EMMCS: An Edge Monitoring Framework for Multi-Cloud Environments using SNMP

Saad Khoudali[1], Karim Benzidane[2], Abderrahim Sekkaki[3]
Computer Science Department, Laboratory of Research and Innovation in Computer
Hassan II University, Faculty of Sciences Ain Chock, Casablanca, Morocco

***Abstract*—Multi-cloud computing is no different than other Cloud computing (CC) models when it comes to providing users with self-services IT resources. For instance, a company can use services of one specific cloud Service Provider (CSP) for its business, as it can use more than one CSP either to get the best of each without any vendor lock-in. However, the situation is different regarding monitoring a multi-cloud environment. In fact, CSPs provide in-house monitoring tools that are natively compatible with their environment but lack support for other CSP's environments, which is problematic for any company that wants to use more than a CSP. In addition, third party cloud monitoring tools often use agents installed on each monitored virtual machine (VM) to collect monitoring data and send them to a central monitoring server that is hosted on premise or on a Cloud, which increases bottlenecks and latency while transmitting data or processing it. Therefore, this paper presents a monitoring framework for multi-cloud environments that implements edge computing and RESTFul microservices for a high efficiency monitoring and scalability. In fact, the monitoring framework "EMMCS" uses SNMP agents to collect metrics, and performs all monitoring tasks at the edge of each cloud to enhance network transmission and data processing at the central monitoring server level. The implementation of the framework is tested on different public cloud environments, namely Amazon AWS and Microsoft Azure to show the efficiency of the proposed approach.**

Keywords*—Simple network management protocol; multi-cloud monitoring; edge computing; edge monitoring; microservices; cloud computing

I. INTRODUCTION

Now-a-days, the IT world witnesses an exponential evolution since the emergence of the CC [1] [13] paradigm where hardware (such as CPUs, memory, storage, unlimited bandwidth, virtual network equipment etc.) and software (such as WEB and application servers, databases, frameworks etc.) are provided as reasonably priced and payed-per-use services compared to acquisition, on premise hosting, self-deployment and maintenance by the client. In order to satisfy all client needs and to minimize services use costs, CC made its services available through multiple levels a.k.a. *-as-a-Service (i.e. Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS)) along with many features such as global high availability, resource polling, scalability, elasticity, and on-demand services delivered in an automated manner.

With these features, CC can offer unlimited computing resources (virtually), reasonably priced and tailored services, accessible from anywhere through a WEB browser. However, the need of using more than one public Cloud became a trend if not a thing of normality. In fact, clients are more frustrated by the vendor lock in practice that most of CSPs do in a way or another. This has encouraged clients to spread their workload across different types of cloud providers, i.e. the multi-cloud strategy [2], rather than relying on one CSP, which gives clients more choices and benefits while making their businesses highly available and fault tolerant. In fact, by using another cloud as a backup site [2], it is possible to avoid downtimes during an unexpected peak of workload, or to lessen security risks by providing a higher level of resiliency against malicious attacks such as Distributed Denial of Service (DDoS).

Aside from Multi-cloud computing, another practice that is gaining popularity in the IT world is the "Edge Computing". Edge Computing [3] relies on keeping processing of client's data at the periphery of their origin, meaning as near as possible from their sources on the network and processed in a decentralized manner. In Edge Computing, a local computer or server, or even the device itself does the processing rather than sending data to the datacenter to be processed. That way, it is possible to minimize network resources utilization that can be throttled due to the massive amount of raw data that are transmitted. Furthermore, processing raw data near their sources will allow sending useful data to the datacenter and relieving it from these tasks in order to apply additional, non-extensive processes or simply to display them. In addition, Edge Computing enables real-time processing by accelerating and streaming data without latency, allowing smart applications to be more responsive by processing data at their creation, which will eliminate any lag time and thus, making time process efficient for critical applications. In addition, Edge Computing will benefit from microservices architectures [14], to allow portion of an application to be moved in order to run on the edge of a network.

Therefore, the EMMCS framework was designed and developed to monitor Multi-cloud environments using SNMP [4] [5] agents and edge computing. The framework has native support for Amazon AWS, Microsoft Azure CSPs as well as the Openstack orchestrator. At the boundaries of each cloud will reside a MP to collect monitoring data from SNMP agents and process them to generate useful metrics, to manage the deployed agents and to execute tasks on behalf of the MMS. Additionally, the architecture of the framework was designed to be microservices oriented [6], where each service provides a RESTful API from where it will be managed. Finally, the

processed data are sent back to the MMS that can run on premise or hosted on a cloud provider.

II. Related Work

Many researches were done in order to provide standardized methods or to implement frameworks that are based on standardized protocols. In [7], the authors present a reliable QoS monitoring facility called QoS MONitoring-as-a-Service (QoS-MONaaS), which approach is to monitor QoS statistics continuously at the Software-as-a-Service (SaaS) level, while enabling a secure and trusted communication channel between monitoring entities. Moreover, a modular monitoring system for private clouds called (PCMONS) [8] is developed by integrating existing techniques and monitoring frameworks, and can be integrated with existing infrastructure management tools such as Eucalyptus cloud orchestrator. However, the PCMONS framework does not provide cross service levels (IaaS, PaaS and SaaS) monitoring and multiple clouds support unlike the proposed framework in this paper.

Regarding the Multi-cloud paradigm, some researches and works have been done regarding the monitoring side but two of them are the most interesting and which approaches take the same direction of the proposed framework in this paper. In [9], the authors proposed a cross-layer monitoring framework for multi-cloud Service-based Applications (SBA). The framework has the ability to monitor multiple cloud environments at different service layers, i.e. infrastructure, platform and application, and uses Time Series Database (TSDB) to store the captured events.

The second work has been presented in the article [10]. The authors of this paper propose a novel approach for monitoring Multi-cloud environments by developing a Monitoring-as-a-Service framework called (CLAMS). The framework is composed of three major components which are the CLAMS monitoring agent, the CLAMS monitoring manager and the CLAMS SuperManager. The first component is an agent, that is deployed on the monitored Virtual Machines (VM), and which role is to gather QoS statistics related to resources and the service layer where they are running, and send them back to the CLAMS manager as requested. The second component is a manager of the deployed agents which role is to collect QoS statistics from them using PULL or PUSH methods. It stores a list of deployed agent in its database wherein the collected statistics will be organized and stored. The third and last CLAMS component, i.e. the SuperManager, is a component used only when using the Multi-cloud strategy, which role is to manage and coordinate between the deployed monitoring managers through their API and to collect QoS statistics gathered by the agents.

The proposed approach is quite interesting since it can be used to collect QoS statistics from a single Cloud or from Multi-cloud environments. The framework has been developed in JAVA, which makes it platform agnostic and can run in any environment. Yet, some downsides in this approach can be noticed. The first one is regarding the monitoring manager where it is provided with a database wherein QoS statistics are stored and organized by service level, i.e. IaaS, PaaS and SaaS, along with the list of the deployed agents, which is important in a single cloud scenario. However, the monitoring manager needs to be as lightweight as possible because in a Multi-cloud scenario, a high number of data is generated and must be processed efficiently, and it is not relevant to store QoS statistics in the CLAMS manager and in the CLAMS SuperManager databases. The second downside is that the framework focuses only on collecting QoS on demand rather than providing complete monitoring features, as a Cloud monitoring solution or a NMS should.

III. EMMCS: An Edge Monitoring Framework for Multi-Cloud Environments using SNMP

As it was introduced in the first section, this paper presents an architecture of a monitoring framework for Multi-cloud environments using SNMP protocol. The proposed framework is scalable and modular, were each module is microservices oriented. Meaning that each one is designed to be lightweight, and is provided with its own RESTful API to communicate with other services. In addition to basic management and monitoring functionalities (e.g. collecting metrics, tasks scheduling, notification and alerting, resources management, check methods definition etc.), the framework implements analytics features such as heuristic data analytics for performance and QoS trends, and behaviors analysis to detect and notify the client beforehand about a potential failure that might occur in order to take preventive measures. It can also make decisions based on preconfigured scenarios and take appropriate actions to avoid downtimes or service degradation. Additionally, the framework has the ability to monitor multiple Cloud environment through all their service levels (infrastructure, platform and application), and thus, allowing the client to have a complete view on the state of its services that are hosted on multiple Clouds.

The EMMCS's architecture, as shown in Fig. 1, offers multiple advantages over previously presented solutions, which are as follows:

- The use of the MP component in this architecture is not only to collect data from SNMP agents like what some NMS do (e.g., ZABBIX NMS uses its proxies for that particular purpose only). In fact, the MP will do all processing tasks on the edge of the cloud to generate metrics from the collected raw monitoring data, rather than sending them as they are to the MMS for processing that create latency in term of network transmission and CPU usage.
- The framework components and their services offer standardized RESTful APIs that make the integration and interfacing with other applications straightforward.
- The approach in this paper uses the standardized management protocol SNMP. The reason of this choice is that not only SNMP gives states about monitored resources, but can also give information about QoS, which is important in case the client is using applications where QoS monitoring matters.
- The microservices oriented design has the advantage of keeping the framework's development simple and easy to scale by using containers for each service, which will facilitate their deployment and orchestration.

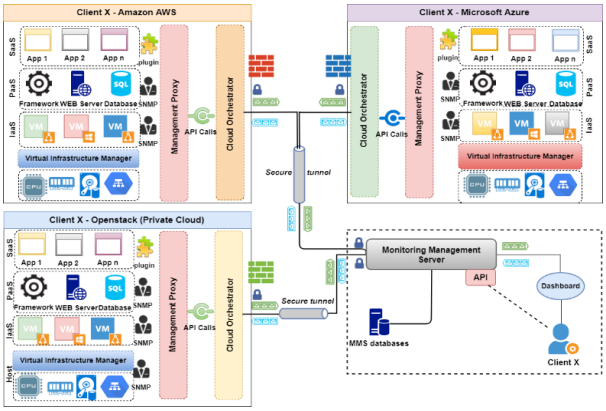


Fig. 1. EMMCS Architecture Overview.

A. Monitoring Management Server

The Monitoring Management Server (MMS) is the management component of the EMMCS framework that acts as a manager and orchestrator of deployed MPs on each cloud. It is also the framework's core where collected metrics from MPs will be consolidated in order to apply additional processing. The MMS communicates with deployed MPs through secure tunnels where the traffic is encrypted for security matter. It implements basic NMS functionalities such as resources monitoring (i.e. hosts, services, processes, hardware etc.), events management, tasks scheduling, notifications and alerts management, components configurations and much more. Additionally, the MMS offers advanced features such as data analytics to predict foreseeable issues (e.g. system or service crash, performance degradation etc.), as well as a decision-making and action service engine that executes preconfigured actions depending on scenarios to prevent breakdowns, performance or quality of service (QoS) impairments. The MMS architecture as shown in Fig. 2 counts ten microservices, which are detailed as follows:

1) Management console (MC): is the EMMCS management CLI from where the framework and its components are managed through their RESTful APIs. With the MC, the user can perform all management (configuration, deployments) and monitoring tasks (resources to monitor, checks intervals, alarm definitions and notification methods, thresholds etc.). Then, according to the roles and privileges assigned to his account, the client can access different objects and services of the MMS and MPs to perform the desired tasks, if allowed.

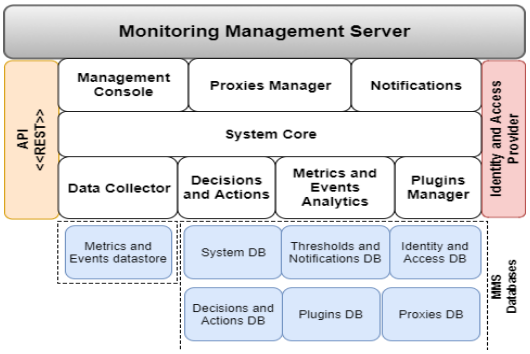


Fig. 2. Monitoring Management Server Services.

2) System core (SC): is considered as the main service in the MMS design. The SC centralizes and describes all MMS services information on a catalog where they can be requested via the SC API. In fact, this catalog is used by all MMS services to get the necessary information (IP addresses, port numbers, etc.) in order to communicate with each other. For instance, when a MMS service A wants to communicate with a MMS service B, the service A will ask the SC through its API for the service B's access information. Then, the SC will look for the requested information in its services catalog and send them back to the service A. The main benefit of this architecture is to simplify the configuration management of the MMS services by centralizing all access information in this catalog, without the need to set them at the level of each service, which can make the expansion and management of the MMS easier. In addition, the SC checks periodically the health of the EMMCS's services to ensure that they are always running, and notify the user in case a problem is detected.

3) Proxies manager (PM): is a service for remote management and deployment of MPs. In fact, this service will install an MP at the CSP when required, by uploading its package on the VM where it will be hosted and executed. The package of the MP contains its configuration that was setted up in advance through the MC, and also contains all necessary information in order to communicate with the CSP's Cloud orchestrator. To keep all information about the deployed MP (i.e. IP address, DNS name, CSP name, MP API key etc.) organized, the PM uses a JSON file as local catalog that will be updated with newly created MPs or when changes are made in some MPs configurations. The deployment also covers updating and upgrading tasks of the MP and its services. In addition, we count three types of PM remote management operations: the PM manages deployed MPs by sending requests to be executed on the monitored environment, managing MPs's services and their configurations, and collecting data from MPs. For the first type of operations, the PM will transform monitoring and action requests that need to be scheduled and executed by a MP on the monitored environment into JSON files. These JSON files are stored in the "system DB" and a copy will be sent to the corresponding MP to be executed. For the second type of operations, the MP is fully manageable and controlled by PM. In fact, the PM controls MP's services, their configurations and their updating/upgrading process. It also controls configurations of deployed SNMP agents by sending them to the MP to apply them on the agents. For the last type of PM operations, it collects from MPs updated lists of deployed VMs, their configurations and their states in order to ensure that the information about the monitored environment stored in the MMS is consistent with their actual state. Finally, the PM service manages SNMP MIBs that are used to monitor specific resources in the cloud environment. In fact, the client can, if needed, upload new MIB files that are not present by default, and translate numerical OIDs into a more "human

friendly" text to help the client distinguish between resources OIDs.

4) Data collector (DC): if the PM is the MMS's management service of MPs, the Data Collector (DC) is the service for collecting monitoring data. When an MP wants to send collected monitoring data from deployed SNMP agents, it is done through the MMS API. The MMS will receive these data through the DC service, which will immediately extract metrics and events to be processed and aggregated, and then store them as time series in the "metrics and events" datastore to be analyzed by the "Metrics and Events Analysis" service.

5) Metrics and events analysis (MEA): is a service which role is to analyze stored metrics and events in the "metrics and events" datastore in order to detect anomalies such as critical states by comparing the stored data with the monitored resource thresholds. This feature is called "instant detection". It can also prevent from abnormal behaviors of monitored resources such as flapping states, increasing memory or disk usage that can led over time to a system crash, services downtime or QoS degradation by using predictive analysis and machine learning algorithms and models. Once a critical state is detected or an event will happen, the MEA service will send notifications to the "Decisions and Actions" service, which role will be decribed. For the moment, predictive analysis is not yet implemented and only the instant detection is available.

6) Decisions and actions (DA): working jointly with the MEA service, the DA service has the main role to make decision and execute actions according to predefined scenarios that are stored in the "Decisions and actions" database. In fact, when the MEA detects that a monitored resource has reached a threshold's limit or will reach a state or a value that can cause service or system failure, the DA service will decide whether it will execute an action or not (e.g. restarting the service, notifying, scaling up/out a VM etc.). Regarding actions that are related to the Cloud orchestrator (e.g. scaling a VM by adding more hardware resources or by adding more instances, restarting a VM, migrating a VM to another CSP etc.), the DA service is provided with the "Cloud Manifest", a JSON file where information about CSPs are stored (e.g. IP addresses, API keys, credentials and secret key etc.).

7) Notifications service: is a service from which notification and alert method definitions and configurations are managed. The client can, through the MC, add, modify or delete alerts, notification methods (e-mail, SMS), notification users and groups, in addition to thresholds. This service is also used by other MMS services as a gateway to send notifications and alerts if needed, since this service provides a RESTful API that makes its use and integration straightforward.

8) Identity and access provider (IAP): is a service responsible for managing and orchestrating authentication (authN) and authorization (authZ) [11] of users and services within the framework. The IAP service is mandatory since all services must, before communicating with each other, be authenticated and granted access to do so. At each successful authentication using credentials, the IAP service will generate a token and store it in its database, then will transmit a copy of this token to the authenticated "service A" so that it will use it during the communication with a "service B" as illustrated in Fig. 3. A token provides all necessary authorizations to its related services and objects in the framework, and replaces the standard authentication method, i.e. using credentials, to increase the control and security level of communications between services. To increase the security of the authentication and authorization process, the token has a limited lifetime and will expire in order to generate a new one.

9) Plugins manager: This service allows the MMS to extend the EMMCS features via plugins.

10)MMS databases: The MMS uses two types of databases due to the nature of data that are handled, namely the MySQL RDBMS to store configurations and the MongoDB NoSQL database server to store metrics and events. The MMS counts seven databases which usage is described as follows:

a) Thresholds and notifications DB: is a MySQL database where notification, alert and threshold configurations will be stored.

b) Decisions and actions DB: is a MySQL database where decisions and actions that are configured through the MC are stored.

c) Identity and access DB: is a MySQL database owned by IAP service where services identity and authorization information are stored, i.e. generated tokens, credentials etc.

d) System DB: is the central MySQL database where EMMCS services configurations are stored.

e) Plugins DB: is a MySQL database where information about installed plugins are stored along with their configurations.

f) Proxies DB: is a MySQL database used by the PM service to store all information about deployed MPs and their configuration.

g) Metrics and events datastore: is a MongoDB database where collected monitoring data, i.e. metrics and events, will be stored.

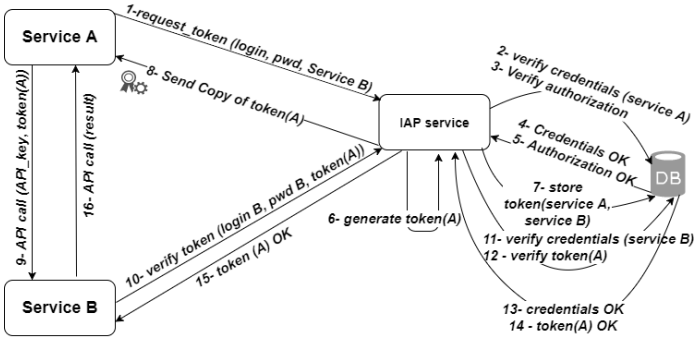


Fig. 3. Example of Communication between Two Services with Token-Based Authentication.

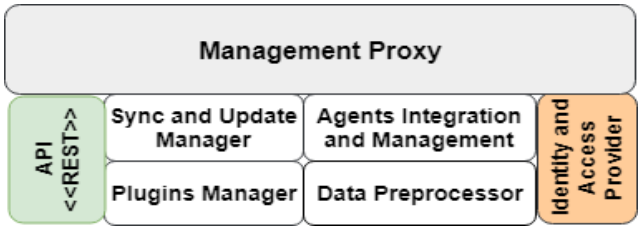


Fig. 4. Management Proxy Services Overview.

B. Management Proxy

The Management Proxy (MP) is a component of utmost importance in the EMMCS's architecture. Its primary role is to act as a manager for SNMP agents installed in monitored VMs that run on the Cloud and to execute requests on behalf of the MMS such as monitoring and action requests. Since the MP will run at the periphery (at the edge) of the cloud, its architecture was designed to keep the system lightweight with minimum resources footprint. Regarding the MP's architecture, it consists of five services (Fig. 4) that are described as follows:

1) Sync and updates manager (SUM): is the service that allows the MMS to manage deployed MPs remotely as well as their services' configurations. It is the access point to the MP's services and from where monitoring requests, configurations and updates that are configured in the MMS will be distributed. In fact, when the client defines requests that must be deployed and executed on targeted VMs or on MP services, the MMS will first transform these requests into JSON files (Fig. 5) and send them to the MP through the SUM service's API. Then, the latter will analyze these JSON files in order to transmit them to the target, i.e. to MP's services or to SNMP agents. Moreover, the SUM service runs synchronization tasks with cloud Orchestrators through their APIs to gather information related to the monitored environment such as VMs properties (name, IP addresses etc.), VMs states (running, halted, newly created, running services etc.), or to execute actions such as VM-related operation (e.g. shutdown, restart etc.), resources scaling, inter-clouds migration etc. To this end, the SUM service will use the provided credentials and the CSP's API keys to connect to the CSP orchestrator in order to execute these tasks. Currently, the SUM service supports natively the AWS and Microsoft Azure cloud providers in addition to the Openstack Orchestrator.

2) Agents integration and management (AIM): This service is of great importance in the MP's architecture because it performs all monitoring tasks. Indeed, the AIM service acts as a manager of all deployed SNMP agents by keeping a list of the deployed ones, managing their configurations, executing SNMP requests and actions on the targets sent by the MMS through the SUM service, collecting monitoring data and capturing events generated by SNMP agents installed in the VMs. As mentioned before, monitoring checks and actions data that will be executed on the targets are scheduled in the MMS by the client and then are sent to concerned MPs through their SUM services which will, in turn, analyze the received files to determine their categories (monitoring, configuration update, action or service update). The structure of JSON files (Fig. 5) that are sent by the MMS to the MP is organized as follows:

For monitoring requests, the JSON file will contain the following information:

- *Id:* Request identifier;
- *Type:* Request type (0 = monitoring, 1 = action, 2 = configuration update, 3 = component update);
    - *Target:* Information about the monitored VM, namely:
    - *uuid:* The Unique Universal Identifier used by the Cloud orchestrator to identify the VM;
    - *Hostname:* The VM hostname;
    - *ipAddress:* The VM IP address;
    - *SNMPversion:* The SNMP protocol version that will be used to query the SNMP agent in the VM;
    - *SNMPcommunity:* The SNMP community string used to query the SNMP agent in the VM.

Fig. 5. Example of a Request File Generated by the MMS.

- Check: definition of the check to schedule, namely:
    - *uuid:* Unique Universal Identifier of this check. This identifier is generated by the MMS at the time of its creation to identify and match the request with its response;
    - *Name:* a symbolic name of the monitoring check;
    - *ObjectID:* OID of the resource that will be monitored within the VM;
    - *CheckPeriod:* is the period during which the monitoring request will be executed, e.g. 24x7;
    - *NormalCheckInterval:* Normal check cycle in seconds representing how often the check will be executed when the last check status is OK;
    - *AbnormalCheckInterval:* check cycles in seconds representing how often the check will be executed when the last returned state is abnormal;
    - *MaxAttempts:* number of check attempts to perform during AbnormalCheckInterval cycles before reporting that the resource is in an abnormal state (all check attempts status need to be NOK);
    - *Priority:* The priority level of the check. This is useful when monitoring critical resources. Two levels are available: normal or high;
    - *PerformInitialCheck:* is a Boolean that is used to tell the AIM service whether an initial check on the target will be executed or not.


```
{
  "requests": [
    {
      "id": 50,
      "type":1,
      "target": {
            "uuid": "2274bdce-3eed-49c3-9794-f14d07fc2eba",
            "hostname":"test-vm-01",
            "ipAddress": "10.10.3.12"
          },
      "action": {
            "method":"ssh",
            "sshPortNumber":1322,
            "command":"restartMySQLService.sh",
            "user":"snmp",
            "uuid":"1f1fcbbd-9121-448f-8c16-07fbed4e6eda",
            "name": "restartMySQLService"
          }
    }
  ]
}
```


Fig. 6. Example of an Action File Generated by the MMS.

For actions that will be executed on the target, the JSON file contains information (Fig. 6) that is described as follows:

- *Id:* The request identifier;
- *Type:* Request category type (0 = monitoring, 1 = action, 2 = configuration update, 3 = component update);
- *Target:* Information about the monitored VM, namely:
    - uuid: The Unique Universal Identifier used by the Cloud orchestrator to identify the VM;
    - Hostname: The VM hostname;
    - ipAddress: The VM IP address;
- *Action:* definition of the action to execute on the targeted VM, namely:
    - *Method:* the protocol through which the action will be executed. There are two methods: using the SSH protocol for Linux-based VMs and using the SMB/RPC protocols for Microsoft Windows VMs. In both methods, the actions will be transmitted and executed in the VMs environments;
    - *Command:* The command or script to run on the targeted VM;
    - *sshPortNumber:* Port number of the SSH server that runs on the VM. This key is used jointly with the "method" key (for the SMB/RPC, it will use the default port number);
    - *User:* the username of the account that has enough privileges to execute scripts on the targeted VM;
    - *Name:* a symbolic name of the action;
    - *uuid:* Universal Unique ID of this action. It is generated by the MMS at the time of its creation.

In addition to monitoring tasks, the AIM can remotely install and configure SNMP agents on VMs that need to be monitored. For that, the AIM service will connect to the VM via the SSH or SMB/RPC protocols to install, enable and configure the SNMP service with appropriate configurations.

3) Data preprocessor (DP): is a service that performs processing tasks between the AIM and the MMS. Indeed, the collected monitoring data or events from SNMP agents may contain raw data (Fig. 7) that need to be cleaned and standardized in order to extract and generate the useful metrics.


```
[root@localhost ~]# snmpget -v2c -c public 127.0.0.1 .1.3.6.1.4.1.2021.4.6.0
UCD-SNMP-MIB::memAvailReal.0 = INTEGER: 7621108 kB
```


Fig. 7. Example of an SNMP Response.

```
{
  "responses": [
    {
      "requestId": 28,
      "timeStampGen": 20180812110636,
      "checkUUID": "591d3be9-3a73-4db4-ac50-461987c1085d",
      "worker": {
            "uuid": "61d0d7be-73a1-48fb-be18-b6cac61eb6ef",
            "name": "mp.awslab.edu",
            "location": "South America",
            "provider": "amazon"
        },
      "target": {
            "uuid": "e13f41d9-3f28-449f-92e4-e92556cc4064",
            "ipAddress": "172.16.10.12",
            "hostname": "test-vm-03"
        },
      "ressource": {
            "objectID": ".1.3.6.1.4.1.2021.4.4",
            "value": 8257532,
            "unit": "kB"
        }
    }
]
}
```


Fig. 8. Example of a Response File Generated by the DP.

Moreover, in order to optimize the overall monitoring performance of the framework, the DP will process the monitoring data in real time, convert them into JSON format (Fig. 8) and stream the response file back to the MMS. To manage failures or errors during data transmission, the DP will cache these files in its memory during their transmission until they are successfully received by the MMS. The JSON format to represent and describe these data was chosen since EMMCS's services use RESTful API calls to communicate with each other, and because the JSON format is lighter [12] in processing and transmission than other data exchange formats (e.g. XML), which will have minimal impact on CPU, memory and I / O utilization.

The response file generated by the DP has a structure that reflects the schema of the database where the metrics and their related data will be stored in the MMS. The non-exhaustive list of attributes that are used in the response file are described as follows:

- *RequestId:* The identifier of the request file that was previously sent by the MMS to schedule the monitoring check;
- *TimeStampGen:* The timestamp of the response (when the check was executed);
- *CheckUUID:* The Universal Unique Identifier of the check related to the request "requestId";
- *Worker:* The MP that generated this file as well as its related information, namely:
    - *uuid:* The Universal Unique Identifier of the MP that generated this file;
    - *name:* The Fully Qualified Domain Name of the MP;
    - *location:* the region of the CSP Datacenter where the MP is hosted;
    - *provider:* The name of the CSP.
- *Target:* Information on the VM that is monitored, namely the UUID, ipAddress and hostname;
- *Resource:* the information on the VM resource that is monitored, namely:
    - *ObjectID:* the OID of the resource that is monitored within the VM;
    - *Value:* the metric that will be extracted and sent to the MMS;
    - *Unit:* the unit used to measure the OID value. Depending on the resource, it can be a percentage (e.g. CPU usage), kilo Bytes or can be omitted in case of a string.

4) Plugins manager: like the MMS, this service allows the management of plugins to extend MP's functionalities. For example, adding support for Cloud and IaaS orchestrators, or monitoring SaaS applications if they offer interfaces from where access to metrics is possible.

5) Identity and access provider (IAP): Operates the same way as in the MMS.

However, unlike the MMS, the MP will not use any type of database to store its data. In fact, since the EMMCS configurations (services included) are stored in the MMS databases, the MP services will receive their configurations from the MMS as JSON files that will be stored locally.

IV. EMMCS Implementation

In this section, an implementation of the proof of concept (PoC) of the EMMCS framework will be presented. The development of the framework's components, i.e. the MP and the MMS, was done using Python3.5 as a primary programming language, which is known for of its multiplatform compatibility, provides a large number of libraries and modules that can simplify applications development, and for its easy syntax that make codes maintainable and readable. Details about the PoC of the MMS and the MP, their requirements and the testbed environment where the framework was implemented will be described further in this article.

A. MMS Proof of Concept

The MMS has been developed in Python3.5 using multiple libraries and SDKs to implement its features, namely RESTful API with the "Django REST" framework, CSP management with "BOTO3" for Amazon Web Services and "Azure SDK for Python" SDKs, and the "PySNMP" library for SNMP implementation. As for data storage, the MMS relies on the documents-oriented database MongoDB where metrics and events are stored in JSON format, and MySQL to store service configurations.

The setup procedure of the framework starts with the deployment of the MMS that needs its requirements to be installed first. The MMS comes in a "tar.gz" archive that is extracted using the "tar" command in a terminal. The extracted package contains a Shell script named "deploy-mms.sh" and a directory that includes the MMS binaries and configuration files. The "deploy-mms.sh" script was developed to automate

the MMS deployment process by checking the system requirements and dependencies and thus, needs to be executed as "root" or any user that has enough privileges, and then install what is missing. Since the script is interactive, it will ask the user for some information that will be needed by the MMS such as the path of "Python3.5", installation path, IP address/port number of the MongoDB and MySQL servers, the name of databases to create, the port number that will be used by the MMS etc. After that, the script will copy the MMS directory and its content to the specified location and finally starts the MMS services.

B. MP Proof of Concept

To implement its features, the MP's development used the PySNMP library that provides all necessary packages that implement SNMP functionalities. Like the MMS, the MP implements BOTO3 for AWS and the Azure SDK for Python to communicate with them through their APIs to stay synchronized. Concerning the MP's API, it was developed with the "Falcon" framework because it allows developing high performance and lightweight RESTful API using Python. It should be noted that since the MP needs to be highly efficient in term of resources usage since the standard Python implementation is not resource-friendly compared to other programming languages such as JAVA, C++ etc., the "PyPy" alternative was adopted to implement the MP. The main difference between the standard Python and PyPy is that the latter integrates a compiler named Just-in-Time compiler (JIT) that compiles Python code into low-level code, which implies less resources usage and high performance execution. In addition, and in order to install the agents on Windows OS, the MP uses PyPsExec library that provides methods to execute remote commands on a Windows OS through the SMB/RPC protocol.

For the MP's deployment procedure, it is done from the MC. It comes as a self-extracting archive that includes scripts and configuration files. The archive is generated using the "makeself" tool. Then, the client will execute the deployment procedure and will provide all necessary information needed by the MMS to install and configure the MP.

In addition, it is important to provide information about the client's CSP subscription (credentials, access keys and the CSP API key) otherwise the deployment will fail. The MMS will then open a SSH tunnel with the remote VM, copy the MP's archive to /tmp location and execute the self-extracting archive to extract its content. Like the MMP, the MP's archive includes a Shell script named "deploy-mp.sh" that will check and install missing requirement and dependencies in the system. Once done, the script will configure the MP environment and then starts its services.

```
[root@localhost ~]# mms-mgmt-cli.py set-request mms-hostname=192.168.1.10 mms-port=20200 request-type=1 \
mp=mp.locallab.edu target=test-vm-01 action=$SCRIPTS_PATH/restartMySQLService.sh action-method=ssh \
user=snmp
***Please enter your credentials***
username: Admin
password:
sending request data to MMS@192.168.1.10, port:20200, transfer_proto:ssh
response from MMS@192.168.1.10: HTTP status=200
done.
```

Fig. 9. Example of an Action Request to be Executed on a VM.

C. Testbed Environment

The implementation of the test environment to validate EMMCS's features took place in three stages: first, preparing the environment and prerequisites, then testing the framework features and finally running performance tests.

1) Stage 1: setting up requirements: The first stage consists of setting up the infrastructure to install the framework components, i.e. installing the MMS, creating the VMs to be monitored and the VMs where the MP will be installed at the CSPs, as well as preparing the execution environment and software prerequisites for the framework's components. The VMs to be monitored were deployed and configured as shown in Table 1.

On the other hand, the framework components are deployed as follows:

- The MMS is installed on an Ubuntu 16.04 LTS server VM with an Intel Core i5 6th Gen CPU, 4 GB of RAM and 500 TB of internal storage and was hosted on premise;
- The MP is deployed on a VM at each Cloud environment as described in Table 1 where it will manage and monitor targeted VMs and their resources. The MPs are executed on an Ubuntu 16.04 LTS server instances with two vCPUs, 4 GB of RAM and 40 GB of internal storage with the Secure Shell (SSH) protocol enabled for remote deployment and execution;
- The SNMP protocol version used in this test is "2c" (the SNMP version 3 is also supported), using a private community string "cloudlab".

It is important to remind that the framework has the capability not only to monitor Multi-cloud environments, but also to monitor hybrid Clouds. For that, the Openstack orchestrator was added to the testing lab that is deployed using the All-in-One (AIO) installer from the Red Hat Distribution of Openstack project (RDO project) in a physical server with an Intel Xeon E3-1240v6 CPU, 16 GB of RAM and 1 TB of internal storage. The use of an AIO installation is to simplify the deployment of Openstack since its main use is for testing purpose.

TABLE I. MONITORED ENVIRONMENT FOR THE FIRST STAGE OF THE EXPERIMENTATION

Targets	IP address	Guest OS	Instance type	Provider	Resources to monitor
Test-vm-01	10.10.3.12/24	Ubuntu Server 18.04 LTS (HVM)	2 vCPUs, 8 GB of RAM, 20 GB storage	**Openstack Queens**	**Processes:** Apache (httpd), mysqld; **Resources:** available disk space on /;
Test-vm-02	10.10.10.12/24	Windows Server 2016	2 vCPUs, 4 GB of RAM, 40 GB of storage	**Microsoft Azure**	**Processes:** lsass.exe, mysqld.exe
Test-vm-03	172.16.10.12/24	Red Hat Enterprise Linux 7.5 (HVM)	1 vCPU, 1 GB of RAM, 20 GB storage	**Amazon Web Services (AWS)**	**Resources:** VM CPU Load (15 minutes); available SWAP memory

2) Stage 2: features validation: Whilst the first stage was about to set up the testing lab's infrastructure and requirements, the second stage of the experimentation is to test the framework's features, i.e. monitoring and actions requests. Since this stage is to validate and to prove that these features are working, the test starts with one VM per CSP and some resources to monitor as described in Table 1. Then the number of monitored VMs will be increased to test the overall performance of the framework in the last stage of the experimentation.

The experimentation at the second stage consists of executing two types of requests: monitoring request and action request. To execute management tasks, a command line script in Python that implements management and monitoring tasks for the framework was developed. This script can be locally (in the MMS) or remotely executed (the MMS's IP address must be provided in the script's parameters) and acts as a RESTful client to communicate with the MMS through its API. An example of the execution of the command-line script "mms-mgmt-cli.py" as shown in Fig. 9 represents the execution of an action on the VM "test-vm-01" that runs on the Openstack orchestrator, where the script will receive a list of parameters (the parameters list is non-exhaustive) such as:

- *Set-request:* this parameter is used for sending requests to the MMS that need to be executed by the MMS itself or by the MP. For pulling data from the MMS, e.g. metrics, the parameter "get-request" will be used with other parameters;
- *mms-hostname:* can be either the Fully Qualified Domain Name (FQDN) or the IP address of the MMS;
- *mms-port:* the port number where the MMS is listening for incoming API calls;
- *request-type:* can take one of the following values: 0 (monitoring request), 1 (action request), 2 (configuration update request) or 3 (component update request);
- *mp:* the FQDN of the MP that the request will be sent to;
- *target:* the IP address or hostname of the targeted VM;
- *action-method:* the protocol to use by the MP for actions execution (i.e. SSH for Linux-based VM or SMB/RPC for Windows VMs);
- *oid:* SNMP Object ID of the resource to query.

In the example as shown in Fig. 10, these parameters will be sent to the MMS through its API that will extract them to generate the JSON file (Fig. 7) that will be sent to the targeted MP to execute the action. If a path of an executable or a script is provided in the "action" parameter, the mms-mgmt-cli.py will send the action script to the MMS, which in its turn will send it with the generated JSON file to the MP.

By analyzing the AIM log file (Fig. 10), the action request that was sent to the MP (mp.locallab.edu on Openstack) to be executed on the VM test-vm-01 was received first by the SUM service which in its turn has send it to the AIM service (all through the DP service) that executed the action successfully on the target VM (return_code=0).

On the other hand, the example as shown in Fig. 11 represents another use of the command-line script where it is possible to request for metrics of a specific resource. In this example, a request was sent to get the available memory on the SWAP partition for the targeted VM, i.e. test-vm-03@172.16.10.12 that runs on the AWS cloud provider.

Finally, to get a near real-time (the difference between near real-time and real-time can be affected by connectivity conditions) display of monitoring data, the script can be executed in "daemon mode" where it will collect from the MMS's metrics and events database any newly stored metrics and display them with their related information, as shown in Fig. 12.

```
[INFO][11:05:25] Starting realtime logging at localhost
[INFO][11:05:25] listening for requests on 10100 port...
[INFO][11:05:33] incoming connection from 192.168.1.110@sync-srv...
[INFO][11:05:33] getting request from 192.168.1.110@sync-srv
[INFO][11:05:33] request received... id:50,target:test-vm-01,action:restartMySQLService.sh,method:ssh,user:ssh,ssh_port:1322
[INFO][11:05:33] storing attached script in /tmp/.mp/scripts/restartMySQLService.sh
[INFO][11:05:33] opening SSH tunnel on test-vm-01:1322, authentication using user:snmp auth_method:password-less
[INFO][11:05:34] connection established
[INFO][11:05:34] copying attached script from /tmp/.mp/scripts to snmp@test-vm-01:/tmp/.mp/scripts/restartMySQLService.sh
[INFO][11:05:35] executing script on test-vm-01
[INFO][11:05:36] return_code=0
[INFO][11:05:38] done.
```

Fig. 10. AIM Log Snapshot.

```
[root@localhost ~]# mms-mgmt-cli.py get-request mms-hostname=192.168.1.10 mm-port=20200 request-type=0\
mp=mpamzn.cloudlab.edu target=test-vm-03 oid=.1.3.6.1.4.1.2021.4.4
***Please enter your credentials***
username: Admin
password:
sending request data to MMS@192.168.1.10, port:20200
response from MMS@192.168.1.10: HTTP status=200
waiting for data...
received data: target= test-vm-03, oid=.1.3.6.1.4.1.2021.4.4, value=8257532, unit=kb
done.
```

Fig. 11. Example of a Monitoring Request to be executed on a VM.

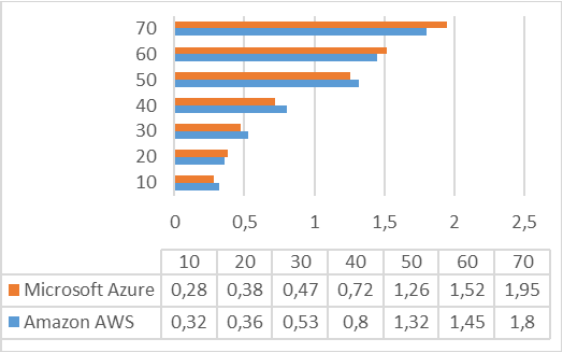
```
[root@localhost ~]# mms-mgmt-cli.py daemon-mode mms-hostname=192.168.1.10 mms-port=20200
[INFO] starting the script in daemon mode.
***Please enter your credentials***
username: Admin
password:
=====> starting realtime monitoring <=====

=====> waiting for new entries in the metrics and events database...
[INFO] new entry detected! fetching data...
[WARN] event detected!
[WARN] worker:mp.locallab.edu,provider:openstack,target:test-vm-01,resource:mysqld,event_type:restart
[INFO] done

=====> waiting for new entries in the metrics and events database...
[INFO] new entry detected! fetching data...
[INFO] worker:mp.locallab.edu,provider:openstack,target:test-vm-01,resources:{check_httpd:1,check_mysqld:1}
[INFO] resources_ok:2,resources_nok:0
[INFO] done.

=====> waiting for new entries in the metrics and events database...
[INFO] new entry detected! fetching data...
[INFO] worker:mp.awslab.edu,provider:amazon,target:test-vm-03,resources:{check_cpuLoad(15):2.15,check_memAvailSwap:8257532kB}
[INFO] resources_ok:3,resources_nok:0
[INFO] done.

=====> waiting for new entries in the metrics and events database...
[INFO] new entry detected! fetching data...
[CRITICAL] worker:mp.azurelab.edu,provider:azure,target:test-vm-03,resources:{check_Proc_lsass:1,check_Proc_mysqld:0}
[CRITICAL] resources_ok:1,resources_nok:1
[INFO] done.

=====> waiting for new entries in the metrics and events database...
```

Fig. 12. Executing "MMS-Mgmt-Cli.Py" Script in Daemon Mode.

3) Stage 3: Performance tests: The last stage of the PoC is to test the performance and resources usage of the framework's components. The validation of the PoC is not only about making sure that the framework's features are working, but also to show that the MP has a small resources footprint. Since all monitoring tasks and preprocessing are done at the edge, i.e. by the MPs, and the latter is running on the Cloud environment, in contrast to the MMS which is hosted outside the Multi-cloud environment, it is important to keep a close eye on the performance and resources usage of the MP, i.e. CPU and RAM. In order to perform these performance tests, another script was developed, named "test-mass-schedule.py" that will schedule monitoring tasks in bulk, with small check periods between each task in order to simulate high workload like in a production scenario at each public cloud.

Then the script is executed multiple times and at each time, the number of monitored VMs was increased on each cloud provider to see how their MPs' resources will be affected. In this experimentation, the test started with 10 VMs at each cloud provider and collected statistics of their corresponding MP, then increase the number of VMs up to 70. To generate statistics of the MP resources usage, "sysstat", a system performance package that comes with various tools used in Linux-based operating systems to monitor usage activity and performance was installed and executed on the MP's virtual machine. The collect of these statistics has been done after the launch of the "test-mass-schedule.py" script. The collected statistics during tests were cleaned from "idle" states of the CPU to leave only relevant values, in order to use their average. For memory usage, the highest value is selected since it will be the real value before the system cleans its memory. The results of the CPU and memory usage are shown respectively in Fig. 13 and Fig. 14.

The results in Fig. 13 show that during the scheduling of checking tasks and the processing of data sent by the SNMP agents, the MP's CPU usage on each cloud provider is minimal despite the increased number of running VMs. This show that the MP's processing is stable and efficient even if it is operating in an environment with high workloads such as Cloud environments.



	10	20	30	40	50	60	70
Microsoft Azure	0,28	0,38	0,47	0,72	1,26	1,52	1,95
Amazon AWS	0,32	0,36	0,53	0,8	1,32	1,45	1,8

Fig. 13. Management Proxy CPU usage in Percentage per CSP.

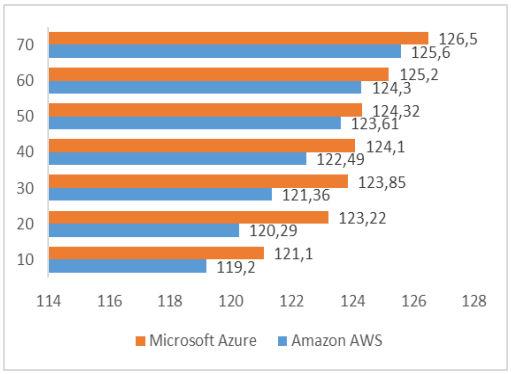


Fig. 14. Management Proxy Memory usage in MB per CSP.

The same conclusion can done by analyzing Fig. 14 where minimal usage of the memory by the MP on each cloud provider can be observed although the number of running VMs has increased from 10 to 70 with no important increase of memory utilization. These results were achieved by optimizing the MP source code and due to the use of the JIT compiler of the PyPy implementation.

V. Conclusion

This paper presented a novel approach of a monitoring framework for Multi-cloud environment where all heavy processing are done at the periphery of the cloud rather than sending raw data to be processed on the main datacenter. The framework, called EMMCS, is scalable and modular using a microservices-oriented architecture where each service is provided with its own RESTful API. For monitoring tasks, EMMCS uses SNMP agents that are installed on each VM on the cloud to collect metrics and QoS statistics.

According to the experiment, the EMMCS framework has proven to be efficient and resources friendly even when monitoring a significant number of VMs. This was achieved by optimizing the code and using high performance technologies such as PyPy, a fast implementation of the Python programming language and Falcon. However, the EMMCS framework is in early stages of development as thus, needs improvement in term of optimization and features. In the future, the focus will be on adding data analysis and early detection using machine-learning systems to prevent from potential problems and to avoid false positives such as high resource consumption due to maintenance tasks, backups and replications, commits in databases, etc. A graphical UI is in the EMMCS's development roadmap to replace the management console that does not provide enough features such as graphing, administration tasks, etc.

References

[1] S. Kolhe and S. Dhage (2012) "Comparative study on virtual machine monitors for cloud", 2012 World Congress on Information and Communication Technologies.

[2] V. Bucur, C. Dehelean and L. Miclea (2018) "Object Storage in the Cloud and Multi-cloud: State of the Art and the research challenges", IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR).

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges", IEEE Internet of Things Journal, Volume: 3, Issue: 5, Oct. 2016.

[4] R. Hillbrecht and L. C. E. de Bona (2012) “A SNMP-based Virtual Machines Management Interface”, IEEE Fifth International Conference on Utility and Cloud Computing.

[5] Y.-C. Peng and Y.-C. Chen (2011) “SNMP-based monitoring of heterogeneous virtual infrastructure in clouds”, 13th Asia-Pacific Network Operations and Management Symposium.

[6] B. Mayer and R. Weinreich (2017) “A Dashboard for Microservice Monitoring and Management”, IEEE International Conference on Software Architecture Workshops (ICSAW).

[7] L. Romano, D. De Mari, Z. Jerzak, and C. Fetzer (2011) "A Novel Approach to QoS Monitoring in the Cloud", First International Conference on Data Compression, Communications and Processing.

[8] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall (2011) "Toward an architecture for monitoring private clouds" Communications Magazine, IEEE, vol. 49, pp. 130-137.

[9] C. Zeginis, K. Kritikos, P. Garefalakis, K. Konsolaki, K. Magoutis and D. Plexousakis (2013) “Towards Cross-Layer Monitoring of Multi-Cloud Service-Based Applications”, ESOCC 2013, LNCS 8135, pp. 188–195.

[10] K. Alhamazani, R. Ranjan, K. Mitra, P. P. Jayaraman, Z. (George) Huang, L. Wang and F. Rabhi (2014) “CLAMS: Cross-Layer Multi-Cloud Application Monitoring-as-a-Service Framework”, IEEE International Conference on Services Computing.

[11] J. L. Hernández-Ramos, M. P. Pawlowski, A. J. Jara, A. F. Skarmeta and L. Ladid (2015) “Towards a Lightweight Authentication and Authorization Framework for Smart Objects”, IEEE Journal on Selected Areas in Communications, Volume 33, Issue 4.

[12] S. Zunke and V. D’Souza (2014) “JSON vs XML: A Comparative Performance Analysis of Data Exchange Formats”, IJCSN International Journal of Computer Science and Network, Volume 3, Issue 4.

[13] S. Khoudali, K. Benzidane, A. Sekkaki and M. Bouchoum (2014) “Toward an elastic, scalable and distributed monitoring architecture for cloud infrastructures”, International Conference on Next Generation Networks and Services (NGNS).

[14] B. Butzin, F. Golatowski and D. Timmermann (2016) “Microservices approach for the internet of things”, IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA).