Deployment Strategies for Distributed Applications on Cloud Computing Infrastructures

Jan Sipke van der Veen[1,2], Elena Lazovik[1], Marc X. Makkes[1,2], Robert J. Meijer[1,2]
[1]*TNO, Groningen, The Netherlands*
[2]*University of Amsterdam, Amsterdam, The Netherlands*
{*jan_sipke.vanderveen, elena.lazovik, marc.makkes, robert.meijer*}*@tno.nl*

***Abstract*—Cloud computing enables on-demand access to a shared pool of IT resources. In the case of Infrastructure as a Service (IaaS), the cloud user typically acquires Virtual Machines (VMs) from the provider. It is up to the user to decide at what time and for how long they want to use these VMs. Because of the pay-per-use nature of most clouds, there is a strong incentive to use as few resources as possible and release them quickly when they are no longer needed. Every step of the deployment process, i.e., acquiring VMs, creating network links, and installing, configuring and starting software components on them, should therefore be as fast as possible. The amount of time the deployment process takes can be influenced by the user by performing some steps in parallel or using timing knowledge of previous deployments. This paper presents four different strategies for deploying applications on cloud computing infrastructures. Performance measurements of application deployments on three public IaaS clouds are used to show the speed differences between these strategies.**

***Keywords*-Cloud Computing; Infrastructure as a Service; Distributed Applications; Deployment; Provisioning; Performance.**

I. Introduction

Cloud computing enables on-demand access to a shared pool of IT resources. The types of resources that can be accessed depend on the service model [1]. In the case of Infrastructure as a Service (IaaS), the cloud user typically acquires Virtual Machines (VMs) from the provider. The user chooses the size of every machine from a set of available VM types, and the operating system from a set of images. As soon as the VM has been set up by the provider, the user can tailor the machine to his or her needs by installing additional software packages on it and configuring and starting the software.

It is up to the users to decide at what time and for how long they want to use these VMs. Most cloud providers bill their customers for the actual amount of resources they consume, the so-called pay-as-you-go model. Users therefore have a strong tendency to use as few resources as possible and, when they are no longer needed, release them quickly. A common approach is to measure some metric(s), e.g., processor usage or request/response latency, and scale up and down in the number of utilized resources based on thresholds of this metric [2], [3].

Another approach tries to minimize the amount of time the deployment process takes. One solution is to deploy an application once and then save the resulting VM as an image [4]. When the application is needed in the future, its predefined image is started instead of a blank one. Although this saves time during deployment, using VM images incurs storage costs. In addition, if the application to be deployed is dynamic in nature, using images would require costly updates. This paper focuses on the case where blank VMs are used and acquiring the VMs, creating network links, and installing, configuring and starting software components on them, is made as fast as possible.

The amount of time the deployment process takes depends on several factors. Some of these factors can be influenced by users while others solely depend on external parties. For example, the amount of time it takes for a VM to be available, depends on the hardware and software of the cloud provider, and therefore cannot be influenced by the user. Also, the amount of time it takes for a specific software package to be installed on a VM depends on its size, its dependencies, the network connection and the availability of mirror sites. However, the user does have influence on timing and ordering of the different steps in the deployment process. Performing some steps in parallel or using knowledge about previous deployment timing may speed up the process.

This paper presents a novel approach for deploying distributed applications on cloud computing infrastructures. Four different strategies for deploying applications are presented, in increasing order of knowledge about the specific application being deployed. The fourth strategy is the most novel, using historical metrics to guide the deployment process. Performance measurements of deployments on three public IaaS clouds are used to show experimental speed differences between these strategies.

II. Related Work

Installation of new and updated packages on UNIX systems is often performed by package managers such as APT [5] and YUM [6]. These package managers operate in several phases, two of which are the most time consuming. In one phase the selected packages are downloaded from mirror sites, and in a later phase the packages are installed on the local machine. Both of these phases are performed in sequence. Downloading in sequence is done to avoid congestion and to maintain availability of the mirror site serving the package, and installing in sequence is done to avoid dependency and process management issues and throttling of the hard drive. There is currently an experimental extension for APT to download packages in parallel [7], and such a feature is also planned for YUM [8].

Configuration management tools such as CFEngine [9], Puppet [10] and SmartFrog [11], provide automated configuration and maintenance of large-scale computer systems. This enables system administrators to get their IT infrastructure from an initial state to a desired state, almost fully automated. Although the software installation and configuration of machines is performed in parallel where possible, the configuration management software does not take the historical deployment performance into account.

Several papers describe the deployment of applications on cloud computing infrastructures. They are mostly focused on placing applications on VMs in such a way that the runtime performance is optimized. There is a distinction between those that optimize once, i.e., during the deployment phase, and those that try to optimize constantly, i.e., during the lifetime of the VMs. The focus of these papers is on different aspects of performance, such as internal communication speed [12], latency between the application and its users [13] or processing speed of the VMs [14]. The mentioned papers do not deal with the speed of the deployment process itself. The strategies discussed in this article can be integrated with the described solutions.

One of the deployment strategies presented in this paper uses performance metrics determined from historical deployments. Others have also studied the performance of public cloud services, focusing on the startup time of VMs [15], tracking performance of several operations within a cloud environment [16], or on the prediction of application performance [17]. This paper adds the metrics of individual deployment steps, such as the installation, configuration and starting of software.

III. Deployment Strategies

A computer-readable description of an application is needed to be able to automate its deployment. Every application needs at least two parts: an infrastructure to run on and its software. The actual notation is not important for this paper, but if needed can be written using OASIS TOSCA [18] or a custom XML or JSON notation.

The description of a webserver is used as an example, see part (a) of Figure 1. The webserver needs a single node, i.e., a physical or virtual machine, to run on and it needs to be reachable from the internet on port 80. A description for the infrastructure of this machine would contain these demands and additional properties such as the cloud provider to utilize, operating system to be installed and the minimum disk size. The software part contains descriptions of the installation, configuration (files and templates) and starting of the software.

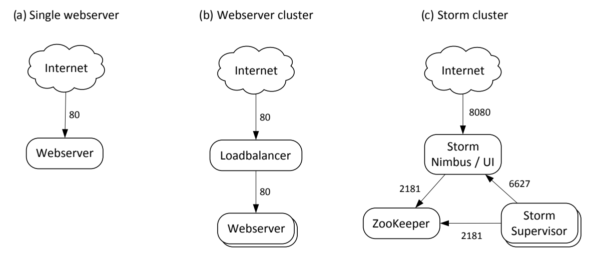

Figure 1. Nodes and links of a single webserver connected to the internet (a), a webserver cluster (b), and a Storm cluster (c). The labels on the edges are TCP port numbers.

The rest of this section discusses four strategies for deploying distributed applications on cloud computing infrastructures. The dependency graphs used in this section show how the steps in the deployment process depend on each other, see Figure 2.

Each vertex represents a step in the deployment process, e.g., creating a node or installing a piece of software. An edge from vertex A to B in the dependency graph denotes that step A depends on B. This means that step B should be performed first and needs to be finished before step A can start. Only vertices with no outgoing edges can therefore be performed immediately.

A. Serial (baseline)

The serial deployment strategy is the most common and simple of the four strategies presented here. Each step in the deployment process is added as a vertex to the graph. After that, an edge is created from step 2 to step 1, from step 3 to step 2, etc. In this manner, each step depends on the previous one and therefore the deployment process is serial. See graph (a) in Figure 2 for details.

B. Partially Parallel

The serial deployment strategy can be improved upon by performing some steps in parallel. The creation of nodes, for example, takes a long time, typically a couple of minutes. If all node creation steps are performed simultaneously, the total deployment time decreases dramatically. This can be done without additional checks, because the creation of one node does not depend on the creation of other nodes. The same applies to creating links, and installing, configuring and starting software. As soon as the current type of step is completed, all steps in the next type can be performed in parallel. See graph (b) in Figure 2 for details.

C. Fully Parallel

The partially parallel deployment in the previous section can be improved upon even further. For example, the software installation step can start as soon as its node is created.

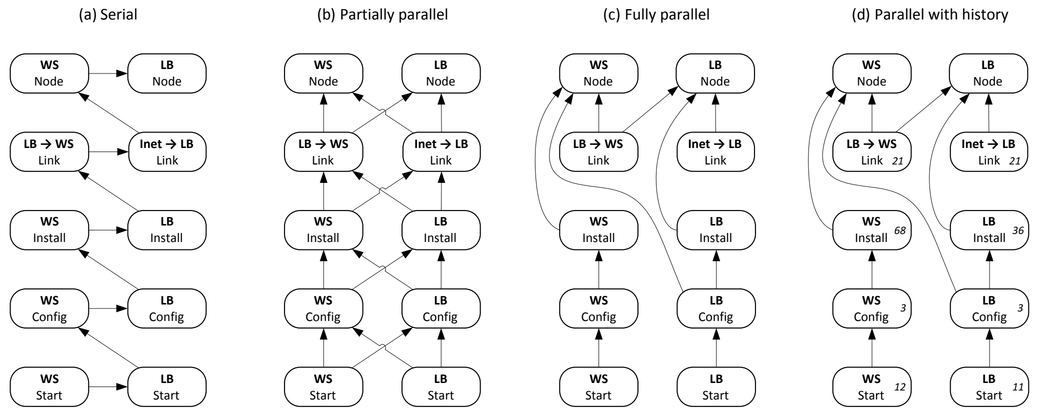

Figure 2. Dependency graphs of the webserver cluster. Each vertex represents a deployment step, and an edge from A to B means that step A depends on B. The label of a vertex shows how much time the deployment took on average for this step. WS = Webserver; LB = Loadbalancer; Inet = Internet.

There is no need to wait for all nodes to be created before this step can be performed. But there is a limit to which steps can be performed in parallel. For example, the configuration of one node may depend on the IP address of another node.

See graph (c) in Figure 2 for details of the fully parallel deployment of the webserver cluster. The following list provides an overview of the dependencies between all steps:

- **Nodes** The creation of a node does not depend on anything, i.e., the webserver node and the loadbalancer node can be created in parallel.
- **Links** The creation of a link depends on the nodes that it refers to, e.g., the link between the loadbalancer and the webserver depends on the creation of both nodes.
- **Installs** The installation of software depends on the node that it wants to install on, e.g., the installation of the webserver software depends on the webserver node.
- **Configs** The configuration of software depends on its installation, and possibly on references in the configuration files to other nodes, e.g., the configuration of the loadbalancer depends on both its installation and the creation of the webserver node.
- **Starts** The starting of software depends on its configuration, e.g., the loadbalancer software can be started as soon as its configuration is done.

D. Parallel with History

If we take historical deployments into account, we can improve upon the fully parallel deployment even further. For example, let us assume that historical deployments have proven that the installation, configuration and starting of software A takes longer than that of B. We also assume that both A and B require the same kind of node, i.e., the operating system and hardware specifications are the same. In that case, we can create both nodes in parallel and pick the first available node for software A. Dependency graph (d) in Figure 2 shows how the historical metrics are added as node labels. Each label represents the number of seconds the deployment step took on average in the past.

The added value of this strategy depends on several factors. A big variance in node creation times leads to a higher potential speed gain. Our experimental results show that there is indeed a substantial difference between node creation times, and this is supported by other experiments [16]. Also, a large difference between average software deployment times is beneficial, e.g., the installation of software A needs to take much more or much less time, on average, than that of software B. If this is the case, then it is useful to install the slowest-to-install software on the first available node.

The parallel with history strategy can only be used at the end of each node creation step. As soon as a node becomes specific, e.g., a link has been created or software has been installed, it cannot be (easily) swapped with another node to speed up the deployment.

IV. Algorithms for Dependency Graphs

An overview of the deployment system is given in Figure 3. Its primary component is the deployer which uses the deployment description as an input to execute the four strategies presented in Section III. There is also a small database present with information about historical deployments. This database is used to record the time taken by each deployment step, every time such a step is executed. All strategies store information in this database, but only the parallel with history strategy reads from it.

The implementation consists of two stages. At the first stage the dependency graph is built based on the deployment description and in the second stage the dependency graph is used for the actual deployment.

A. Creating the Dependency Graph

The creation of the dependency graph starts with the vertices. Each deployment step in the description becomes

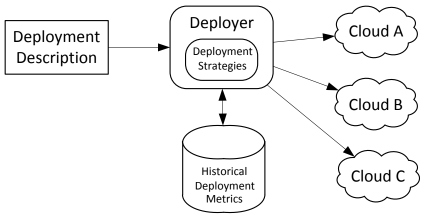

Figure 3. Deployer

```
procedure DEPLOY($graph, cloud$)
    while $graph.vertices \neq \varnothing$ do
        for $vertex$ in $graph.vertices$ do
            if $vertex.outgoing\_edges = \varnothing$ then
                $cloud.deploy(vertex)$
            if $vertex.finished()$ then
                REMOVE($vertex, graph$)

procedure REMOVE($vertex, graph$)
    $graph.remove(vertex)$
```

Figure 4. Algorithm for deploying a distributed application using the simple dependency graph

a vertex. For example, the software configuration step is a vertex and the link creation step is also a vertex. The vertices are the same for each strategy, because the steps to deploy an application are the same regardless of the order in which these steps are performed. In the case of the parallel with history strategy, each vertex is labeled with its historical average deployment time.

Each dependency in the description becomes an edge. One edge, for example, is the dependency that the software installation step can only be started when the node creation step has finished. Each edge is drawn as an arrow from a source to a target vertex. The deployment step in the target vertex must have finished before the step in the source vertex can be started. In contrast with the vertices, the edges of the graph do depend on the deployment strategy that is used, see Section III for details of each strategy.

B. Deploying with the Dependency Graph

The algorithm in Figure 4 shows how a dependency graph can be used for deploying applications. A while loop checks every vertex in the graph. If a vertex does not have outgoing edges, it is started immediately. If there are multiple such vertices, they are all started simultaneously. As soon as a deployment process finishes, the corresponding vertex is removed from the graph. When that happens, the while loop may find new vertices available for immediate processing. This process repeats itself until all vertices have been removed from the graph.

The algorithm in Figure 5 is an extension of the previous algorithm. It uses the same basic steps, but functions differently when a deployment process finishes. The previous algorithm would just remove the vertex from the graph, but the extended algorithm checks if it is preferable to swap

```
procedure REMOVE($vertex, graph$)
    if $vertex.type = node$ then
        $mw \leftarrow 0$
        $mv \leftarrow vertex$
        for $v$ in $graph.vertices$ do
            if $v.type = node$ and WEIGHT($v$) $> mw$ then
                $mw \leftarrow$ WEIGHT($v$)
                $mv \leftarrow v$
        if $mw >$ WEIGHT($vertex$) then
            $cloud.swap(vertex, mv)$
            $graph.remove(mv)$
        else
            $graph.remove(vertex)$
    else
        $graph.remove(vertex)$

procedure WEIGHT($vertex$)
    if $vertex.indegree = 0$ then
        return $vertex.value$
    else
        $mw \leftarrow 0$
        for $edge$ in $vertex.entering\_edges$ do
            $mw \leftarrow max(mw,$ WEIGHT($edge.src\_vertex$))
        return $vertex.value + mw$
```

Figure 5. Algorithm for deploying a distributed application using the weighted dependency graph

the finished vertex with another one. It first checks if the deployment step of the vertex is the creation of a node. If it is, it calculates the weight of the finished vertex and compares it with the weights of all other node creation vertices in the graph. If another vertex has a higher weight, it swaps this one with the originally finished vertex, and at the same time swaps the designation of the involved VMs in the cloud. If one of the mentioned checks is negative, the original vertex is removed from the graph.

The calculation of the weight is a recursive function. If the number of entering edges is zero, the value of the vertex is returned. If there are entering edges, on the other hand, it calculates the weight of the source vertices of these edges. The function then returns the maximum of these values plus the value of the vertex itself. The resulting weight is then the total number of seconds previous deployments on average took for completing this deployment step and all steps that depend on it.

V. EXPERIMENTS

Two examples of distributed applications are used to experiment with the presented deployment strategies. The first example is a simple webserver cluster. The setup of such a system consists of an HTTP loadbalancer and several webserver nodes, see part (b) of Figure 1 for more details. The second example is a Storm [19] cluster. Its setup consists of a ZooKeeper node for distributed coordination, a master node called Nimbus and several worker nodes containing Supervisors. Part (c) of Figure 1 shows the components of a Storm cluster and their communication links in more detail.

Each cluster is deployed more than 200 times to three

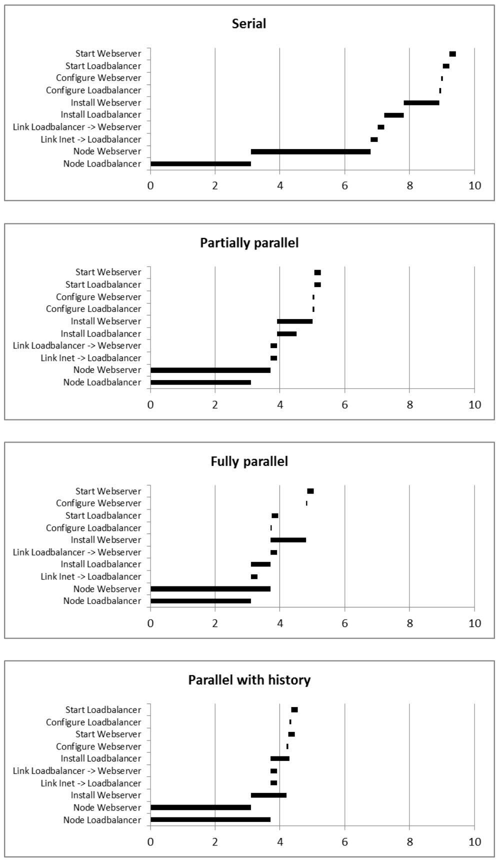

Figure 6. A typical example of timelines for the deployment of a webserver cluster. Deployment time is in minutes.

public cloud providers (Amazon [20], GoGrid [21] and Rackspace [22]) using the four deployment strategies. Figure 6 presents a comparison of the four strategies for a sample deployment of the webserver cluster. The serial strategy is the slowest, needing 9 minutes and 25 seconds to complete the deployment. The other strategies are much faster, needing 5:15, 5:03, and 4:33 respectively. The parallel with history strategy pays off here, because the installation of the webserver takes a long time (1:06) compared to the installation of the loadbalancer (0:36), and the node creation of the webserver (3:42) is quite slow in this sample compared to that of the loadbalancer (3:06).

Table I presents the statistics for the individual deployment steps of the webserver cluster and the Storm cluster. The software installation step depends heavily on the type of software that is to be installed. For example, the installation of Nimbus takes 103 seconds on average at Amazon, while the ZooKeeper installation only takes 46 seconds. The software configuration and starting steps on the other hand are very much alike for the different types of software, and are therefore not shown in the table. The same applies to the creation of the nodes and links.

As was argued in Section III-D, the parallel with history

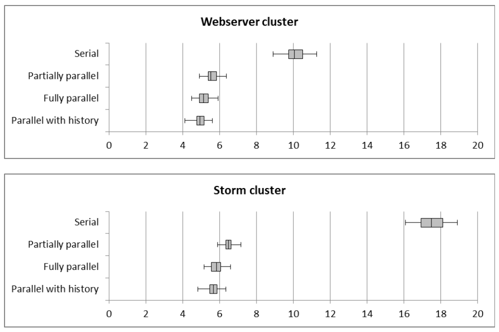

Figure 7. Statistics for the total deployment time, in minutes, of two clusters using four deployment strategies. The box with whiskers shows the minimum, first quartile, median, third quartile and maximum values.

strategy works best if there is a large variety in node creation time. The Amazon, GoGrid and Rackspace clouds have standard deviations of 23, 9 and 34 seconds respectively for node creation. This strategy will therefore work best in the Amazon and Rackspace clouds.

Figure 7 shows the statistics for the total deployment time of the webserver cluster and the Storm cluster, based on all deployments performed during the experiments. The first fact to stand out is that the serial strategy is heavily influenced by the number of nodes that are created, hence the Storm cluster deployment (with three nodes) takes about 17 minutes compared to around 10 minutes for the webserver cluster (with two nodes). The differences between the three parallel strategies are less profound. For the webserver cluster, the deployment times are 5:31, 5:08 (7% speed gain) and 4:57 (4%). For the Storm cluster, the deployment times are 6:30, 5:50 (10% speed gain) and 5:41 (3%).

VI. Conclusions and Future Work

The amount of time the deployment of distributed applications takes depends on several factors. Some of these factors are under the influence of the user and some of them are not. The user does have an influence on the timing and ordering of the different steps in the deployment process. This paper shows that it is possible to decrease the deployment time of distributed applications on cloud computing infrastructures by automating a specific ordering of the deployment steps. In the most simple, serial strategy, each step must wait until the previous step has finished. The partially parallel strategy uses no knowledge of the application being installed, but is able to limit the required time substantially. The fully parallel strategy uses some knowledge about the application being deployed and can achieve even better deployment times. The parallel with history strategy uses a historical database with performance metrics to assign fast-booted VMs to the most time-consuming software installations.

In this paper we have looked at deploying exactly the prescribed number of nodes. If it is necessary to improve the speed of the deployment even further, there are some

Table I
STATISTICS FOR THE INDIVIDUAL DEPLOYMENT STEPS OF THE WEBSERVER CLUSTER AND THE STORM CLUSTER AT THREE PUBLIC CLOUD PROVIDERS, IN SECONDS. MIN = MINIMUM; MED = MEDIAN; MAX = MAXIMUM; AVG = AVERAGE; SD = STANDARD DEVIATION.

	Amazon					**GoGrid**					**Rackspace**				
	min	med	max	avg	sd	min	med	max	avg	sd	min	med	max	avg	sd
Node	181	211	302	218	23	171	179	219	183	9	170	231	407	236	34
Link	8	23	37	21	7	10	12	23	12	2	15	34	43	30	9
Install															
Loadbalancer	32	36	46	36	3	17	19	22	20	2	22	28	40	29	5
Webserver	63	68	96	68	6	17	19	21	19	2	25	31	46	32	6
ZooKeeper	40	45	72	46	6	40	44	58	45	4	58	74	149	80	21
Nimbus	99	102	115	103	4	75	81	96	81	5	158	210	330	222	38
Supervisor	93	101	119	102	6	73	82	88	82	4	164	209	363	223	46
Configure	1	4	5	3	1	1	1	2	1	1	3	5	6	4	1
Start	7	12	16	12	2	10	11	13	11	1	15	17	24	17	2

additional methods we can use. At the start of the deployment process, we can create more nodes than necessary and use the nodes that are available first. Another option is to create a few spare nodes and keep them standby until a deployment is needed. Both methods involve renting resources that are unused for a potentially long amount of time, and their benefits must therefore be carefully compared with their costs. In future work we would like to explore these possibilities more and elaborate on the most balanced strategy incorporating the available options.

VII. ACKNOWLEDGMENTS

This publication is supported by the Dutch national program COMMIT and the TNO program AMSN.

REFERENCES

[1] P. Mell and T. Grance, “The NIST definition of cloud computing,” 2013.

[2] M. Hasan, E. Magana, A. Clemm, L. Tucker, and S. Gudreddi, “Integrated and autonomic cloud resource scaling,” in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 1327–1334.

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

[4] E. Unal, P. Lu, and C. Macdonell, “Virtual application appliances in practice: Basic mechanisms and overheads,” in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010, pp. 213–222.

[5] N. Silva, “APT howto,” http://www.debian.org/doc/manuals/apt-howto, 2005.

[6] “YUM,” http://yum.baseurl.org.

[7] M. Parnell, “apt-fast,” https://github.com/ilikenwf/apt-fast.

[8] “Planned features for YUM,” http://yum.baseurl.org/wiki/YumFuture.

[9] M. Burgess, “A site configuration engine,” *Computing systems*, vol. 8, no. 2, pp. 309–337, 1995.

[10] S. Walberg, “Automate system administration tasks with puppet,” *Linux Journal*, vol. 2008, no. 176, p. 5, 2008.

[11] R. Sabharwal, “Grid infrastructure deployment using smartfrog technology,” in *Networking and Services, 2006. ICNS ’06. International conference on*, 2006.

[12] P. Fan, J. Wang, Z. Zheng, and M. Lyu, “Toward optimal deployment of communication-intensive cloud applications,” in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 460–467.

[13] D. Wu, J. He, Y. Zeng, X. Hei, and Y. Wen, “Towards optimal deployment of cloud-assisted video distribution services,” *Circuits and Systems for Video Technology, IEEE Transactions on*, 2013.

[14] M. Rehman and M. Sakr, “Initial findings for provisioning variation in cloud computing,” in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 473–479.

[15] M. Mao and M. Humphrey, “A performance study on the vm startup time in the cloud,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 423–430.

[16] 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*, 2011, pp. 104–113.

[17] S. Kadirvel and J. A. B. Fortes, “Grey-box approach for performance prediction in map-reduce based platforms,” in *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, 2012, pp. 1–9.

[18] P. Lipton and S. Moser, “Topology and orchestration specification for cloud applications,” 2013.

[19] “Storm,” http://www.storm-project.net.

[20] “Amazon web services (AWS),” http://aws.amazon.com.

[21] “GoGrid,” http://www.gogrid.com.

[22] “Rackspace,” http://www.rackspace.com.