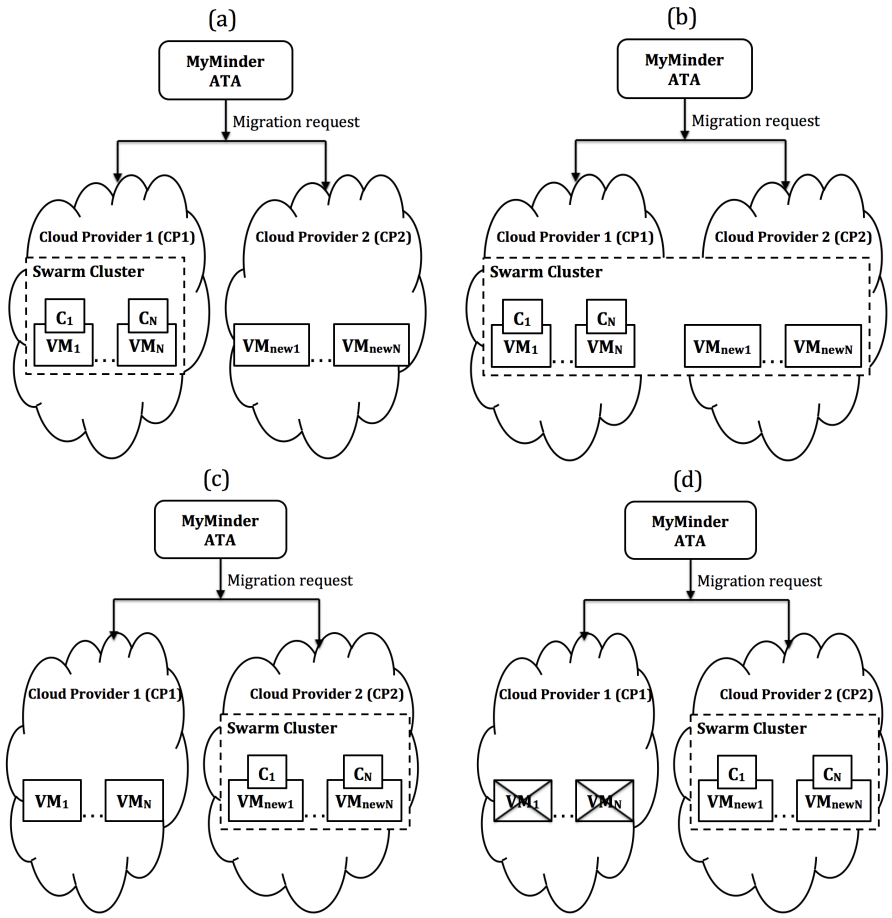

Fig. 4: MyMinder migration scenario 2 (migration across cloud providers (CP1 to CP2)): (a) start new VMs in CP2, (b) allocate new VMs to Swarm Cluster, (c) de-allocate old VMs from Swarm cluster, (d) terminate the old VMs from existing cloud provider (CP1)

5.1 Automated Triggering Algorithm (ATA)

This section describes MyMinder's migration operation using ATA, which includes four stages. We present the pseudocode of the algorithm in Algorithm 1 and explain the stages as follows.

Algorithm 1

Automated Triggering Algorithm

input:
VM_{new} : list of N new VMs (name and flavour of the VMs) to be allocated to Swarm cluster
VM_{old} : list of N old VMs (name and flavour of the VMs) to be de-allocated from Swarm cluster
CP_{sel} : selected CP where the new VMs need to be allocated
$Cred$: credential file that includes the authentication credentials for the new VMs
$Driver$: IaaS driver for the selected CP

output: $M_{overhead}$ - total application migration overhead, which is calculated as the summation of VM allocation, VM de-allocation, and application transfer time.

explanation: *token = Swarm cluster joining token, one for master node and one for worker node*,

1: **for each** $VM_{new\,i}$ **where** i=1,...,N **do**
2: ssh to $VM_{new\,i}$
3: *start_VM($VM_{new\,i}$, Cred, Driver)* in the CP_{sel}
4: **end for**
5: allocation_start_time = *current_time()*
6: **for each** $VM_{new\,i}$ **where** i=1,...,N **do**
7: **if** ($VM_{new\,i}$ is INITIAL_VM) **then**
8: *install_docker_engine()*
9: token[master/worker] = *swarm_init()*
10: **else**
11: *allocate_VM(token[master/worker])*
12: **end if**
13: **end for**
14: allocation_end_time = *current_time()*
15: allocation_time = allocation_end_time - allocation_start_time
16: de-allocation_start_time = *current_time()*
17: **for each** $VM_{old\,i}$ **where** i=1,...,N **do**
18: *de-allocate_VM(SwarmLeave[master/worker])*
19: **end for**
20: de-allocation_end_time = *current_time()*
21: de-allocation_time = de-allocation_end_time - de-allocation_start_time
22: transfer_start_time = *current_time()*
23: *transfer_application (from VM_{old} to VM_{new})* ▷ this is performed by Swarm cluster's auto-restart feature
24: transfer_end_time = *current_time()*
25: transfer_time = transfer_end_time - transfer_start_time
26: **for each** $VM_{old\,i}$ **where** i=1,...,N **do**
27: *terminate_VM($VM_{old\,i}$, Cred, Driver)* in the $CP_{existing}$
28: **end for**
29: $M_{overhead}$ = allocation_time + de-allocation_time + transfer_time
30: **return** $M_{overhead}$

1. **Resoruce (VM) provisioning**: ATA runs a configuration file for provisioning new VMs (as decided by the decision making module) either from the same CP (as shown in figure 3(a)) or from a different CP (as shown in figure 4(a)). This config-

uration file includes the detailed information about the VMs (e.g. flavor and name) along with the authentication credentials and IaaS drivers. In this stage all the requested VMs are started *(pseudocode lines 1-4)*. This installation is done by running SSH-based scripts. ATA starts the second stage only when all the requested VMs are running successfully and all the security groups are ready.

2. **VM allocation to Swarm cluster**: ATA installs Docker Engine in the VMs *(pseudocode line 8)* and allocates these VMs to the Swarm cluster (as shown in figures 3(b) and 4(b)) *(pseudocode line 11)*. During the allocation process their roles (master/ worker) are defined by calling the CP specific platform driver [6]. In Docker Swarm platform nodes acts as either masters or workers. The master performs all scheduling tasks (auto-restart, auto-scale etc.) and the workers run the application containers. Therefore, the VMs which are master nodes are added first and then the worker nodes. If any of joining fails then ATA again runs the joining procedure until all the requested nodes are successfully added to the Swarm cluster.
3. **VM de-allocation from Swarm cluster**: To reach the desired state (as given by the output of the decision-making module) the user application containers need to be rescheduled to the newly added Swarm nodes (new VMs). This is achieved by de-allocating the current Swarm nodes (old VMs) (as shown in figures 3(c) and 4(c)) *(pseudocode lines 17-19)* which in turn triggers the Swarm master to auto-restart the application containers in the available nodes (new VMs). Thus, in this stage all the old VMs are de-allocated from the Swarm cluster. After the de-allocation procedure, the Swarm scheduler recognises this as node failure and then using its auto-rescheduling features it automatically transfers all the application containers to the newly added Swarm nodes (new VMs) *(pseudocode line 23)*.
4. **Resource (VM) termination**: After the containers are auto-restarted in the new Swarm nodes, ATA terminates the old VMs (as shown in figures 3(d) and 4(d)) and also deletes the old security groups *(pseudocode lines 26-28)*. Finally, the total application migration overhead is calculated by adding the time required for VM allocation, VM de-allocation, and application transfer *(pseudocode line 29)*.

6 Experimental Evaluation

In this section we evaluate the performance of MyMinder's migration operation while migrating user applications from one VM to another VM in a lab-based OpenStack cloud. As the decision making module is not yet fully functional, we trigger the migration module manually by requesting migration of a user application from one VM to another VM.

6.1 Experimental Set-up

We built a multi-node Swarm cluster in an OpenStack cloud test-bed which consists of four compute nodes. All the compute nodes are Dell PowerEdge R420 servers which run CentOS 6.6 and have 6 cores, 2-way hyper-threaded, clocked at 2.20 GHz with 12GB DRAM clocked at 1600 MHz. The nodes include two 7.2K RPM hard drives with 1TB of SATA in RAID 0 and a single 1GBE port. KVM is the default hypervisor of the compute nodes.

To measure the application transfer time we run a simple voting application [29] from Docker [7] that is representative of real world microservice cloud applications. The application has several microservices. The voting application is composed of: (i) Python web app (vote-app) which allows users to vote between two options (cats or dogs), (ii) Redis queue which collects new votes, (iii) .NET worker which consumes votes and stores them in a database, (iv) Postgres database backed by a Docker volume (volume is created in the Swarm manager node), and (v) Node.js webapp (results-app) which shows the results of the voting. The services are deployed in the Swarm with certain constraints. The Python web app and the redis are deployed with two replicas and with a restart policy which restarts the containers on node failures. The Node.js webapp is also deployed with node failure restart policy and with one replica. But the Postgres database and the .NET worker are deployed with a placement constraint which starts them on the Swarm manager node only and without any restart policy. The database is stored in the host machine (Swarm manager node) which provides data persistence for the application. Therefore, in our experiments the Swarm manager node is not de-allocated (also referred to as drained) as the Docker volume is attached to this VM. We have put this placement constraint because if we deploy the Postgres container in one of the worker nodes which is drained later then losing the data of the Postgres container would cause the application to fail.

However, this approach does not allow migration of application containers with attached databases: to perform such migrations Docker Swarm requires additional storage plugins. Open-source container data volume orchestrators such as Flocker [9], Portworx [21] and REX-RAY [27] can be used for migrating stateful Dockerized applications. Unlike a Docker data volume which is tied to a single server, the data volume provided by these storage drivers is portable and can be used with any container in the Swarm cluster. Flocker can only be used within a single data centre whereas, Portworx and REX-RAY can migrate data across CPs. In our future work we will consider stateful application (databases) migration across CPs by using the storage drivers such as Portworx or REX-RAY.

6.2 Experimental Results and Discussion

MyMinder's migration operation is performed by Docker Swarm and with the help of ATA allocation and de-allocation of VMs (Swarm nodes) to/from Docker Swarm cluster. We examine the migration performance by measuring the allocation and de-allocation time of the Swarm nodes and the application transfer time taken by Docker Swarm node manager (scheduler). In order to collect these measurements we initially set the Swarm cluster with four VMs allocated as the Swarm nodes where one of the VMs acts as the Swarm master node and rest of the three VMs act as Swarm worker nodes. All four VMs are 'medium' flavour instances from OpenStack. Later we add three new 'large' flavour VMs (as Swarm worker nodes) to the Swarm cluster and remove the three 'medium' flavour VMs (Swarm worker nodes) one by one. The list of the VMs for addition and deletion are stored in a configuration file which is sent to the ATA to trigger the Swarm cluster's node allocation and de-allocation steps.

In the Figure 5, we present the time taken to allocate new VMs (the new "large" flavour VMs as Swarm worker nodes) to the existing Swarm cluster, the time taken to

de-allocate the existing VMs (the ”medium” flavour VMs running as Swarm worker nodes) from the Swarm cluster, and the application transfer time from one VM to another VM which is performed by the Docker Swarm. We present the averaged values of the observed measurements, where the allocation, de-allocation, and transfer were performed 20 times. The allocation and de-allocation time are almost the same every time but the the transfer time showed some variation in a 10 seconds range when we repeated the transfer procedure. As shown in Figure 5 the time taken to allocate one Swarm worker node is less than 1 second and to allocate all the three worker nodes together is between 1 to 3 seconds. The de-allocation time is between 1 to 2 seconds to remove 1 worker node and between 3 to 6 seconds to remove all three worker nodes one by one. De-allocation takes longer than the allocation time because the nodes are removed sequentially to avoid over stressing the Swarm master in rescheduling the application containers, whereas allocation is done in parallel as the Swarm master does not assign any existing container on the newly allocated nodes until any new application is deployed or any existing node is failed [13]. Importantly, we observe that once the ‘medium’ flavour VMs are de-allocated from the Swarm cluster, the application containers running on those VMs are rescheduled to the new VMs (the ‘large’ flavour VMs). The application reschedule/transfer time is around 20 seconds for a single node and it is around 50 seconds for the three nodes, which is shown in the Figure 5. The reschedule/transfer is performed using Docker Swarm’s auto-restart feature. Since the de-allocation is performed sequentially, the Swarm scheduler performs the rescheduling of the application containers in the same order in which the their hosted nodes are de-allocated. We observe that during the transfer period when the Swarm node with the results-app (Node.js) containers is drained we are not able to browse the results of the poll until the container is rescheduled and restarted.

If we add up the time taken for allocation, de-allocation, and application reschedule/transfer, we get the overall migration time as observed in the Figure 5, which is around 23 seconds if the migration requires a single node allocation/de-allocation and around 59 seconds if the migration requires multi-node allocation/de-allocation. These migration overhead results give us an understanding of the effectiveness of the proposed ATA in migrating applications across VMs in an OpenStack cloud environment and we do not intend it to be compared with the inter-cloud migration performance across public clouds. We performed the migration in a private cloud environment in order to build the proof of concept of the migration functionality of MyMinder. In future, we will carry out further experiments on evaluating ATA in migrating cloud application across public CPs in order to build the proof of concept of the framework as a whole.

7 Conclusion and Future Work

In this paper we present the architecture of MyMinder [1], a post-deployment decision making framework, which can detect QoS violation and performance degradation and dynamically decide whether a user’s VM requires migration from the current provider to another provider. Also, we present the problem formulation [1] for selecting the most suitable CP in the case that the VM requires migration from the current provider. As an extension, we present MyMinder’s migration module and demonstrate its feasibility in

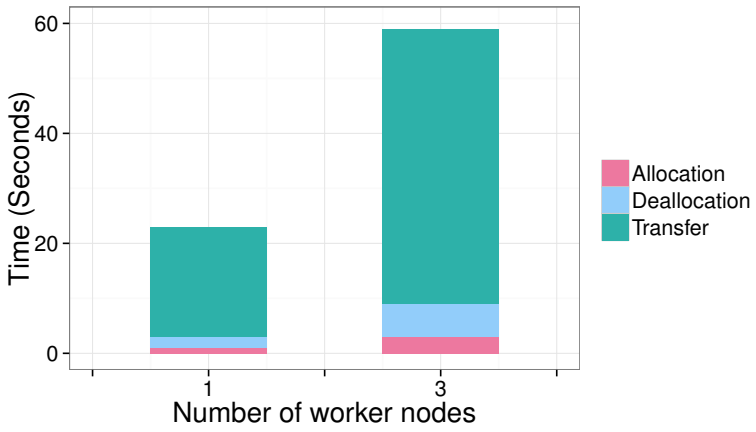

Fig. 5: Total application migration time as summation of VM allocation, de-allocation, and transfer time

performing user application migration across VMs from either the same CP or different CP. The MyMinder migration prototype adopts the widely accepted Docker Swarm technology. To merge and automate the migration steps, we propose an Automated Triggering Algorithm (ATA) that performs VM allocation and de-allocation to/from the Docker Swarm cluster in addition to the core Docker Swarm auto-rescheduling feature. We evaluate the performance of MyMinder's migration process by deploying it in a lab-based OpenStack testbed, where a cluster of Docker Swarm nodes is created using the VMs and application containers are transferred amongst the Swarm nodes. The experimental evaluation demonstrates that we can migrate user applications from one VM to another VM within a single CP without depending on the CP and with minimum migration overhead. This evaluation is an attempt to verify the feasibility of MyMinder's migration process and does not include performance results from inter-cloud migration across public clouds. In future, we consider to evaluate the performance of the proposed ATA in migrating applications across public CPs once the monitoring, detection, prediction, and decision-making modules are fully functional.

References

1. Barlaskar, E., Kilpatrick, P., Spence, I., Nikolopoulos, D.S.: Myminder: A user-centric decision making framework for intercloud migration. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science. pp. 588–595 (2017)
2. Behzadian, M., Khanmohammadi Otaghsara, S., Yazdani, M., Ignatius, J.: Review: A state-of the-art survey of topsis applications. Expert Syst. Appl. 39(17), 13051–13069 (Dec 2012), http://dx.doi.org/10.1016/j.eswa.2012.05.056
3. Brock, M., Goscinski, A.: Toward ease of discovery, selection and use of clusters within a cloud. In: 2010 IEEE 3rd International Conference on Cloud Computing. pp. 289–296 (July 2010)

4. Ciuffoletti, A.: Application level interface for a cloud monitoring service. Computer Standards and Interfaces 46, 15 – 22 (2016), http://www.sciencedirect.com/science/article/pii/S0920548916000027
5. Docker: Docker Containers. https://www.docker.com/ (2013), [Online; accessed 25-October-2016]
6. Docker: Docker Machine Drivers . https://docs.docker.com/machine/drivers/ (2017), [Online; accessed 1-August-2017]
7. Docker: Try Swarm at scale . https://docs.docker.com/swarm/swarm_at_scale/about/ (2017), [Online; accessed 2-August-2017]
8. Docker, S.: Docker Swarm . https://docs.docker.com/engine/swarm/ (2017), [Online; accessed 29-May-2017]
9. Flocker: FLOCKER. https://clusterhq.com/flocker/introduction/ (2016), [Online; accessed 29-December-2016]
10. Hadley, J., Elkhatib, Y., Blair, G., Roedig, U.: MultiBox: Lightweight Containers for Vendor-Independent Multi-cloud Deployments, pp. 79–90. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-25043-4_8
11. Han, S.M., Hassan, M.M., Yoon, C.W., Huh, E.N.: Efficient service recommendation system for cloud computing market. In: Proceedings of the 2Nd International Conference on Interaction Sciences: Information Technology, Culture and Human. pp. 839–845. ICIS '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655925.1656078
12. Jia, Q., Shen, Z., Song, W., van Renesse, R., Weatherspoon, H.: Supercloud: Opportunities and challenges. SIGOPS Oper. Syst. Rev. 49(1), 137–141 (Jan 2015), http://doi.acm.org/10.1145/2723872.2723892
13. Kratzke, N.: Smuggling multi-cloud support into cloud-native applications using elastic container platforms. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,. pp. 57–70. INSTICC, SciTePress (2017)
14. Kratzke, N., Quint, P.C.: About automatic benchmarking of iaas cloud service providers for a world of container clusters. Journal of Cloud Computing Research 1(1), 16–34 (2015)
15. Leitner, P., Cito, J.: Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. ACM Trans. Internet Technol. 16(3), 15:1–15:23 (Apr 2016), http://doi.acm.org/10.1145/2885497
16. Li, A., Yang, X., Kandula, S., Zhang, M.: Cloudcmp: Comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement. pp. 1–14. IMC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1879141.1879143
17. Li, W., Tordsson, J., Elmroth, E.: Modeling for dynamic cloud scheduling via migration of virtual machines. In: Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science. pp. 163–171. CLOUDCOM '11, IEEE Computer Society, Washington, DC, USA (2011), https://doi.org/10.1109/CloudCom.2011.31
18. Li, Z., OBrien, L., Zhang, H.: Ceem: A practical methodology for cloud services evaluation. In: 2013 IEEE Ninth World Congress on Services. pp. 44–51 (June 2013)
19. Li, Z., O'Brien, L., Zhang, H., Cai, R.: On a catalogue of metrics for evaluating commercial cloud services. In: 2012 ACM/IEEE 13th International Conference on Grid Computing. pp. 164–173 (Sept 2012)
20. Meng, S., Liu, L.: Enhanced monitoring-as-a-service for effective cloud management. IEEE Transactions on Computers 62(9), 1705–1720 (Sept 2013)
21. PORTWORX: The Solution for Stateful Containers in Production. Designed for DevOps. https://portworx.com/ (2017), [Online; accessed 16-August-2017]
22. Ravello: Ravello Systems: Virtual Labs Using Nested Virtualization. https://www.ravellosystems.com (2016), [Online; accessed 15 November-2016]

23. Razavi, K., Ion, A., Tato, G., Jeong, K., Figueiredo, R., Pierre, G., Kielmann, T.: Kangaroo: A tenant-centric software-defined cloud infrastructure. In: Cloud Engineering (IC2E), 2015 IEEE International Conference on. pp. 106–115 (March 2015)
24. u. Rehman, Z., Hussain, O.K., Hussain, F.K.: Multi-criteria iaas service selection based on qos history. In: 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA). pp. 1129–1135 (March 2013)
25. ur Rehman, Z., Hussain, O.K., Chang, E., Dillon, T.: Decision-making framework for user-based inter-cloud service migration. Electronic Commerce Research and Applications 14(6), 523 – 531 (2015), http://www.sciencedirect.com/science/article/pii/S1567422315000575
26. Rehman, Z.U., Hussain, O.K., Hussain, F.K.: Parallel cloud service selection and ranking based on qos history. Int. J. Parallel Program. 42(5), 820–852 (Oct 2014), http://dx.doi.org/10.1007/s10766-013-0276-3
27. REX-RAY: Rex-ray container storage management . http://rexray.readthedocs.io/en/stable/ (2017), [Online; accessed 16-August-2017]
28. Roy, B.: The outranking approach and the foundations of electre methods. Theory and Decision 31(1), 49–73 (1991), http://dx.doi.org/10.1007/BF00134132
29. samples, D.: Voting Application . https://github.com/dockersamples/example-voting-app (2017), [Online; accessed 12-August-2017]
30. Scheuner, J., Leitner, P., Cito, J., Gall, H.C.: Cloud workbench - infrastructure-as-code based cloud benchmarking. CoRR abs/1408.4565 (2014), http://arxiv.org/abs/1408.4565
31. Shen, Z., Jia, Q., Sela, G.E., Rainero, B., Song, W., van Renesse, R., Weatherspoon, H.: Follow the sun through the clouds: Application migration for geographically shifting workloads. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 141–154. SoCC '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2987550.2987561
32. Silas, S., Rajsingh, E.B., Ezra, K.: Efficient service selection middleware using electre methodology for cloud environments. Information Technology Journal 11(7), 868 (2012)
33. Silva-Lepe, I., Subramanian, R., Rouvellou, I., Mikalsen, T., Diament, J., Iyengar, A.: SOAlive Service Catalog: A Simplified Approach to Describing, Discovering and Composing Situational Enterprise Services, pp. 422–437. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-89652-4_32
34. Williams, D., Jamjoom, H., Weatherspoon, H.: The xen-blanket: Virtualize once, run everywhere. In: Proceedings of the 7th ACM European Conference on Computer Systems. pp. 113–126. EuroSys '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2168836.2168849