UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering

Iurii Tverezovskyi

CloudML based dynamic deployment configuration for scaling enterprise applications in the cloud

Master's Thesis (30 ECTS)

Supervisor(s): Satish Narayana Srirama, Phd

Tartu 2015

CloudML based dynamic deployment configuration for scaling enterprise applications in the cloud

Abstract:

In light of the popularity of cloud computing, it is really important to be free to change cloud providers when needed. However, in most cases, system configuration relies on provider services, such as load balancing or databases and this makes it much harder to change the provider. This thesis describes a CloudML based solution that is capable of deploying complex systems using different cloud providers with minimum changes including embedded load balancers. This feature allows the creation of scalable configurations that are independent from providers' load balancing services and allow any component of the system to be scalable on demand. The proposed solution also has an integrated generic LP (linear programming) model to control system scaling. We conducted a number of experiments to show that the system could be used for deploying complex systems that follow most popular workflows. The results of the experiments show that this system is capable of scaling properly to support incoming workflow regardless of chosen workflow or number of the system components.

Keywords:

Cloud computing, CloudML, load balancing

CloudMLil põhinev dünaamiline paigalduskonfiguratsioon enterprise-rakenduste skaleerimiseks pilves

Lühikokkuvõte:

Arvutustehnika populaarsuse seisukohalt on väga tähtis pilve tarnija muutmise võimalus. Kuigi, enamikul juhtudest süsteemi konfiguratsioon sõltub tarnija teenusest. Näiteks koormuse tasakaalustamisest või databaasidest, ning see teeb tarnija muutmise raskemaks. See töö seletab CloudMLile põhinevaid lahendusi, mis on võimeline rakendama keerulisi süsteeme kasutades erinevaid pilve tarnijaid. Selline võimalus lubab luua kaleeritavat konfiguratiooni mis on iseseisvad tarnija koormuse tasakaalustamise teenusest ning lubavad igal osal süsteemis olla skaleeritav nõudlus. Pakutav lahendus omab ka integreeritudgeneerilist LP(lineaarne programeerimine) mudelit kontrollimaks süsteemi ketendamist. Me tegime eksperimente läbi näitamaks kuidas süsteem võib olla kasutusel rakendamaks keerulisi süsteeme mis on väga populaarsed. Tulemus näitas et süsteem on võimeline skaleerima ja toetama sissetulevat töökorraldust hoolimata komponentide arvust.

Võtmesõnad:

Pilvearvutus, CloudML, koormuse tasakaalustamine

Table of Contents

List of Figures

List of Tables

1 Introduction

For the past few years cloud solutions has been rapidly growing category, and it does not seem like it will change anytime soon. Every year more and more companies will use the advantage of cloud solutions. Gartner Research predicts that by the end of 2014, cloud solutions will be a $150 billion industry. Many companies decided to move their IT infrastructure to cloud completely or at least partly [1]. Without a doubt, the main reason is a possibility to decrease the cost of managing IT infrastructure. Another very important reason is that by using cloud solutions companies can easily scale services to support new clients. After all, companies that are using cloud solutions are more adaptable [2]. Every company is trying to make their services fast, reliable and reachable for their clients and cloud solutions propose to make this much easier task. Additionally cloud solutions can bring new features for instance synchronization between all devices etc. Talking about cloud solutions we can distinguish three main deployment models, depending on owner of computation resources:

- Public cloud – is a model where computation resources available virtually for anyone. The client can rent almost infinite amount of resources. The provider takes care about infrastructure managing and usually take responsibility for managing some level of SLA.
- Private cloud – is a model where the client owns all computation resources, which are available in boundaries of the organization. The client has to manage whole infrastructure and is responsible for SLA by itself.
- Hybrid cloud – is a combination of public cloud and private cloud. Main advantages are: client can store sensitive information in the private cloud with restricted access and use public cloud for all other services.

Each deployment model usually can support any category of cloud computing service: PaaS (Platform as a Service), SaaS (Software as Server) or IaaS (Infrastructure as a Service).

- Software as a Service (SaaS) is a model where the client has access to software and resources that are hosted by the provider.

- Platform as a Service (PaaS) model provide frameworks and tools necessary for development, management and provision of applications. Usually used for software that has typical requirements and architecture. The provider handles all hardware and most of software management.
- Infrastructure as a Service (IaaS) is a model where the client has access to computation resources without any limitation on software, but the drawback is that all software management has to be handled by the client.

Cloud solutions whether they are PaaS, SaaS or IaaS have proved their reliability, cost effectiveness and even more importantly elasticity. The notion of elasticity embodies the ability of the infrastructure to scale up or down depending on the current workload. Scaling allows for minimizing the cost of running the application and, importantly, the ability to change the structure.

1.1 Problem statement

Scaling of enterprise size systems is a complicated task, especially if the system has any legacy components [3]. To use all advantages of cloud software should be restructured and divided into components to maximize overall productivity [4]. In this case, each component can be represented as a single VM instance.

There are many tools that can help to transfer applications to cloud or to build one from scratch using all advantages of the cloud infrastructure. Unfortunately, not everyone can use those tools, for example, companies that are dealing with enterprise level software. First of all, usually those companies have special requirements for security and control over infrastructure and they are really big systems that are hard to modify [5]. Concerning security and control it is hard for companies to rely on someone else to handle all infrastructure because it is a lot of additional and unpredictable risk. That is why hybrid deployment model is becoming more and more popular. Eventually to make the transfer to cloud faster the company will have to cooperate with the provider to optimize software and infrastructure if needed. However, the catch is, that this will make company dependable on this provider because in case company will try to move to another provider it will have to do migration process once again.

Talking about big enterprise software, it is hard not to mention the problem of legacy systems. Parts of the system might be outdated, but it takes a lot of resources and time to

update them. This can make infrastructure management more complicated. Beside this deployment to a hybrid cloud might require two different tools sets, one for public cloud and one for private. This situation does not make deployment any easier and can create additional risks. With all of this considered, the deployment process becomes a not so trivial task.

One of the main reason to move to the cloud is a possibility to scale the application up and down. The scaling process requires an entity that will handle the traffic flow, which have to be distributed between VMs. To accomplish this task in most cases load balancers are used, they may use a different technique to control flow, but the general idea stays the same – distributing traffic between few VMs. Surely, all cloud providers support load balancer but there are a couple of problems in the context of enterprise applications and vendor lock-in. First of all each cloud provider requires us to set load balancer or define a rule to create one in its environment. The second problem is that we need to set a rule for a load balancer for each VM or system component. Moreover, this might create a problem that is hard to solve when we are moving an application from one cloud provider to another.

1.2 Goals of the thesis

The goal is to find the possible solution for building vendor independent deployment models that are optimized for scaling. The optimal solution should compensate for the main problems described before, which are:

- deployment of same configuration to different cloud providers with no or minimal changes
- scaling capability of the deployment model
- integration with load balancing models

1.3 Organization of Thesis

This thesis is organized in this way:

Chapter 2 describes related solutions and techniques that are used now for deployment to a cloud including proprietary solution and multi-cloud deployment tools.

Chapter 3 introduces set of tools and concepts that were used to achieve the goal of the thesis.

Chapter 4 explains overall solution including all algorithms and reasons for architectural decisions. This chapter also includes solution implementation overview.

Chapter 5 includes description and results of three experiments that we conducted in order to evaluate the solution. The first experiment shows the possibility of using proposed solution for deploying real world software to the cloud. Second and third experiments display deployment of the systems that are using parallel and exclusive workflow respectively.

Chapter 6 explains what was achieved as well as possible areas where presented deployment tool can be used.

2 Related work

This chapter presents existing solutions for deployment of software applications into the cloud. First we will review platform specific tools and solutions from major cloud providers. Such solutions [8] [9] provide better integrations with all services and could be more efficient, however, deployed configurations cannot be migrated easily. To solve this problem multiple multi-platform deploying systems [10] [11] has been developed over last few years. Furthermore, we will review techniques for deployment software into the cloud.

2.1 Platform specific deployment tools

It is obvious that all major players in cloud solutions industry propose their own tools designed to make deployment to the cloud as easy as possible. The main advantage is ability to benefit fully from all internal services and infrastructure.

2.1.1 Google Cloud Deployment Manager

Google Cloud Deployment Manger [8] is available for Google Cloud Platform users. Cloud Deployment manager allows to declare, deploy and manage infrastructure using the concept of templates. These templates are a JSON or YAML file that consist of descriptions for how to deploy services to the cloud. Example of deployment configuration using YAML:

```
 1 resources:
 2 - type: compute.v1.instance
 3   name: vm-my-first-deployment
 4   properties:
 5     zone: us-central1-b
 6     machineType: https://www.googleapis.com/...
 7     disks:
 8     - deviceName: boot
 9       type: PERSISTENT
10       boot: true
11       autoDelete: true
12       initializeParams:
13         diskName: disk-my-first-deployment
14         sourceImage: https://www.googleapis.com/...
15     networkInterfaces:
16     - network: https://www.googleapis.com/...
17       accessConfigs:
18       - name: External NAT
19         type: ONE_TO_ONE_NAT
```

Each template contains a number of modules, where each module is a resource that has to be deployed into single VM. There is a possibility to specify action after instance deployment like software installing, configuration, etc. This deployment mechanism allows to

modify deployed configuration by updating configuration file and executing the deployment again.

2.1.2 Amazon CloudFormation

Amazon has a number of solutions dedicated for deployment software application, in comparison to Google, for example, AWS Elastic Beanstalk, AWS OpsWorks, AWS CloudFormation [9]. AWS Elastic Beanstalk is a most basic solution designed for easy deployment. However it lacks some configuration options. AWS Elastic Beanstalk is designed mostly for the deployment of a single application and because of this there is no good way to connect and manage multiple connected software components that are supposed to be deployed in different VMs in the cloud. Next in the line is much more powerful AWS OpsWorks. AWS OpsWorks uses Chef recipes for deployment monitoring and changing software configuration. Example of such recipe:

```
include_recipe 'deploy'

node[:deploy].each do |application, deploy|
  opsworks_deploy_dir do
    user deploy[:user]
    group deploy[:group]
    path deploy[:deploy_to]
  end

  opsworks_deploy do
    deploy_data deploy
    app application
  end
...
```

The main aadvantage of AWS OpsWorks is the notion of stacks and layers. The stack is a set of EC2 instances, load balancers and DB instances that represent same software component of the system. In other words stack in this context is an infrastructure needed to support particular system component. Layers describe set of software that have to be installed to support system components. Nevertheless, considering the amount of control over deployment process AWS CloudFormation is most obvious choice. This service allows to use the same notion of stacks from AWS OpsWorks to describe service architecture and resources but instead of creating Chef recipes it allows the use of the JSON format. Furthermore AWS OpsWorks makes it possible to use all services including AWS Elastic Beanstalk and set any policies available for AWS infrastructure. For example scaling police:

```
"WebServerScaleUpPolicy": {
  "Type": "AWS::AutoScaling::ScalingPolicy",
  "Properties": {
```

```
    "AdjustmentType": "ChangeInCapacity",
    "AutoScalingGroupName": {
      "Ref": "WebServerGroup"
    },
    "Cooldown": "60",
    "ScalingAdjustment": "1"
  }
}
```

Single deployment file also includes infrastructure configuration, which makes it really easy to keep configurations for software components and overall system.

All those tools share the same problem - they are platform dependent and, therefore, there is no easy way to switch cloud provider. More importantly for enterprise clients, there is no clear way to use those tools in case of using hybrid cloud services.

2.2 Multi-platform deployment tools

In comparison to platform-specific deployment tools multi-platform deployment tools allow the deployment of the configuration to a number of different cloud providers. Since in most cases such tools are open source they do not have as strong support from providers as their own tools, but can propose a solution of vendor lock problem.

2.2.1 Jclouds

Jclouds [10] is Java based open source library created to work with more than 30 different cloud providers. Basically, jclouds is a wrapper that allows one to write configuration once and then launch it using any supported provider, in this way the library can guarantee a high level of portability. It is possible to describe system architecture using templates. Template contains information about software, hardware, location and some additional options of each resource in the system. Library allows to upload any binary files to created resource.

Another possible option is Libcloud [11]. This python based library supports a number of providers as well as third party extensions. There is a list of APIs to describe system architecture “Compute”, “Object Storage”, “Load Balancer”, “DNS”. “Compute” helps to create and manage virtual servers. “Object Storage” is responsible for communication with any storage object, for instance, Amazon S3, OpenStack Swift, etc. “Load Balancer” API provides unified access to load balancers in different platforms. And last but not least is “DNS” API provides control over specific DNS options, in case if provide support this.

2.2.2 Apache Brooklyn

Apache Brooklyn is power tool based on Jclouds for deploying and managing applications in the cloud. It allows the storing of configurations in YAML file called blueprints. Such blueprints contain both descriptions of hardware infrastructure such as VM configuration and networking as well as a description of software setup. Example of such blueprint:

```
location: localhost

services:

- type: brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  brooklyn.config:
    wars.root: ...
    java.sysprops:
      brooklyn.example.db.url: >
        $brooklyn:...

- type: brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: My DB
  brooklyn.config:
    creationScriptUrl: ...
```

Blueprints can be stored into local catalogs. After deployment Apache Brooklyn provides web based management and provision system, however, changes in architecture require redeployment of the whole system and there is no good way to control scaling of the system.

2.3 Private and hybrid cloud management platforms

Talking about deployment to cloud it would be wrong not to mention hybrid and private cloud management platforms. There are a lot of cases when the company might need to have their own private or hybrid clouds. Some of the reasons might be security requirements or economic necessity. However not every company can build their own platform from scratch, and best solution is this case to use publicly available or open source platforms. There are a few main players in this market - some of them are OpenStack, VMware vCloud Suite, IBM Cloud and OpenNebula. OpenStack and OpenNebula are open source projects and VMware vCloud Suite and IBM Cloud are proprietary software.

OpenStack [12] is supported by OpenStack Foundation and developers' community. OpenStack allows the control of all aspects of infrastructure networking, storage and of

course computing. OpenStack is an operating system created to manage a large pool of resources. Using the dashboard or using API administrator can easily create and manage all available resources, assign automatic tasks and monitor the current situation of the whole cloud. It is also possible to use OpenStack to create a hybrid cloud in pair with, for example, Amazon.

OpenNebula [13] is a really similar project as in any other cloud management platform it allows to create templates of VM's and make them available for users similarly to any public platform. OpenNebula has automatic auto scaling mechanism if this allowed by the administrator.

Another interesting project is VMware vCloud Suite [14], its main advantage is proprietary VMWare virtualization technologies which are used to create new VMs for the client. VMware has whole ecosystem capable to solve any problem with virtualization, but this is a problem as well because most of technologies and tools are proprietary and will work only in this ecosystem.

It is easy to see that tools to create public or even hybrid cloud are available on the market, but their focus on managing and supervising hardware infrastructure, networking and not on software deployment. So the problem with dynamic deploying to the cloud is still actual.

3 Background

This chapter gives an in-depth review of frameworks and technologies that were used to create proposed solution. Firstly we will describe CloudML its advantages and disadvantages as well as internal structure and main functionality. After that we will show the main idea behind LP (Linear Programing) load balancing model and also explain in depth why this type of load balancing model has been used.

3.1 CloudML

CloudML [15] is a framework for deploying and provisioning application in the cloud. CloudML is built on top of Apache jclouds and is used for actual deployment to the cloud. Moreover, since Apache jclouds is a multi-cloud toolkit CloudML supports almost the same number of providers [15]. Also, it allows simple system provision during deployment and after for example querying VM instance status. Unfortunately the framework does not allow monitoring VM instances, CPU load or number of incoming requests. CloudML allows to describe system once and deploy it to any supported cloud platform without changes or with minimal changes. For this purpose the framework contains a list of elements that help to describe deployment model configuration.

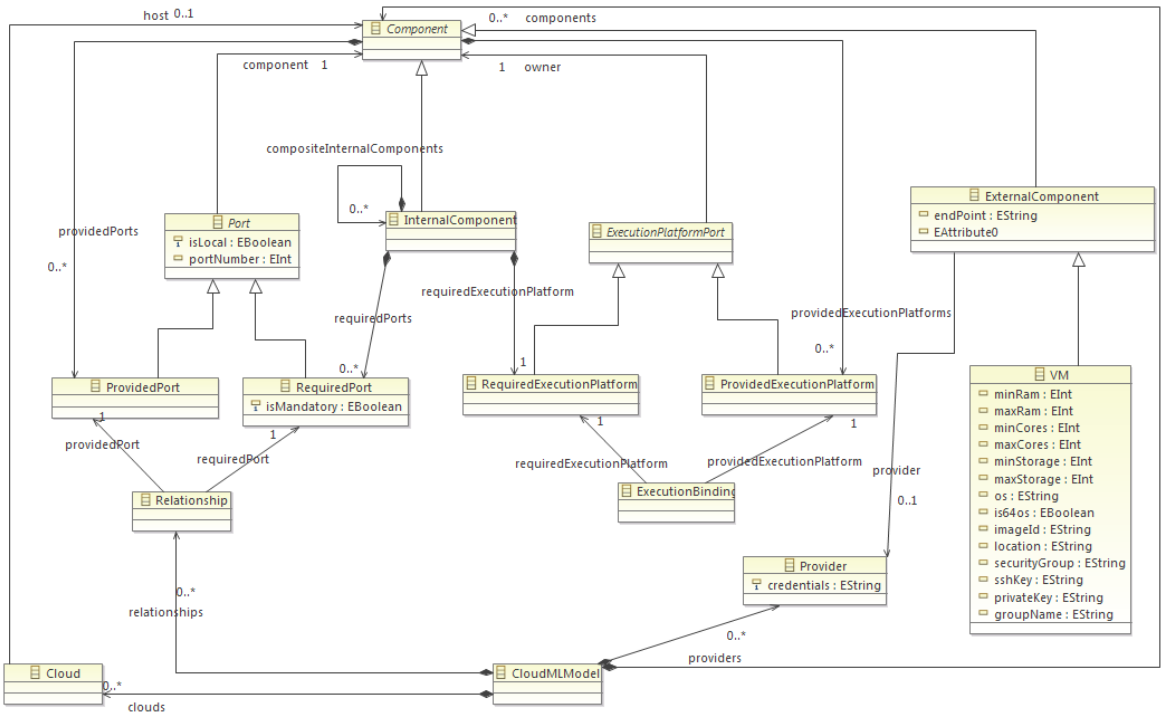

Figure 1. CloudML Metamodel [15]

- Component – abstract parent class for ExternalComponent and InternalComponent
- InternalComponent – entity that represents service that will be created during deployment and stores set of Resources, Ports and ExecutionPlatforms
- ExternalComponent – represents service that already exists outside of current model
- VM – represent VM instance configuration in model
- ExecutionPlatform – represent requirement that Component has regarding VM
- Relationship – represents binding between two components (Internal or External) and has its own set of Resources.

CloudML allows to describe configuration model using JSON file or by writing configuration using Java. For actual deployment CloudML framework uses CloudML Engine, which proposes three deployment methods. Java API for the model described using Java, command line interface for the model described by JSON and web sockets interface. Part of model description using JSON:

```
"internalComponentInstances": [
  {
    "eClass": "net.cloudml.core:InternalComponentInstance",
```

```
    "name": "client--1",
    "type": "internalComponents[client]",
    "requiredPortInstances": [
      {
        "eClass": "net.cloudml.core:RequiredPortInstance",
        "name": "requiredPort-433795083",
        "type": "internalComponents[client]/requiredPorts[requiredPort]"
      }
    ],
  }
]
```

The main advantage of such method for presenting configuration is its simplicity in terms of tools independency. On the other hand, there are a lot of disadvantages. It is really hard to write the configuration for a large; there is not any type of validation of any kind and the only way to check if the configuration is correct is to deploy it. Java configuration gives a bit more tools in developers' hands. First of all it provides at least simple type matching for model validation. Secondly it is easier to read and understand configuration model. Part of model description using Java:

```
InternalComponent wiki = new InternalComponent("wiki", platform);
wiki.getResources().add(wikiSoft);
wiki.getRequiredPorts().add(dbRequiredPort);
wiki.getProvidedPorts().add(defaultProvidedPort);
wiki.setRequiredExecutionPlatform(platform);
```

The downside of using Java is that it requires a fairly complicated process of setting up. Regardless of chosen way of model description deployment process stays the same (see Figure 2).

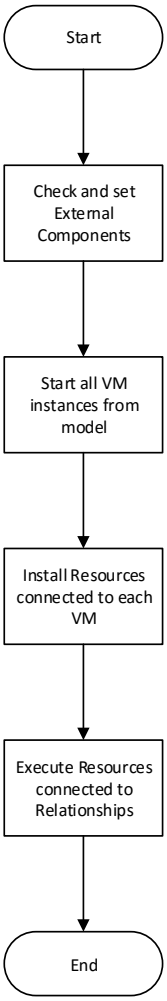

Figure 2. CloudML deployment process

3.2 Load balancing models

A number of solutions for load balances have been proposed. Some models are trying to predict workload depending on factors like workload trends during the work day, daily history of workload. Others react based on the current situation [16] [17] CPU usage, response time. Such load balancing models require extensive monitoring mechanism. Unfortunately, each provider has a proprietary monitoring system, for instance, Amazon has Amazon CloudWatch [18]. Considering that we are aiming for multi-cloud deployment tool, we can rely on platform specific monitoring platforms. Therefore to support described type of load balancers and provide multi-cloud deployment capability, such system has to include monitoring platform. However, in this case, monitoring platform would require additional resources to support this monitoring platform and make the system overall less efficient and cost effective.

To address above problem we propose to use LP (Linear Programing) load balancing model [19]. This model takes into account all major parameters such as the type of the instance, cost, and performance capably, limit of the instances and as a result produces an optimal configuration. Moreover to be as cost effective as possible this load balancing model heavily uses the idea that most cloud providers charge for resources hourly. This means that if the instance was started only for 1 minute will still would have to pay for one hour. Therefore, since VM instance is already started, there is no reason to kill it, during the first hour. This load balancing model has proved that such approach is very cost effective [19].

One of the key notions used to describe this LP based load balancers is time bag. Time bag describes time instance life time e.g. if instance has been created at 2:21 PM and currently its 2:55PM then VM is in 34 time bag, and if right now is 3:22PM VM is in 2nd time bag.

The main principle that dictates system behavior is keeping in balance cost or terminating instance and cost of keeping instance alive. The key in this process is calculating start time and killing time of each instance. Time of VM life minus start time and killing time is a period during which we actually use this VM. Considering all this, we need to add one more state for VM – marked to be killed. The instance is marked to be killed if current performance of Scalable Component is more than needed, which means that we don't need so many instances right now and some could be killed. However, since we already paid for them we will keep them till the end of running hour. This approach has one more advantage, in case if during this hour load on the server will increase we could get in a situation where we actually need this instance. However, how to determine which one to mark to kill. In the same paper have been developed evaluation technique. Key in there is to keep balance between killing cost and retaining cost. Killing cost is an amount of money that we will lose killing instance immediately and can be calculated using this formula:

$$KC = (TB_m - TB_c) * C_t$$

, where TB_m – maximum number of time bags, TB_c – current time bag of the instance and C_t – cost per hour of instance of the type t. To calculate retaining cost we can use next equation:

$$RC = TB_c * C_t$$

Retaining cost shows how much we already payed for VM instance.

4 Solution

This chapter describes proposed solution and its implementation. Initially, we will review changes that have been made in CloudML to support needed functionality for load balancing. Changes in CloudML includes implementation of new components as well as algorithms for scaling up and down. To support scaling and the possibility to change network routing without changes in the configuration we used DNS rerouting. Last element described in this solution is CMLDep. CMLDep is a simple REST based solution for deploying applications into the cloud, designed to support advanced load balancing.

4.1 CloudML extensions

Since the proposed solution is based on CloudML framework, first of all we need to identify what changes have been made in the framework itself. To allow CloudML to perform scaling more efficiently we extended it with the list of components described below:

- DirectCommand – is a component that allows to the execution of a command in VM after it was created. The general problem of CloudML is that there are only two ways to execute scripts in VM. First – add a resource that will be executed when VM is created and Relationships Resources, which will be executed when both VMs in a relationship are created. However, we need to mention that Relationships require connecting two instances using RequiredPort and ProvidedPort, which is a huge overhead for just executing the script. Moreover, since there is no straight forward way to execute script in VM when it is needed. DirectCommand component solves this problem and it requires only two VMs (server and client) and a command as well. However, the main advantage is that it can be added to the model at any given time.
- LoadBalncerEntity – is a wrapper that contains Component, ComponentInstance, VM and VMInstance of a LoadBalancer. This object can be created only by the system.
- ScalableComponent – is a new component that allows one to create truly scalable and flexible deployment models. The idea behind ScalableComponent is that it allows to register any ExecuteInstance (VM and its software) as a ScalableComponent which can instantiate LoadBalncerEntity or new VMs. During scaling up the component will automatically create LoadBalancer and connect instances of given

component to it. Also, if the given component had incoming connected from other component using Relationship it will automatically reroute it to LoadBalancer.

List of terms that are used in this chapter:

- System component – is a set of software that should be installed in a single VM.
- Initial configuration – is a deployment configuration that represents fully viable system where each model component represented by one VM and all necessary connections between components are established.
- Scaling up scalable component – process of creating and adding a new instance into the scalable component. May include initiating of the load balancer.
- Scaling down scalable component – is the process of removing VM instances from the scalable component. It consists of two phases according to [19]. In phase 1 instance or instance may be marked to be „killed "depending on the output of LP model. In the second phase if there are no changes in LP model instance or set of the instance will be terminated. In case if only one VM is left in scalable component load balancer will be also removed.

4.2 Scalable component

Scaling is a key functionality for any cloud application. In proposed solution, each scaling component has the ability to scale separately. To support such functionality in scalable component two algorithms were implemented: scaling up and scaling down. For practical use they were combined in a single function scale module. To change the configuration of a particular scalable component function scale has to be called with a parameter that contains the new desirable configuration. According to the new configuration scalable component is able to determine by itself which instances have to be killed or created. A configuration parameter is a simple array of integers that describes a number of VM of each type that scalable component should contain. For example if incoming configuration is [1,0,0,2] and types specified for this scalable components are ["M1.micro", "M1.small", "M3.medium", "M3.large"] then scalable component ideal configuration at this moment should be 1 instance of m1.micro, and 2 instance of "M1.large".

4.2.1 Scaling up scalable component

Scaling up is relatively simple process displayed by Figure 2. where the biggest complication is keeping the connection between scalable components and inside scalable components.

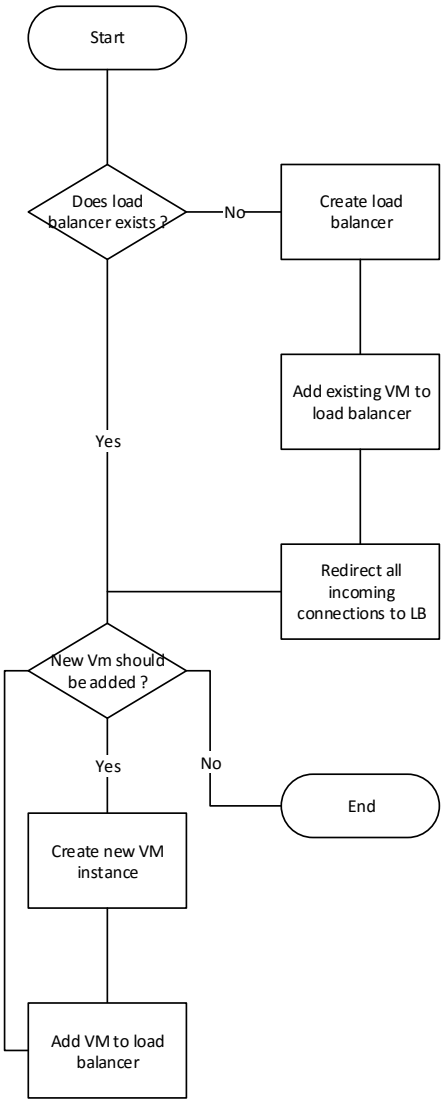

Figure 3. Scaling up algorithm

Another part of the algorithm that is important is initializing load balancer. In initial configuration all relationships connects VM directly as it is showed by Figure 3.

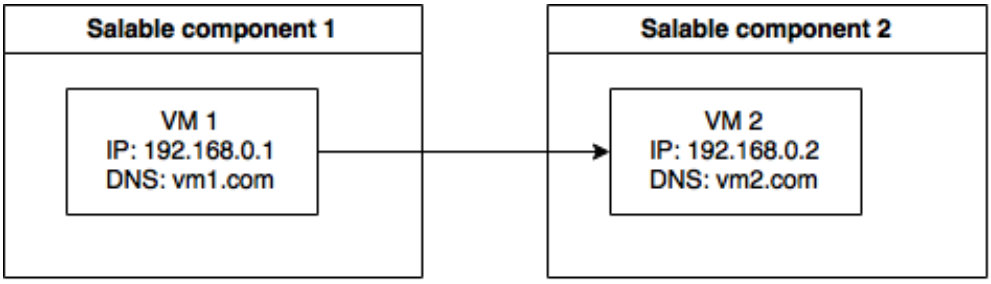

Figure 4. Connection between VMs in initial configuration

However if the scalable component contains more than one VM instance all incoming connection should be rerouted to load balancer as it displayed in Figure 4.

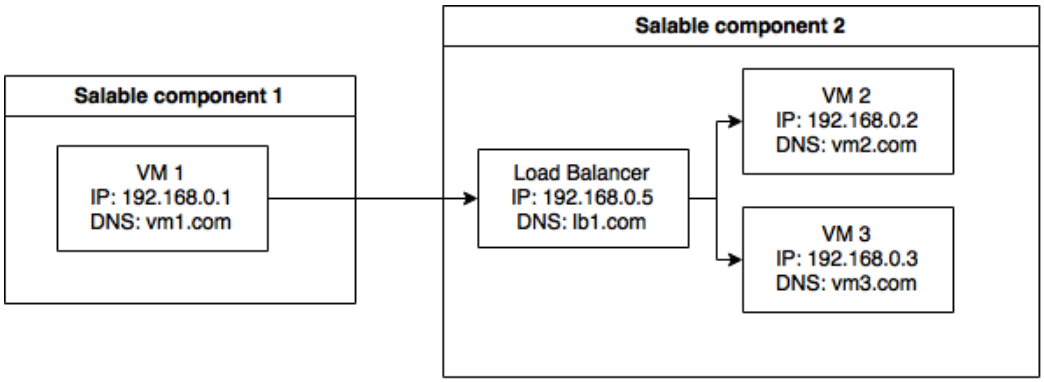

Figure 5. Representation of the deployment mode after scaling up

To achieve this goal in proposed solution we are using DNS caching resolver Unbound[1]. In most cases, DNS name resolving is a simple process. Any OS has a list of trusted DNS servers, and in the case when the system needs to resolve DNS name OS sends a request to one of the DNS servers from the list and receives IP of desired server. In case if Unbound is installed this process gets one additional step. Before to send a request to DNS server system checks if Unbound has a rule for this DNS name and if so the system will use IP that is specified in Unbound configuration file. Otherwise system behaves exactly the same as default scenario. This technique allows us to control traffic flow by setting up rules for DNS names resolving. For example, we need to move from the configuration represented by Figure 4 to a configuration that is displayed by Figure 5. To make "VM 1" to send requests to load balancer with IP 192.168.0.5 instead of "VM 2" with IP 192.168.0.2 we need to add a rule to Unbound that is installed in "VM 1". This rule will specify to resolve DNS name "vm2.com" to IP of load balancer that is 192.168.0.5 instead of 192.168.0.3 that

[1] http://www.unbound.net/

would be resolved by actual server IP. An additional benefit of this solution is that because Unbound is local resolver there is no need to do any changes in the network configuration of all VMs.

Part of Unbound configuration with one rule:

```
server:
    local-zone: "example.com." transparent
    local-data: "example.com. IN A 192.168.1.1"
```

4.2.2 Scaling down scalable component

Scaling down is more complicated process compared to scaling up and it consists of two phases see Figure 6.

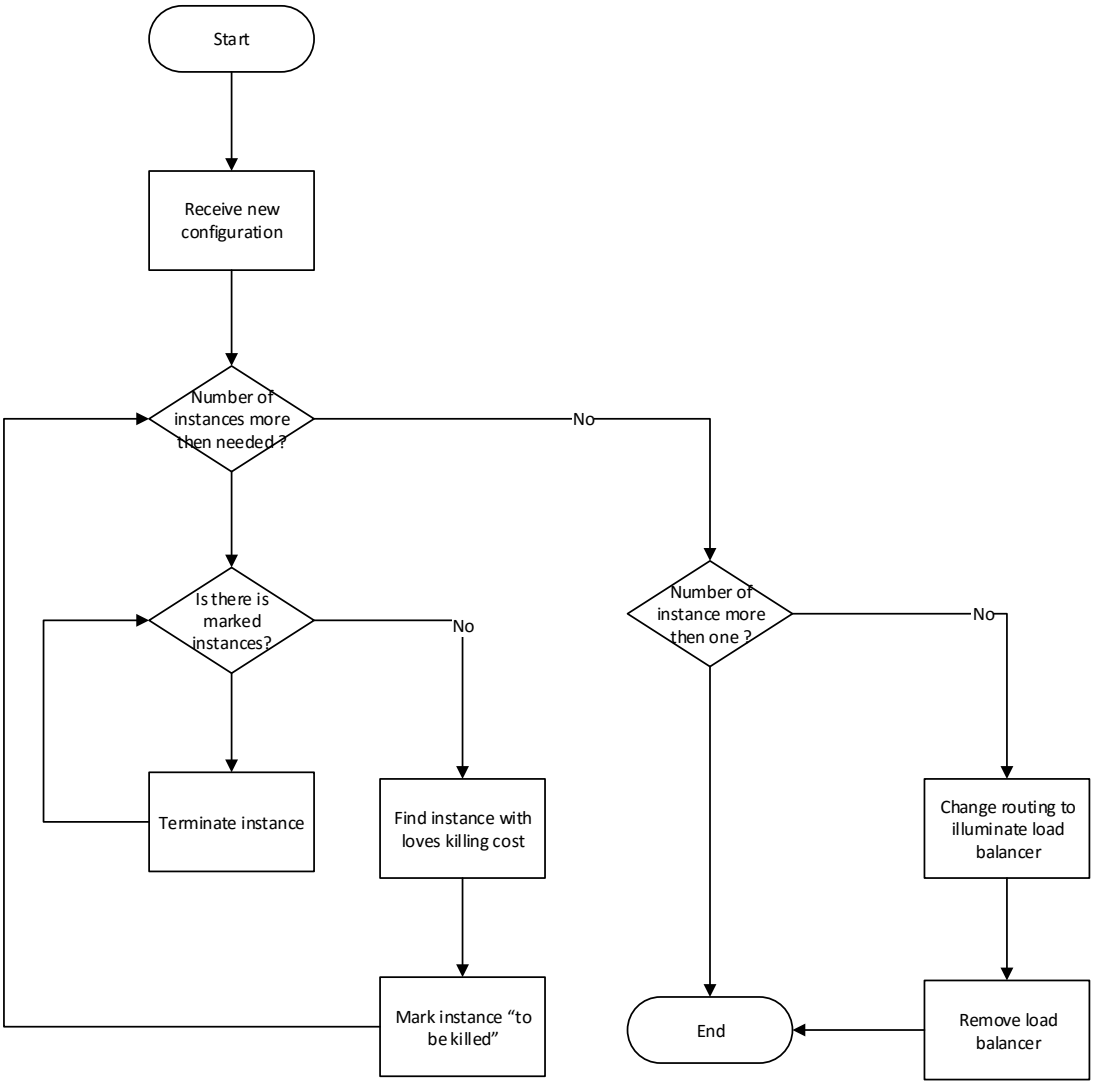

Figure 6. Scaling down algorithm

Since we build system with the integrated LP model[2] we are using all principles proposed in it. In first phase component will find an instance that should be removed from the model but instead of immediate action instance will be marked as instances „to be killed". This means that instance might be killed if it meets LP model requirements. If the instance is actually terminated it will be automatically removed from the load balancer. Moreover, if after termination of particular instance ScalableComponent will contain only one VM, load balancer will also be killed and removed from the model and all connections will be redirected to the only VM in that component. The reason to do this is too keep cost as low as possible

4.3 CMLDep tool

CMLDep (CloudML Deployment tool) is a REST service that connects extended version of CloudML and LP model into one system. As a base for CMLDep we used Spring Boot and Maven 2 as a project management tool. Both CloudML and LP model have been included as local maven dependencies. CMLDep was designed to support any kind of model that can find optimal configuration of the system and not to be limited to current LP model[2], however it was optimized to work with it. See Table 1 to find all available in CMLDep API calls.

URL	Method	Parameters	Purpose
/initial	GET		Deploy initial model
/initial/start	PUT	period	Start automatic scaling of the system
/instance/up	PUT	type, component	Start additional instance of specific type in specific Scalable Component
/instance/down	DELETE	type, component	Remove instance of specific type in specific Scalable Component
/model	GET		Download current deployment configuration

[2] Load balancing models

/initial	POST	Model	Upload initial configuration

Table 1. CMLDep REST API

There are few ways how to set up the initial configuration for CMLDep. First one is to upload JSON based model using API interface, the second option is to write Java class in CMLDep and deploy it with the software itself. Understandably first option is more preferable and probably the main reason is that CMLDep can be deployed first and then configuration model can be added later. As was mentioned before to create JSON based model without any tools is not a trivial task the base way to address this issue is to create Java class that describes the model and then export it as a JSON file.

Once CMLDep is deployed into the cloud it allows to control system deployment process with API calls or can be managed automatically. To support automatic scaling LP model have to calculate optimal configuration continuously or depend on some kind of schedule. As was shown [19] calculating the optimal solution for the large system can take a lot of time. To resolve this obstacle and to move closer to continuous calculation, by default, CMLDep calculates new configuration right after finishing previous calculation, However the period between calculations optimal solutions can be set manually using API. CMLDep relies on LP model to calculate optimal configuration. In its turn, LP model requires knowledge of load of each Scalable Component or using terminology from [19] in each region. We used similar technique described in [19], to get a load from each ScalableComponent - NGINX module HttpStubStatusModule, which allows us to request a number of connections . To monitor this parameter CMLDep has a monitoring module that can access the instance in deployment configurations and request load on each of them. This monitoring module starts when auto scaling is started and runs in separate thread. It is easy to notice that this algorithm of auto scaling creates one series that are demonstrated by Figure 7.

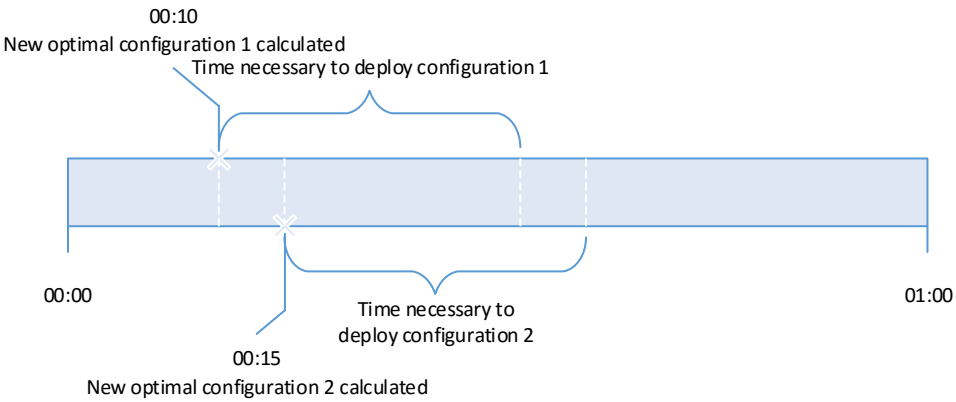

Figure 7. Deployment time line using CMLDep

After optimal model has been calculated it takes some time to actually update the current model. Moreover, during this period system will continue to calculate optimal configuration, which will not include VMs that are not fully configured yet. To eliminate this problem we included in the system configuration all VM's that are in a progress of initiation. In this way LP model will calculate new configuration taking into account all VM instances that are deploying during new calculation. The actual deployment process described in Figure 8.

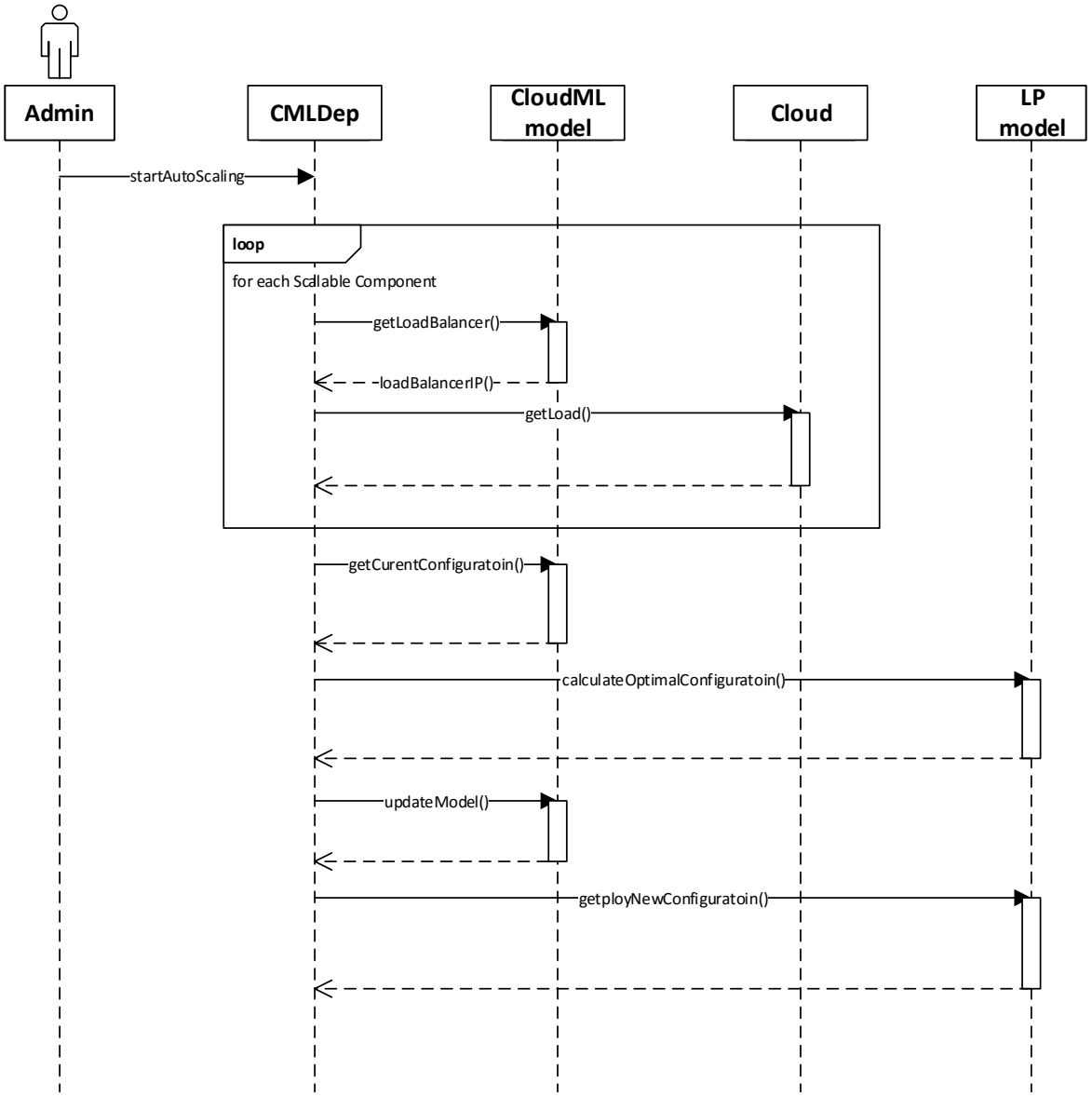

Figure 8. Auto scaling sequence diagram.

Another important part of CMLDep is logger module. Logger is responsible for keeping a record of all log data during deployment of scaling. For example, after each model update, logger module will record new configuration etc.

5 Validation

We conducted three experiments to show that proposed solution can be used in deployment of a vast range of applications into the cloud. First experiment has in mind to show that proposed solution can, in fact, deploy usual applications such as WikiPedia. MediaWiki set up includes connection to database, however, whole system is one component architecture. The next two experiments were designed to show ability of the proposed solution to deploy systems that are built using workflows. We used systems that follow parallel and exclusive workflows.

5.1 Environment

As a cloud provider we chose Amazon EC2 as it is one of the most popular and recognizable. All instances were running Ubuntu 14.04 as an operating system. We used three types of EC instances for all tests M1.small, M3.medium and M3.large, however system is not limed by a number of different types of VMs that can be used. We used instances from two different tiers since M3 tier does not provide small instances. M1.small and M3.medium have one core CPU and M3.large has two cores, however M1 is the previous generation to M3 so performance is different.

5.2 Simulation tools

To simulate load in all tests we used Tsung. Tsung is load testing tool and allows one to simulate load with various rates request per second. To make the test as realistic as possible we used archive of ClarkNet that contains logs from a real web server. However, a number of request per second that we got from ClarkNet was not enough for all experiments. In order to address this problem we scaled a number of requests per seconds. We used a number of Python scripts to extract data from ClarckNet log, convert into a convenient format, scale number of requests, automatically create Tsung configuration file etc. Tsung has additional functionality that, unfortunately, is really poorly described. Tsung can request system information such as CPU load, the amount of used RAM etc. using SNMP protocol [20]. SNMP or Simple Network Management Protocol allows a remote client to request system information over UDP. In our case Tsung already includes SNMP management module, however, client module has to be installed to each VM instance manually in order to conduct measurements. Luckily, setting up and installing SNMP to client VM can be handled by

CloudML. For client side we picked Snmpd application [21]. Example of Tsung configuration file:

```
<arrivalphase phase="1" duration="1" unit="second">
    <users interarrival="0.030303030303030304" unit="second"/>
</arrivalphase>
<arrivalphase phase="2" duration="1" unit="second">
    <users interarrival="0.04" unit="second"/>
</arrivalphase>
```

To include monitoring option config file has to be a bit modified, most basic example is

```
<monitoring>
    <monitor host="myserver" type="snmp"></monitor>
</monitoring>
```

It is worth mentioning that Tsung does not generate an exact number of request every second. Nonetheless this can be ignored, since average is correct. At the end of each simulation Tsung provides statistics with system average response time, total number requests and CPU load because of enabled SNMP. For our experiment we needed different load every second according to ClarkNet archive. The only way to achieve this was to create new arrivalpase for each second of testing.

Considering that our LP model heavily relies on a maximum number of concurrent requests per second for each VM instance type, we tried to measure this parameter as accurately as possible. To do so we will consider CPU load and response time as main parameters. Average CPU time has to be about 80% and average response time under 500ms [22]. To find how many request each type of VM can handle we conducted number of small experiments using Tsung each for 2 minutes. In respect to MediaWiki and our custom application generates different CPU load we made experiments for both of them for each instance type. For MediaWiki setup we populated the database with logs from public available Wikipedia. For testing MediaWiki we requested random page to simulate real workload.

Since we need to have information about workload in each scalable component for LP model, in all experiments we used CMLDep logging functionality. Moreover we gathered data about amount of instances in general. Considering the fact that to get workload in each scalable component we need to make http request we used a 5 minute delay to avoid creating additional workload. All the results presented in this work calculated taking into account this information including all plots.

5.2 One component system deployment

In this experiment we used MediaWiki as a test application for deployment. The idea behind this experiment was to show that scalable component can scale efficiently and doesn't influence overall system performance. Another part of the experiment aimed to show that CMLDep is capable of deploying and scaling single scalable component model which has relationships with external services.

5.2.1 Experiment configuration

The only scalable component will be presented by MediaWiki itself, and MySQL DB presented as an external service installed into M3.medium EC2 instance see Figure 9.

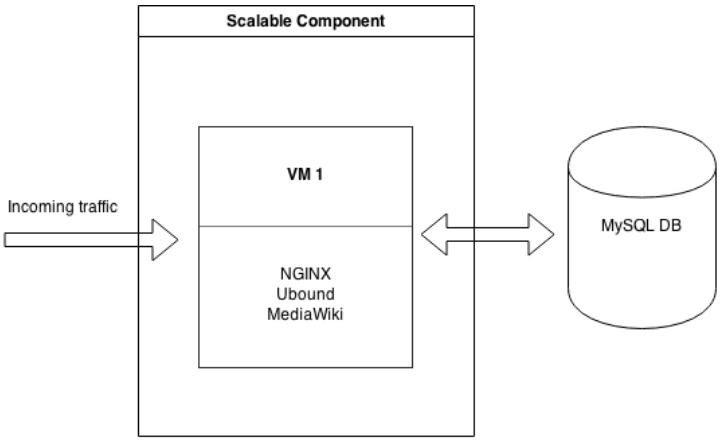

Figure 9. Experiment with one component initial configuration

After bombarding initial configuration with requests generated by Tsung according to workload log from ClarkNet we got modified configuration which is able to handle increased workload see Figure 10.

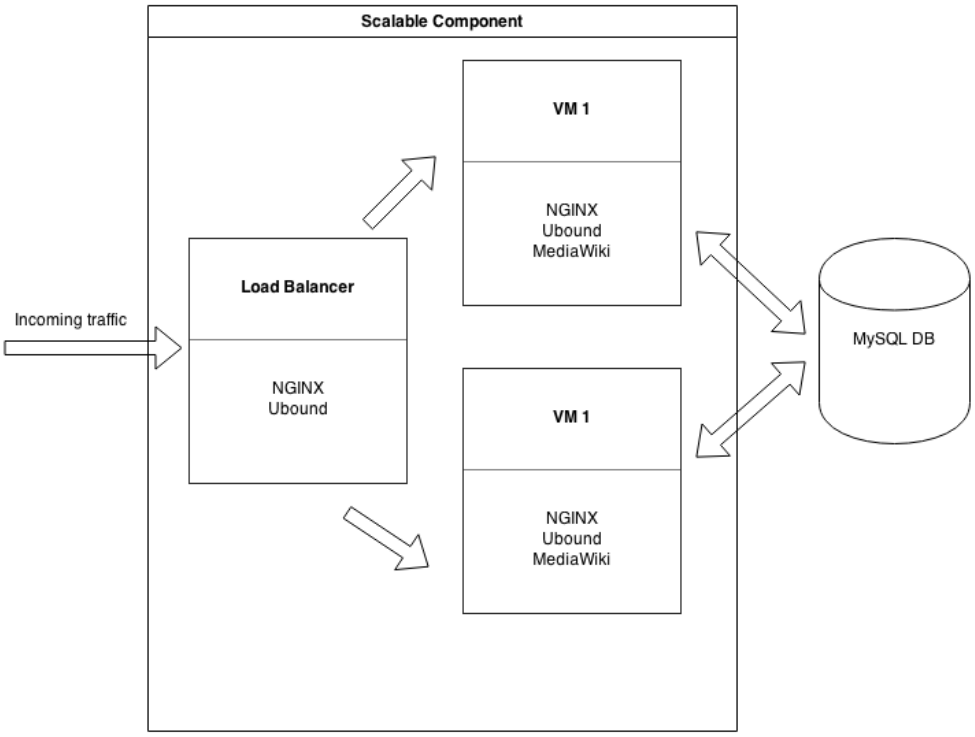

Figure 10. One component deployment configuration in case of increased workload

Results of performance testing for this configuration for each type of instance can be found in Table 2.

EC2 instance type	Average number of request per sec	Average CPU load	Average response time in sec
M1.small	15	~73 %	0.3
M3.medium	31	~82%	0.34
M3.large	63	~78%	0.4

Table 2. EC2 instances performance with MediaWiki

5.2.2 Experiment

To validate results we used data from CMLDep logger and compared it with the load on scalable component see Figure 11. In 78% of the time system was able to handle the incoming workload.

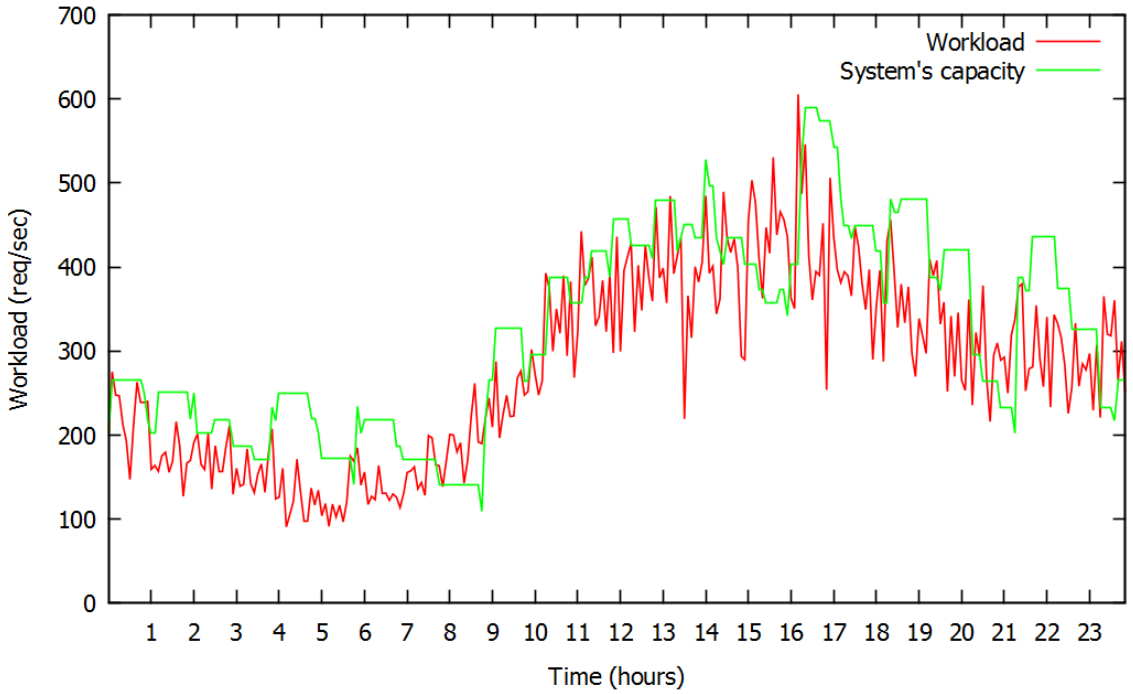

Figure 11. Workload and system capacity during one component experiment

Figure 12 shows the amount of instance of each time in each hour. It is easy to see that system preferred M3.medium instances over M1.small and M3.large.

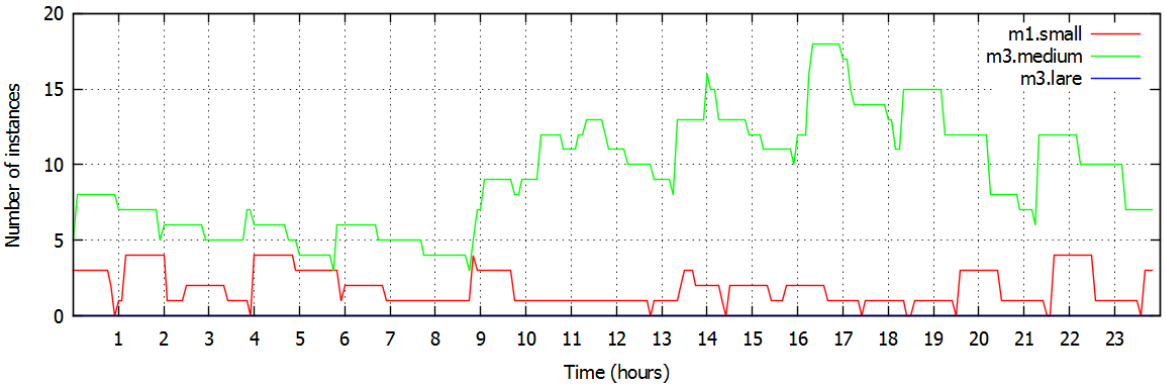

Figure 12. Number of instances of each time during one component experiment

As well we calculated how different average response time during the experiment was to response time in initial configuration model. To do this we tested response in initial model bombarding it with 15 requests per second which is maximum for instance type M1.small. This result can be set as a “normal” response time of this system which is 0.42 seconds. The

difference in “normal” response time and average response time that we got during experiment shows how much scaling process influence the overall performance of the system. For this experiment average response time was 0.51 second ~ 21% slower than normal. Some additional information can be found in Table 3.

Total number of requests	80212
Requests lost	5428
Successful requests	~ 93.2 %

Table 3. One component experimnets results

5.3 Parallel workflow

The goal of the second experiment is to show the possibility of building applications with multiple scaling component that uses parallel workflow. Workflow is a sequence of action or task completion of which will lead to completion[3]. Obviously, for this test we used the system with three scalable components. An example of the model for this experiment presented by Figure 13. In parallel workflow tasks executing simultaneously, this means that after completion Task 1 - Task 2 and Task 3 would start and run at same time.

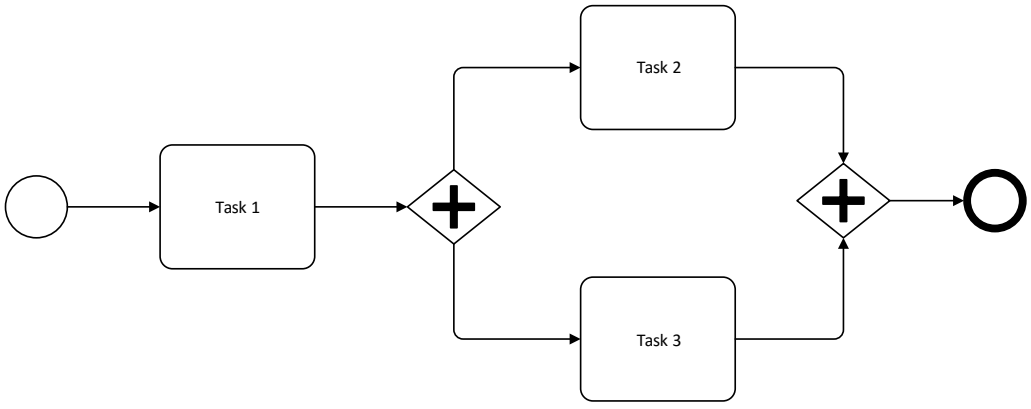

Figure 13. Example of parallel workflow

[3] https://en.wikipedia.org/?title=Workflow

5.3.1 Experiment configuration

For this experiment we chose to build a custom application to simplify deployment configuration and have more control over each element of the system. All three applications were developed using JavaScript and Express framework to support requested model. As HTTP server, however, we used NGINX. There two different application, first one is *mainApplication* represented by Task 1 in Figure 13 and second application *fibApp* represents Task 2 and Task 3. The only responsibility of the *mainApplication* is to send a request to two instance of *fibApp* using round robin. Moreover, *fibApp* is a simple app that calculate Fibonacci number. In the end *mainApplication* response on request only after both instances if *fibApp* responded see Figure 14. In order to fulfill such functionality we used module "async", that allow to send asynchronous requests.

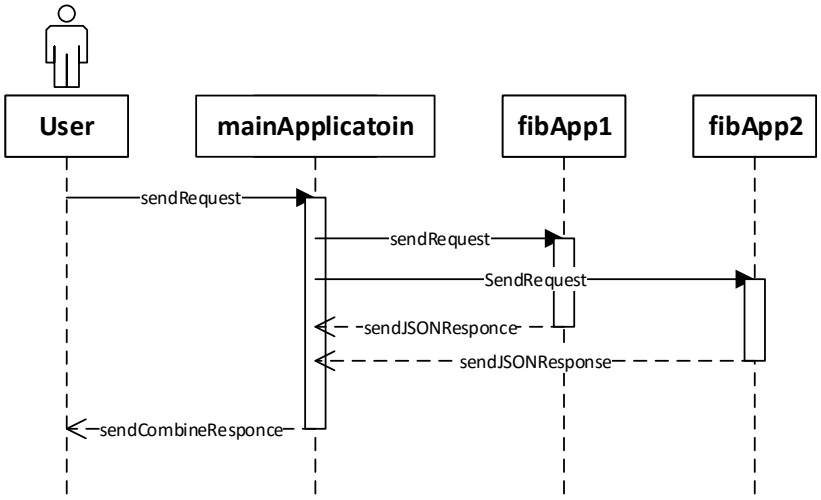

Figure 14. Sequence diagram for Parallel workflow experiment

The result of performance testing for both types of applications can be found in Table 4 and Table 5.

EC2 instance type	Average number of request per sec	Average CPU load	Average response time in sec
M1.small	18	~80%	0.35
M3.medium	40	~82%	0.41

M3.large	84	~79%	0.46

Table 4. EC2 instances performance with custom applications type of type *mainApplication*

EC2 instance type	Average number of request per sec	Average CPU load	Average response time in sec
M1.small	3	~83%	0.49
M3.medium	7	~76%	0.46
M3.large	15	~83%	0.51

Table 5. EC2 instances performance with custom applications type of type *fibApp*

The goal of the experiment is to show deployment of a system that is designed using parallel workflow but it is also important to test how deployment and auto scaling affect the performance of the system. In order to perform such tests application in the system should perform real tasks, to load CPU and memory. That is why *fibApp* is fairly CPU demanding task of calculation Fibonacci number. As a response both applications answer with simple JSON file that contains the result of calculation, IP address of VM that preformed calculation and timestamp. An example of the response from *fibApp*.


```
{
  "res": 5702887,
  "ip": "10.30.176.166",
  "time": "2015-06-24T22:27:03.482Z"
}
```


To avoid the possibility of caching result and skewing results of the test we added a random parameter for both calculations that is taken from the predefined range. Since the range is not too wide results of calculations bottom limit and the top limit is negligible.

5.3.2 Experiment

Figure 15 represents initial deployment configuration. In initial state all VM connected directly and each Scalable Component contains only one VM.

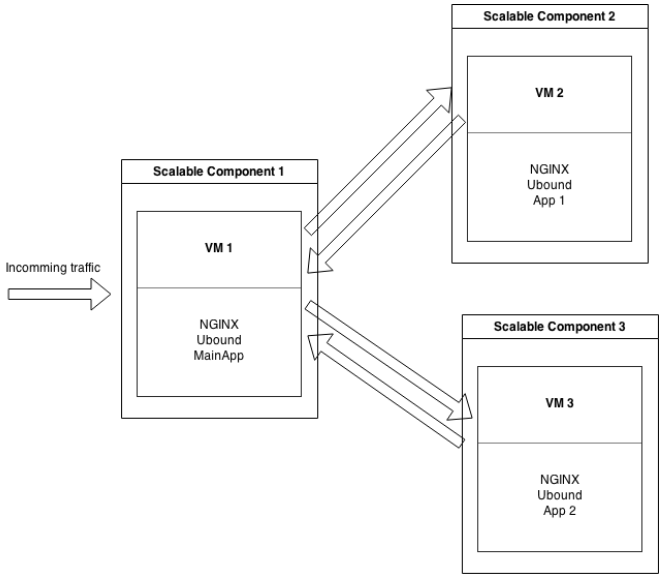

Figure 15. Initial deployment model from experiment with multiple scalable components

Scaling of this model is more complex then model from the previous experiment. First of all we need to take into account that each scalable component will be scaled differently depending on workload since initial deployment consists of multiple components Figure 16. Considering this fact it is really important to be sure that all relationships between scalable components and between VMs inside scalable components are correct during the scaling process.

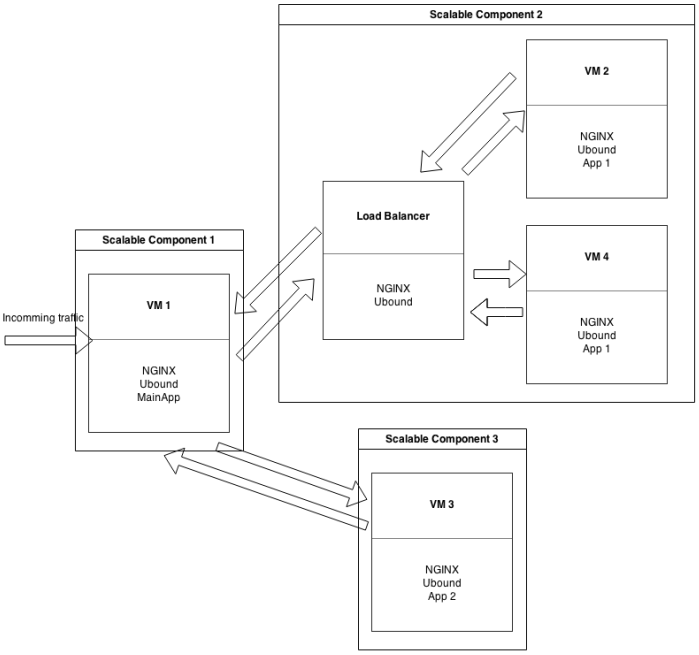

Figure 16. Scaled model with multiple scaling components

To validate results we used data from CMLDep logger module and compared it with the load on each scalable component. For a first component with installed *mainApplication* results can be found in Figure 17 and Figure 18.

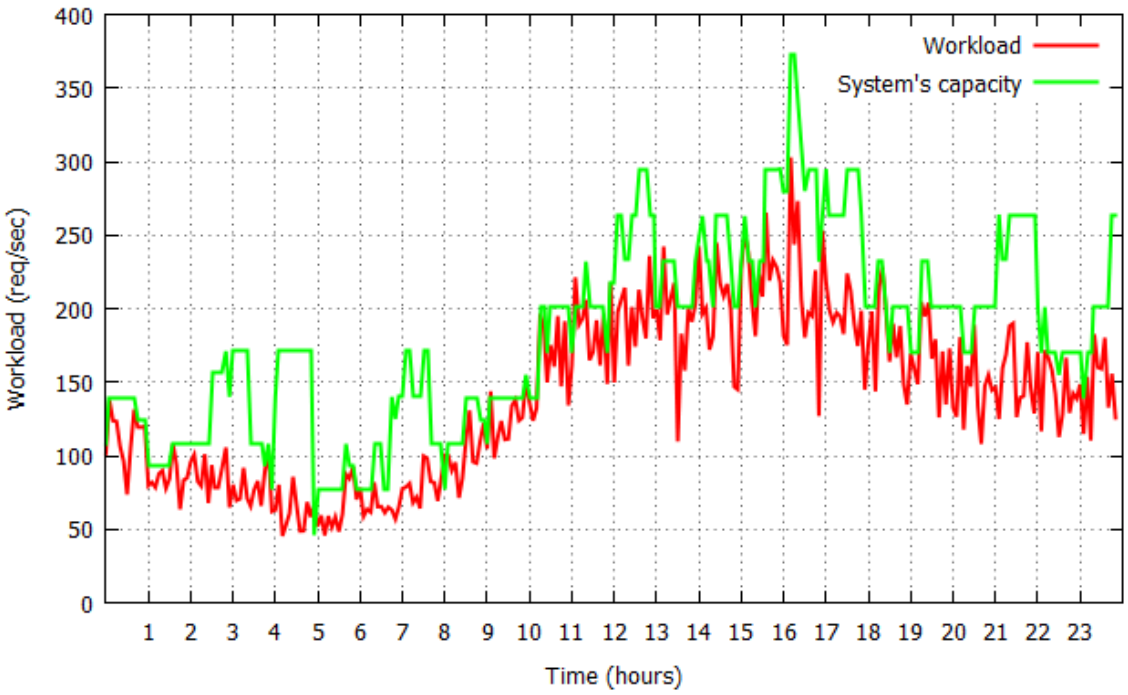

Figure 17. Workload and system capacity during parallel workflow experiment first component

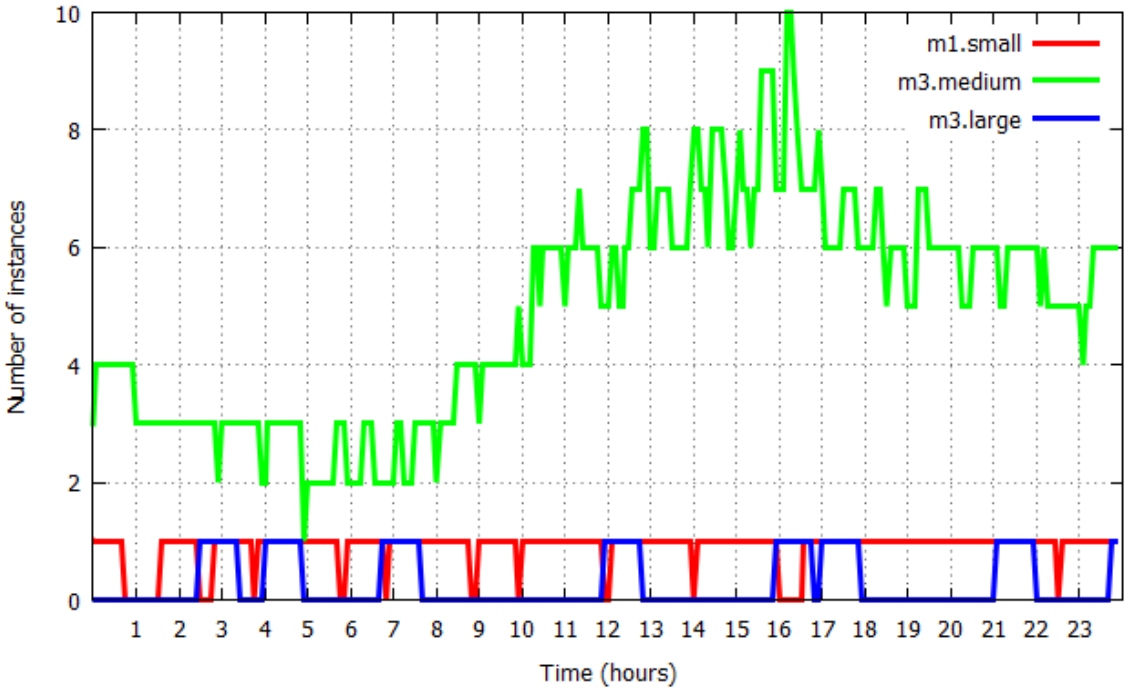

Figure 18. Number of instances of each time during parallel workflow experiment first component

For second and third components with installed *fibApp* can be found in Figure 19, Figure 20 and Figure 21, Figure 22 respectively. Since both instances of *fibApp* receives exactly same workload results of the experiment are almost completely the same.

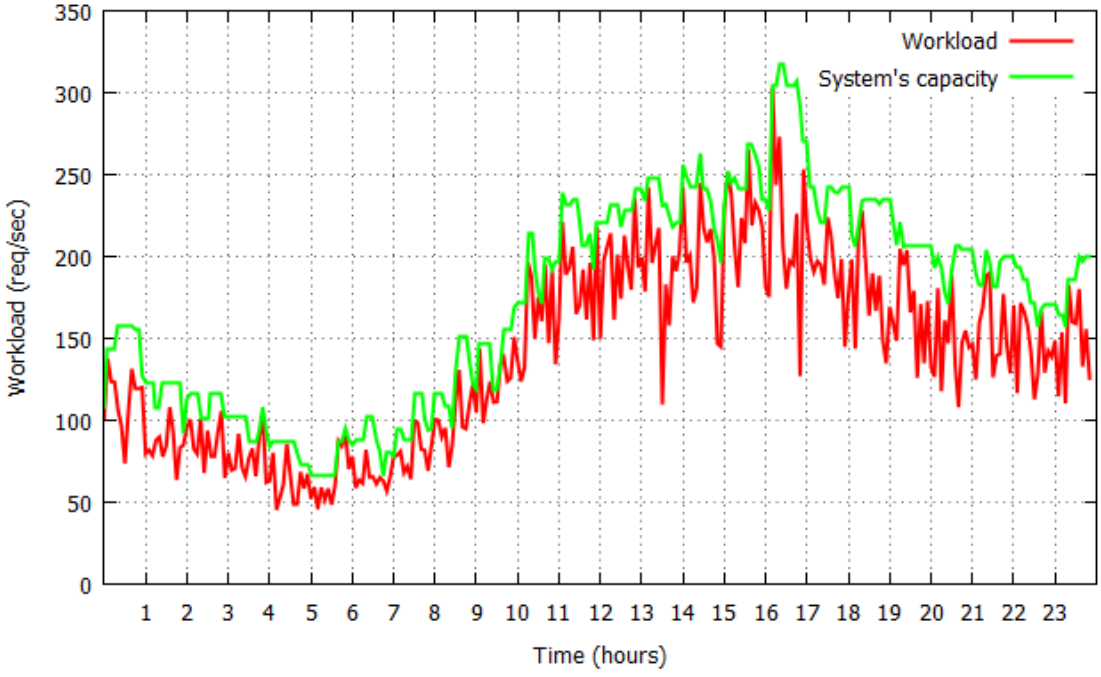

Figure 19. Workload and system capacity during parallel workflow experiment second component

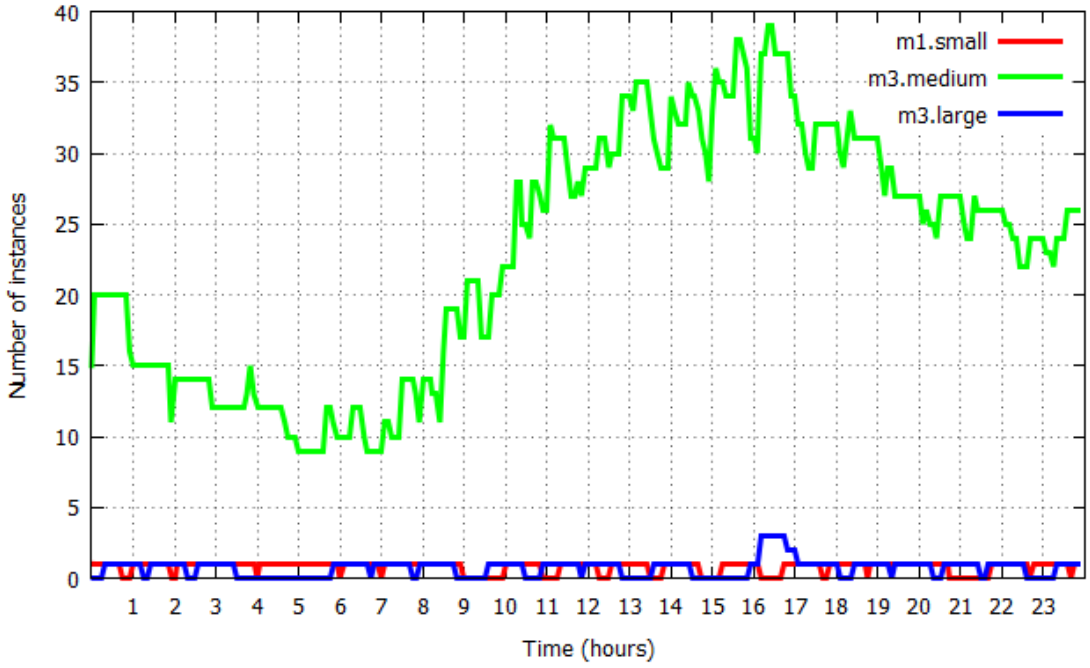

Figure 20. Number of instances of each time during parallel workflow experiment second component

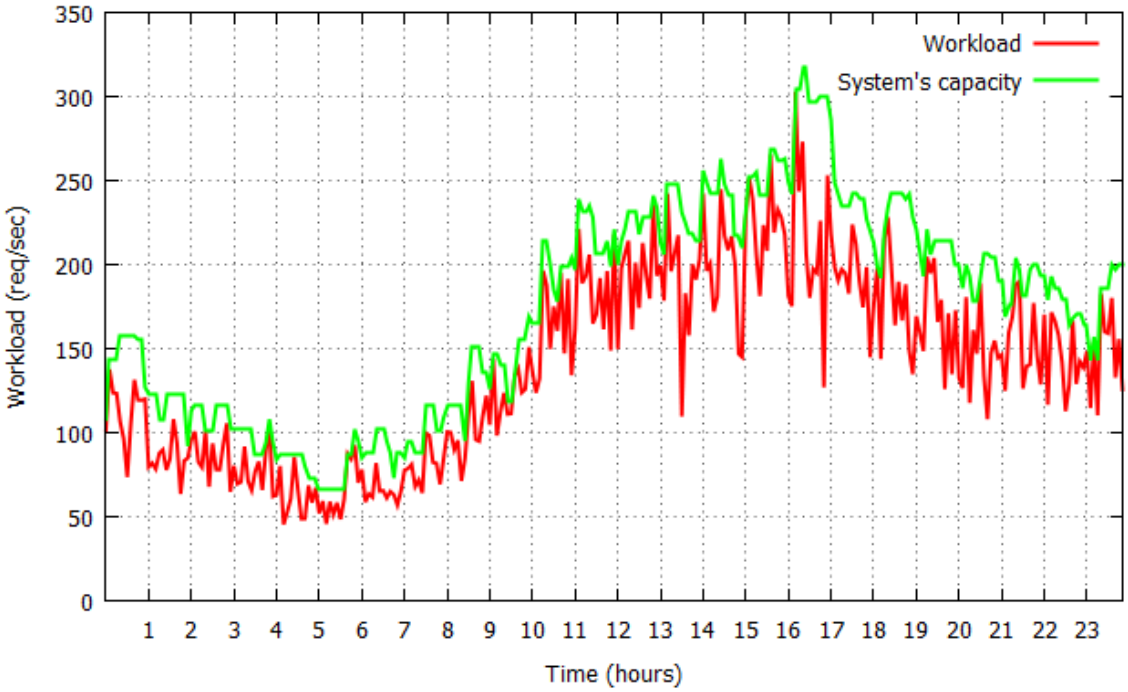

Figure 21. Workload and system capacity during parallel workflow experiment third component

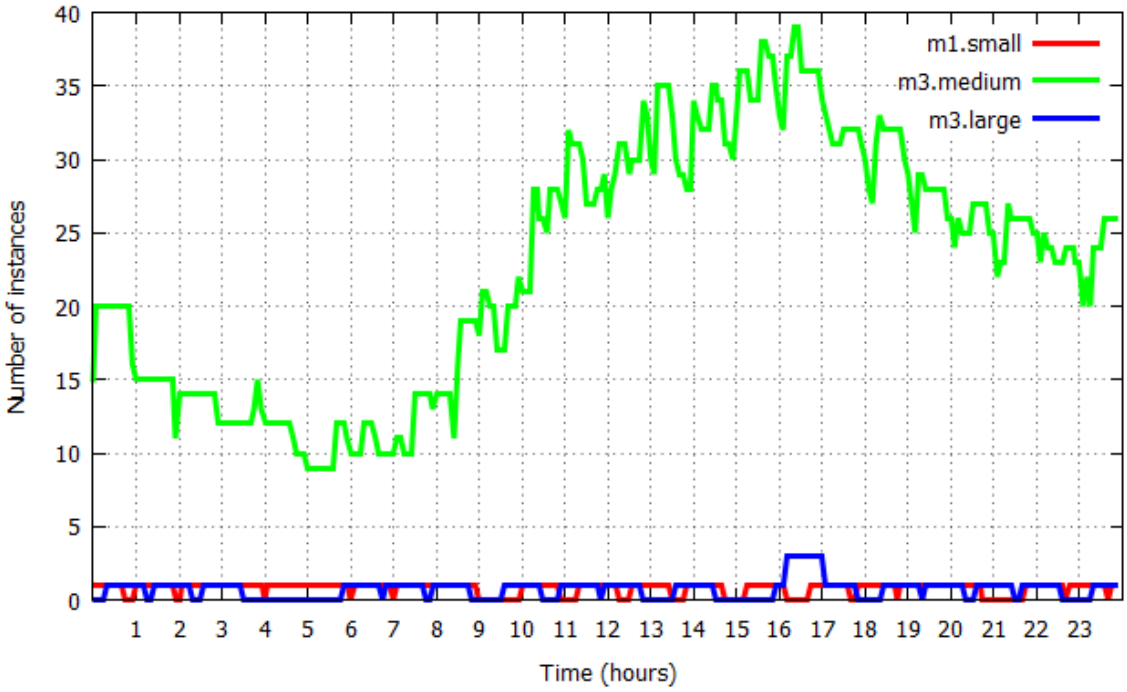

Figure 22. Number of instances of each time during parallel workflow experiment third component

As we can see from the results the system was able to provide needed performance power in 96% for the first component and 98% for the second and third experiment. As well we calculated how much slower system has become similarly like in previous experiment. Average response time for M1.small instance in the initial state was ~ 0.49 seconds and after experiment average response time increased to ~0.54 seconds, which is 10% slower. Additional information from this experiment can be found in Table 6.

Total number of requests	40126
Requests lost	2426
Successful requests	~ 94.9 %

Table 6. Prallel workflow experiment results

5.4 Exclusive workflow

For this experiment we will use exclusive workflow described by Figure 23. In this type of workflow will be executed one of two tasks either Task 2 or Task 3. We used the same

configuration from the previous experiment. Performance power for this experiment is the same as for pervious experiment see Table 4 and Table 5.

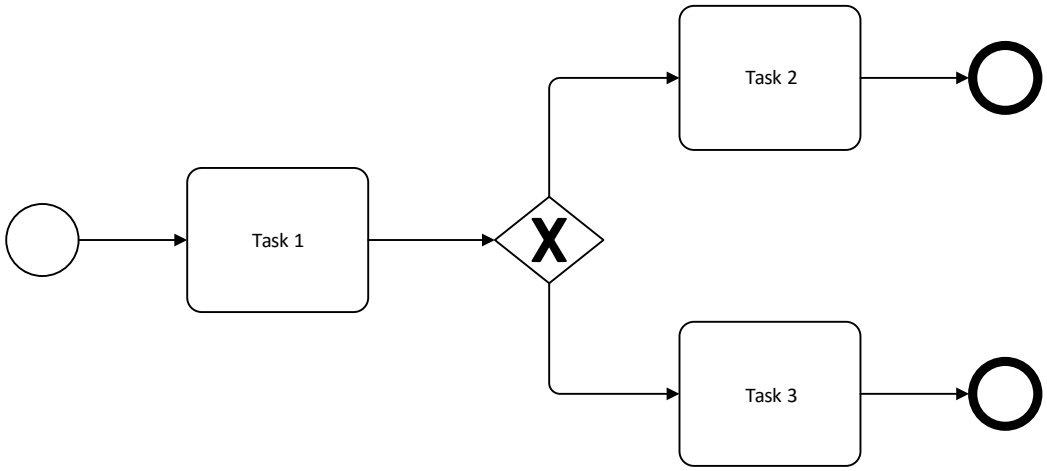

Figure 23. Example of exclusive workflow

5.4.1 Experiment configuration

Obviously to support such workflow we had to slightly modify an application that we used in the previous experiment. Application *fibApp* does not need any changes, however *main-Application* represented by Task 1 in Figure 23 has to be changed. Application should send request only to one of two applications using round robin scheme.

5.4.2 Experiment

In this experiment we pursued similar goal as in previous experiment to show that deployment of this workflow is possible. We used same tools for monitoring results of Tsung monitoring.

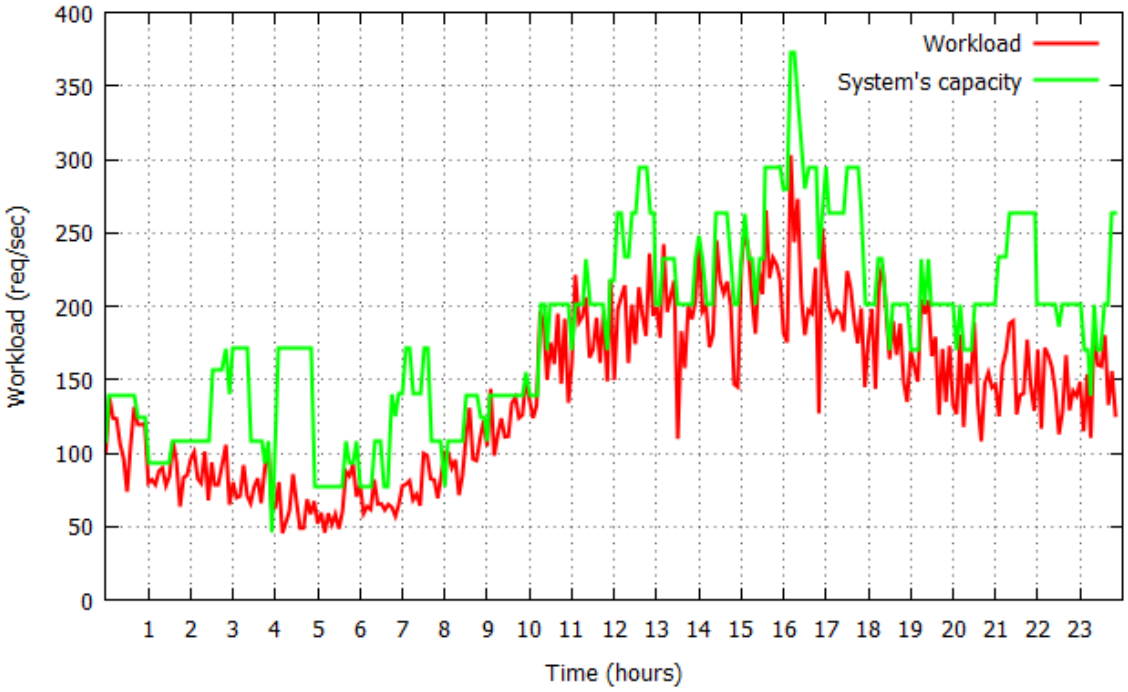

Figure 24. Workload and system capacity during parallel workflow experiment first component

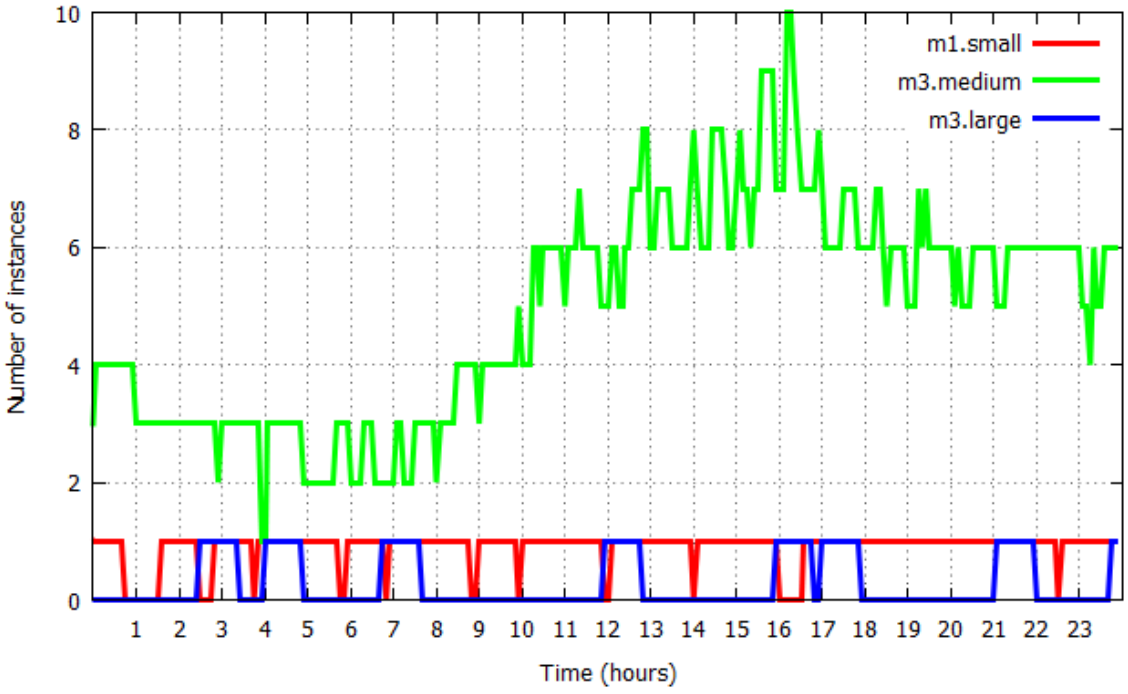

Figure 25. Number of instances of each time during exclusive workflow experiment first component

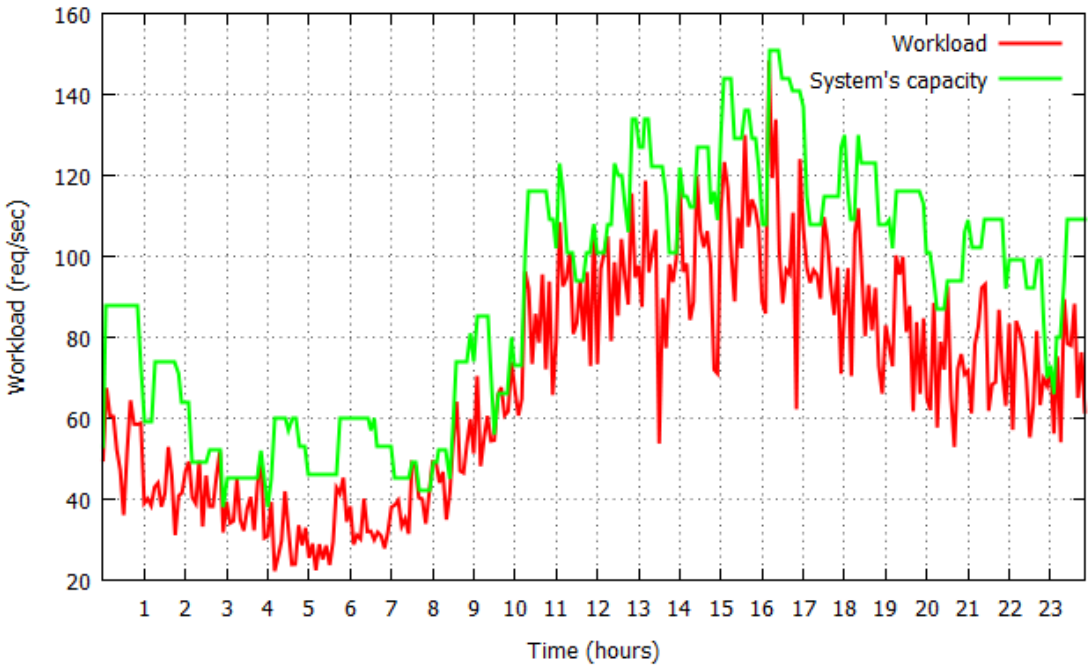

Figure 26. Workload and system capacity during parallel workflow experiment second component

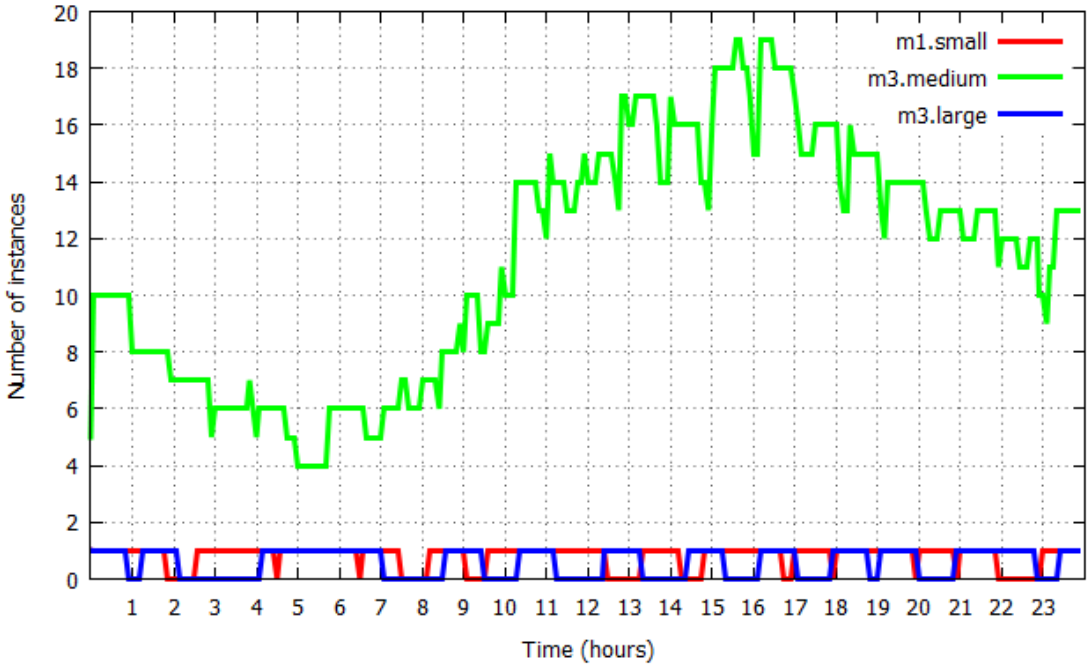

Figure 27. Number of instances of each time during exclusive workflow experiment second component

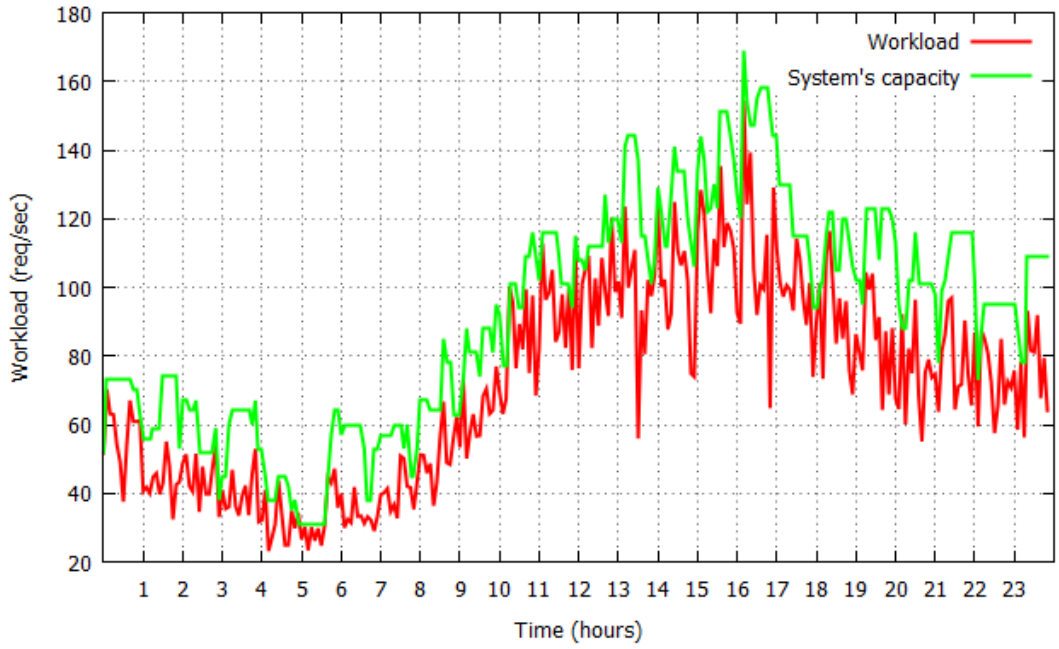

Figure 28. Workload and system capacity during parallel workflow experiment third component

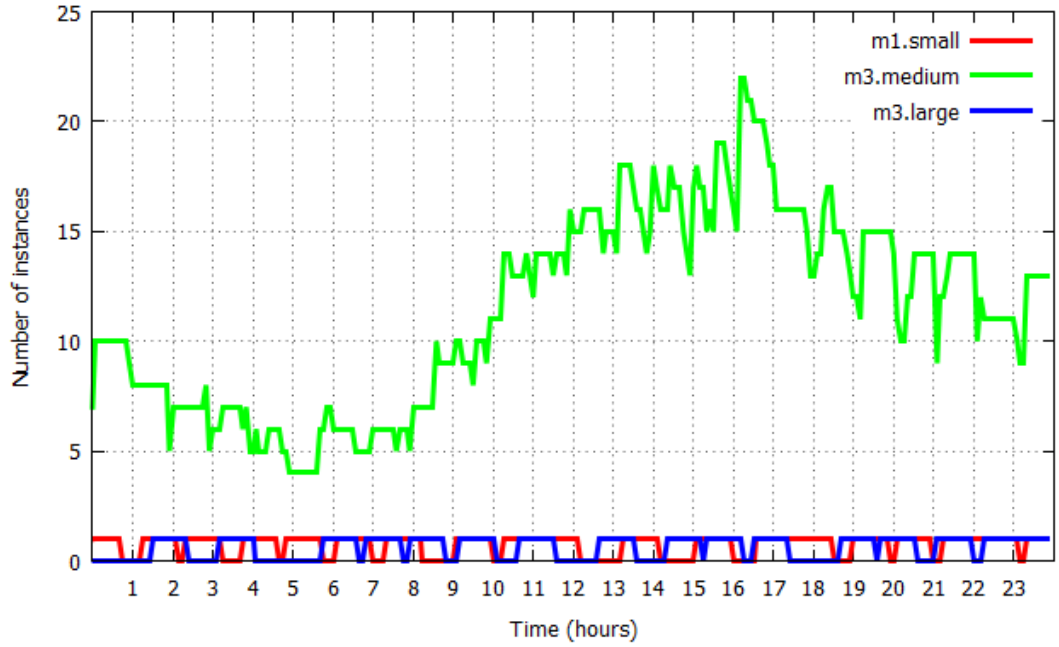

Figure 29. Number of instances of each time during exclusive workflow experiment third component

Similarly to previous experiment it is easy to notice that the system was able to support the incoming workload in 96% for the first component and about 99% for the second and third experiment. Moreover, again similar to previous experiment system become

slower with base response time for M1.small 0.49 seconds and 0.52 seconds after the experiment about 6% slower. Additional information from this experiment can be found in Table 7.

Total number of requests	40126
Requests lost	1913
Successful requests	~ 95.2 %

Table 7. Exclusive workflow experiment results

5.5 Cost overhead

One of the main reasons why cloud grows so fast is because its cost efficient and it should stay that way. The proposed solution, however, required some overhead as load balancers which are regular VMs. However, actually it is not the case for example Amazon charges $0.026 per hour for using Elastic Load Balancer which is exactly price of Amazon M1.small instance.

6 Conclusions

The thesis presented a solution for deploying dynamically scalable systems in the cloud. Moreover, the system is platform independent and can be easily redeployed to another cloud provider with almost no effort. The main reason for this is that solution is built on top of CloudML that uses jclouds, and this allows us to say that proposed system is truly platform independent. The key component added in this system was a scalable component. Scalable component makes designing deployment configuration for a complex system much easier. And what is more important each scalable component is able to scale on its own, includes own load balancer and all procedure to support scaling up and down without any changes to configuration or additional commands from the administrator. Another advantage of the solution is integrated LP model that is able to find best deployment configuration considering cost and performance. To achieve this goal and help support LP model in its full capacity we fully integrated the notion of time bags. Part of the idea behind time bad is that we should not kill instances before the end of the time that we already had paid for. This helps to keep the cost of the system low. Another important part of the developed solutions is CMLDep tool that allows to make deployment process as easy as possible.

To prove that we have achieved all the goals we conducted three experiments. In all experiments our solution has showed its ability to scale to support the incoming workload more than 90% of the time. However, unfortunately experiments showed as well that system becomes notably slower from 6% to 20% depending on the configuration. We showed that system can be used in enterprise size system that consist of many components and uses parallel or exclusive workflow.

7 Future work

Despite all described advantages proposed solution still can be improved in many ways. One of the weakest points is the monitoring system. Right now there is no way to get information about the performance of each particular VM in particular scalable components that is why round robin has been used in load balancing. Monitoring CPU load and RAM would allow us to use much more efficient algorithms inside each scalable component and would give much deeper picture about the system in overall.

Another area where improvement is needed is CMLDep tool. It has been designed as simple as possible and that is one of the main reasons why it lacks some needed functionality such as some kind of dashboard for administrating the system or at least health monitor.

Moreover, third main problem that can be fixed in the future is the process of creating deployment configuration. Despite all the effort it is still not really clear and easy and the system should lean in the future towards approaches used in Apache Brooklyn.

Bibliography

[1] J. Erbes, Nezhad Motahari, H.R. S. Graupner, The Future of Enterprise IT in the Cloud, Computer 2012 , pp 66 – 72

[2] IBM. Under cloud cover: How leaders are accelerating competitive differentiation. October 2013

[3] M.G. Avram, Advantages and Challenges of Adopting Cloud Computing from an Enterprise Perspective, 7th International Conference Interdisciplinary in Engineering, Volume 12, 2014, pp. 529–534

[4] S. N. Srirama, J. Viil: Migrating Scientific Workflows to the Cloud: Through Graph-partitioning, Scheduling and Peer-to-Peer Data Sharing, 16th IEEE International Conference on High Performance and Communications (HPCC 2014) workshops, August 20-22, 2014, pp. 1137-1144. IEEE

[5] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi,. Cloud computing — The business perspective, Decision Support Systems, vol. 51, pp. 176-189, 2011.

[6] Gonidis, F., Paraskakis, I., Simons, A.J.H.. A Development Framework Enabling the Design of Service-Based Cloud Applications. In Workshop in the Third European Conference on Service-Oriented and Cloud Computing. Manchester, UK.

[7] J. Guillen, J. Miranda, J. M. Murillo and C. Cana. Developing migratable multicloud applications based on MDE and adaptation techniques, in the 2nd Nordic Symposium on Cloud Computing & Internet Technologies, Oslo, 2013, pp. 30-37

[8] Google Cloud Deployment Manager. URL https://cloud.google.com/deployment-manager

[9] Amazon CloudFormation. URL http://aws.amazon.com/cloudformation

[10] Apache JClouds. URL https://jclouds.apache.org

[11] Apache Libcloud. URL https://libcloud.apache.org

[12] OpenStack. URL http://www.openstack.org

[13] OpenNebula. URL http://opennebula.org

[14] VMware vCloud Suite. URL http://www.vmware.com/products/vcloud-suite

[15] CloudML. URL http://cloudml.org/

[16] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “Auto-scaling techniques for elastic applications in cloud environments,” Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK- -09-12, 2012.

[17] M. Vasar, S. N. Srirama, and M. Dumas, “Framework for monitoring and testing web application scalability on the cloud,” in Nordic Symp. on Cloud Computing & Internet Technologies (NORDICLOUD). ACM, 2012, pp. 53–60.

[18] Amazon Cloud Watch. URL http://aws.amazon.com/cloudwatch/

[19] S. N. Srirama, A. Ostovar: Optimal Resource Provisioning for Scaling Enterprise Applications on the Cloud, The 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom-2014), December 15-18, 2014, pp. 262-271. IEEE.

[20] SNMP. URL https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

[21] Snmdp. URL https://wiki.archlinux.org/index.php/Snmpd

[22] Al-Haidari, F.; Sqalli, M.; Salah, K., "Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources," Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, vol.2, no., pp.256, 261, 2-5 Dec. 2013

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Iurii Tverezovskyi** (date of birth: 03.05.1990),

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

CloudML based dynamic deployment configuration for scaling enterprise applications in the cloud,

supervised by Satish Narayana Srirama, Phd,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **05.08.2015**