Elasticity for MQTT Brokers in IoT Applications

Linh Manh Pham[1,2,3], Tien-Quang Hoang[4], Xuan-Truong Nguyen[4]

[1] Graduate University of Science and Technology, Vietnam Academy of Science and Technology, 18 Hoang Quoc Viet, Cau Giay, Ha Noi, Vietnam

[2] Institute of Information Technology, Vietnam Academy of Science and Technology, 18 Hoang Quoc Viet, Cau Giay, Ha Noi, Vietnam

[3] VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

[4] Hanoi Pedagogical University 2, 32 Nguyen Van Linh, Xuan Hoa, Phuc Yen, Vinh Phuc, Vietnam

Correspondence: Linh Manh Pham, linhmp@vnu.edu.vn

Communication: received 7 November 2020, revised 7 December 2020, accepted 31 January 2021

Digital Object Identifier: 10.32913/mic-ict-research.v2020.n2.941

***Abstract*: Many domains of human life are more and more impacted by applications of the Internet of Things (IoT). The embedded devices produce masses of data day after day requiring a strong network infrastructure. The inclusion of messaging protocols like MQTT is important to ensure as few errors as possible in sending millions of IoT messages. This protocol is a great component of the IoT universe due to its lightweight design and low power consumption. Distributed MQTT systems are typically needed in actual application environments because centralized MQTT methods cannot accommodate a massive volume of data. Although being scalable decentralized MQTT systems, they are not suited to traffic workload variability. IoT service providers may incur expense because the computing resources are overestimated. This points to the need for a new approach to adapt workload fluctuation. Through proposing a modular MQTT framework, this article provides such an elasticity approach. In order to guarantee elasticity of MQTT server cluster while maintaining intact IoT implementation, the MQTT framework used off-the-shelf components. The elasticity feature of our framework is verified by various experiments.**

Keywords: *Elasticity, MQTT broker, Cloud computing, Internet of Things.*

I. INTRODUCTION

The Internet of Things (IoT) has had a significant influence on many aspects of life at various scales, ranging from households to enterprises to large organizations or industries. Millions of connected computers, with or without human interference, are behind IoT applications, attempting to communicate and transfer data over the Internet. It is predicted that, by 2050, there will be about 170 billion objects or things connected to the Internet [1]. Additionally, the IoT network can hold between 50 and 100 trillion connected objects, and it can track the movement of each object. Every person living in metropolitan areas can be surrounded and monitored by 1000 to 5000 IoT objects. Currently, by 2020, our world has 4 billion connected people, over 25 million applications, over 25 billion embedded and intelligent systems, and 50 trillion Gigabytes of data generated. This market produces 4 trillion dollars in revenue for service providers [2].

In order to support the communication of billions of IoT devices and to rotate the produced masses of data, IoT service providers need to deploy and maintain robust and scalable network infrastructures. When IoT applications scale beyond the scale of household applications to larger systems such as cities or nations, the number of IoT devices can grow very quickly with unpredictable degrees. That is why IoT infrastructures are being developed that not only have strong fault tolerance but also need to be scalable. Most modern IoT infrastructures today deploy an essential component known as the Message Queuing Telemetry Transport (MQTT) server. These brokers use the MQTT protocol, a Machine to Machine (M2M) open protocol that has been around since 1999. It is an open industry standard issued by OASIS and ISO (ISO/IEC 20922) [3]. Thanks to advantages such as compact size, bandwidth savings, low battery power consumption, and space/time separation, MQTT brokers are increasingly undeniably dominating the field of IoT.

Although MQTT broker solutions recently applied both centralized and distributed approaches to manage millions of incoming connection from end devices in a short amount

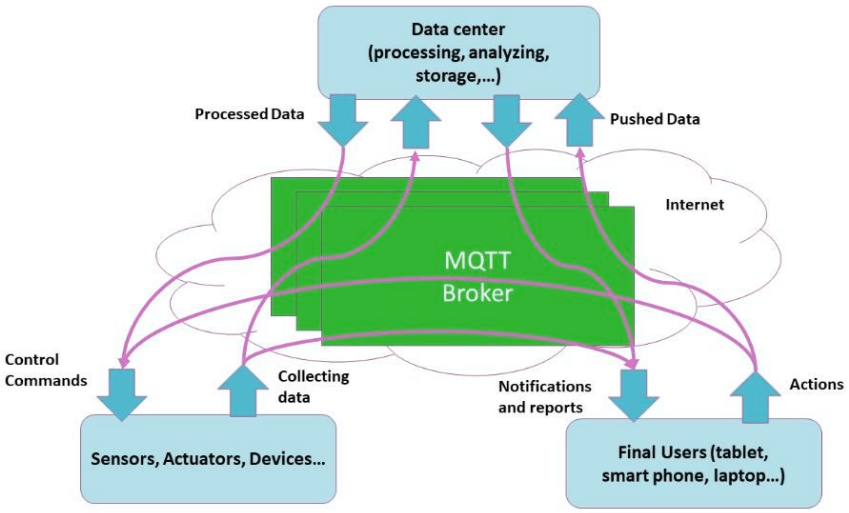

Figure 1. Typical structure of an IoT application using MQTT brokers

of time, there are only several solutions which have functions of adapting to fluctuations in workload generated by loT devices. This variation in load can occur when the number of loT devices fluctuates unpredictably during certain times. In reality, citywide loT applications often deal with dispersed and sporadic terminal equipment, resulting in a shift in the number of devices involved in the application. For example, smart cars are more likely to link to vehicle networks connected during peak hours than during off-peak hours, resulting in the generation of more IoT data during specific times. This necessitates the development of new MQTT-based systems that not only understand how to scale to meet demand but also keep up with changes in workload created by the terminal. In other words, it is a strategy for making MQTT's brokers more flexible.

Elasticity is a fundamental property of Cloud Computing as defined by NIST [4]. Because of scalability, cloud resources are not overused or under-utilized. This does not only save money for the Cloud service provider, but also positively enhances customer experience with the service provided. It is a fact that many IoT applications have already been deployed or are in the process of being deployed on the Cloud. It is worth noting that loT resources such as MQTT brokers deployed on the Cloud may also benefit from this elasticity characteristic of the Cloud.

In this article, we suggest an architectural framework for making MQTT brokers elastic. To achieve this aim, here is what to do:

- We propose a new architecture framework that can flexibly enable elasticity to distributed MQTT brokers while maintaining all the original features of the MQTT protocol.
- We specifically implement the proposed architectural framework using the open-source MQTT brokerage service (EMQX) and private cloud platform (OpenStack).
- We perform experiments to validate the proposed architecture framework using the version deployed with the aforementioned open-source applications.

The rest of the paper is laid out as follows. Section 3 describes in detail the overall architecture of the proposed MQTT architectural framework after highlighting related studies in Section 2. Section 4 discusses implementing the architecture framework using suitable open-source tools. Section 5 presents the validation as well as the results. Finally, in Section 6, we make our conclusions.

II. RELATED WORK

1. MQTT

MQTT is a message-oriented middleware (MOM) software following the publish/subscribe model [5]. To ensure secure transmission in environments with many limitations, IoT applications clearly need an efficient and powerful solution such as the MQTT broker model. Some of the most commonly used MQTT broker solutions today are EMQX, HiveMQ, VerneMQ, Moquette, Mosquitto, etc.

In recent decades, many IoT applications have used MQTT brokers such as [6], [7], [8], [9], [10], [11]. The typical structure of an IoT application using MQTT brokers deployed with remote centralized data centers (as in a Cloud environment) is depicted in Figure 1. The application aims to gather data from a variety of loT devices and sensors, then process and store these data, and finally sends notifications and reports to end-users (using laptops, mobile

devices, tablets, etc.). In certain situations, collected data may be released directly to the topics subscribed to by the end-user without any data analysis. Control commands, like every other form of MQTT message, can be published to the command topic in the brokers. These messages will be stored in cloud archives and transmitted to IoT devices or sensors using some programming mechanisms. In the case of time-sensitive applications, command messages can also be sent directly to IoT devices without having to be sent to the Cloud. We discovered that the IoT devices, end-user interfaces, and data analysis systems are all different types of MQTT clients that produce and ingest remotely measured data.

2. Distributed MQTT broker

Many IoT applications typically implement a centralized MQTT broker capable of keeping all subscribed topics. However, in this model, it is easy for the server to become an obstacle to the entire system. To prevent this, a variety of distributed solutions have been suggested, which can be divided into two categories: a bridged broker and a clustered broker. The two brokers could be bridged in the first model to serve more messages from the client while remaining separate from each other's locations. Published messages are forwarded from a broker to a broker bridged with its specific access policies. A complete mesh network needs to be formed between the brokers (each one must have connections to all the other machines) so that any MQTT client can connect to any brokers that they want. Therefore, using the bridging model to gain the elastic function is too complex. It is only suitable for simple networks which have a few MQTT brokers. The MQTT brokers that support bridging include EMQX, HiveMQ, VerneMQ, Moquette, Mosquitto, JoramMQ, etc. [12]. In the study of Collina et al. [13], Schmitt et al. [14], and Zambrano et al. [15], some implementations of this model have been reported.

MQTT brokers in a cluster model utilize subtopics in a hierarchical structure. One of the brokers (B_0) keeps the original topics and the child topics that have registrations related to them. The other servers (B_1, B_2, etc.) only keep relevant subtopics that originated from B_0. Topic branches operate in a server that corresponds to MQTT registrations for these brokers. As a result, the costs between servers based on the cluster model are significantly reduced compared to the bridging model. Thanks to the brokers' transfer knowledge about the topic and the routing table, any MQTT client may establish or continue their session by connecting or reconnecting to any broker. Only a few MQTT brokers, such as RabbitMQ, VerneMQ, EMQX and HiveMQ [16], currently support the full capabilities of the cluster model. The work of Jutadhamakorn et al. [17], Thean et al. [18], and Detti et al. [19] are some of the studies that follow this trend.

3. Elastic MQTT broker

One of the fundamental properties of Cloud Computing is elasticity. Cloud resources can be provisioned or released when demanded thanks to this unique feature. Today, loT applications are often deployed on the Cloud to utilize the advantage of this environment such as on-demand measurement services, wide-area network capability as well as radically supported resource elasticity. Many solutions, including DOCKERANALYZER [20], Proliot [21], BDAaaS [22] and ACD [23] have attempted to provide scalability for components of IoT applications. However, only a few elasticity solutions for MQTT brokers have been proposed, including Broker [24], E-SilboPS [25]. Many MQTT-specific customizations have been simplified or ignored because Brokel only defines a multilevel elasticity model for brokers through the publish/subscribe model (including MQTT) in general. E-SilboPS is a scalable content-based subscription/publishing system specifically designed to support sensing and contextual communication in IoT-based services. Therefore, it also ignores many of the customized Quality of Service (QoS) parameters of the MQTT protocol and only provides content-based elasticity. One of the prerequisites for a comprehensive elastic MQTT is that the solution must use one of the distributed models mentioned in sub-section II.2.

III. ARCHITECTURE OF ELASTIC MQTT FRAMEWORK

This section presents the proposed elastic MQTT architectural framework. The system is designed with a flexible architecture containing a representative set of modules. When a version of the framework is deployed on the Cloud, each representation module is specialized into a specific open-source software component that is already available. Framework's modules can therefore be flexibly replaced to gain new features, for higher efficiency, or lower software licensing costs. The architecture framework also supports the MQTT clustered broker model to reduce communication costs among the cluster servers. Figure 2 shows the framework's overall architecture, which includes the following modules:

- **MQTT Broker Cluster**: A group of MQTT brokers implements a distributed publish/subscribe model with the customized QoS parameters. The cluster contains several systems that provide an execution environment called a node. The nodes connect to each other using

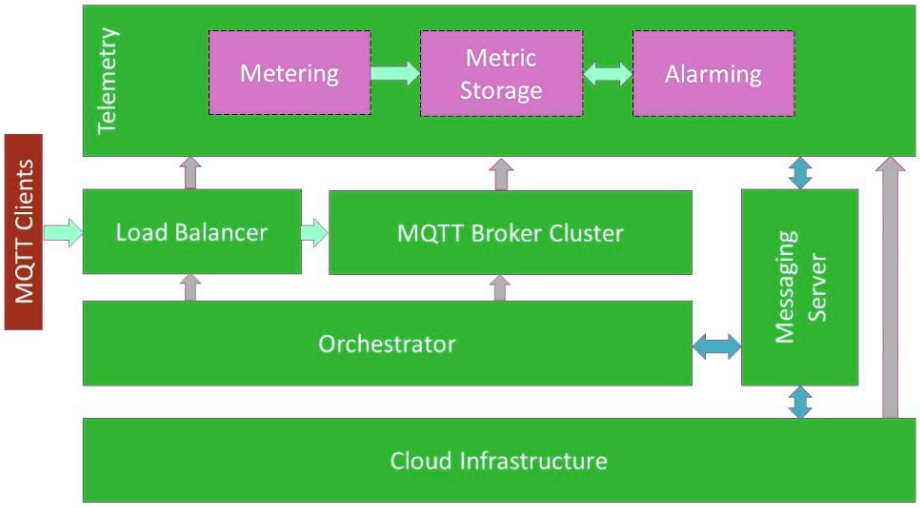

Figure 2. The architecture of Elastic MQTT Framework

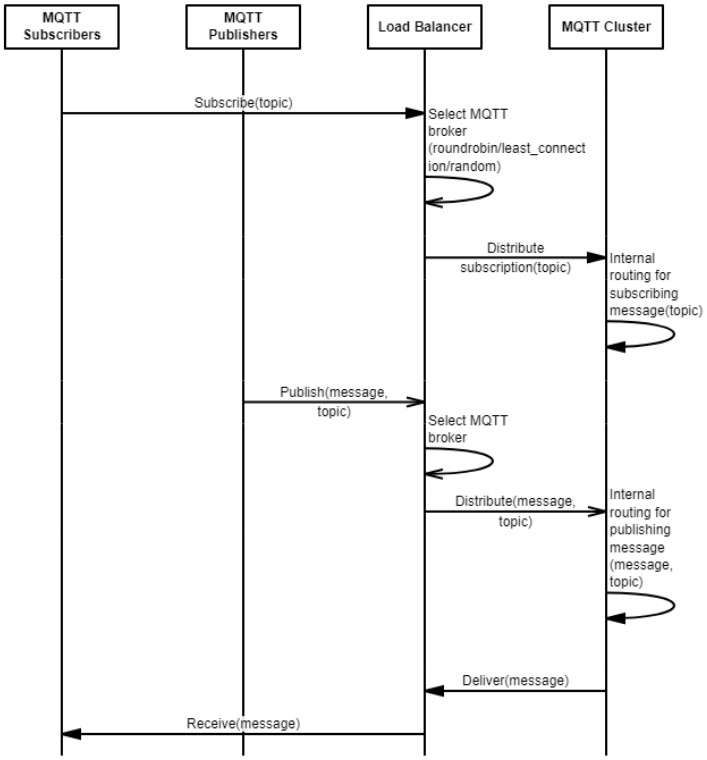

Figure 3. Schematic diagram of data flow during publishing - subscribing messages using load balancer

TCP/IP and communicate by transmitting the message. Each node holds sections related to itself in the topic system and the current subscriptions. This mechanism enables the published messages to be routed over the entire cluster from the first node that receives the message to the last node that sends it to the subscriber. Nodes can join the cluster manually or automatically. With automatic manner, automatic mechanisms for cluster identification and joining such as IP multicast, dynamic DNS or ETCD [26] need to be supported. Nodes can be deployed on either public or private clouds. Public Cloud providers, such as AWS, Azure, or private cloud platforms, such as OpenStack, CloudStack, could all be viable options.

- **Load Balancer**: A Load Balancer (LB) is usually deployed in front of an MQTT cluster to distribute the

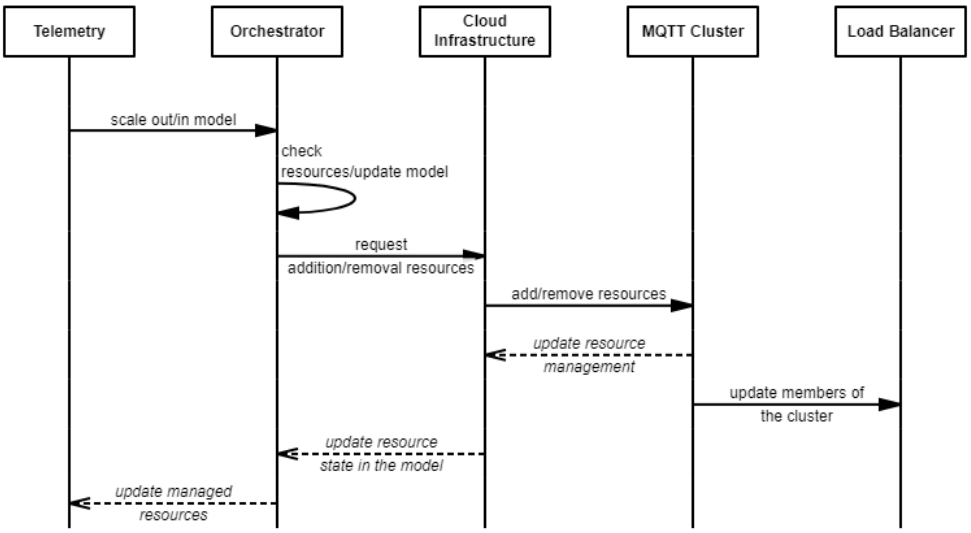

Figure 4. Data flow of components when it has a resource scaling event on MQTT brokers

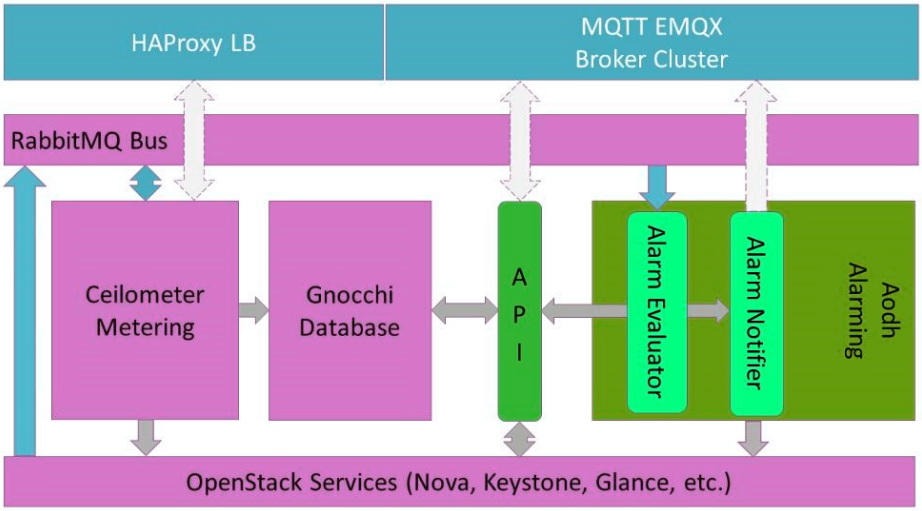

Figure 5. An implementation of the elastic MQTT architectural framework

connections and messages from the devices into the MQTT cluster. LB also enhances the availability of clusters, balances load between nodes in the cluster, and provides dynamic scalability. The links between LB and the nodes in the cluster are simple TCP connections. By doing this, a single MQTT cluster can serve millions of clients. Thanks to LB, the MQTT client only needs to know one connection point instead of maintaining a long list of MQTT brokers. A schematic diagram of the data flow during publishing - subscribing messages using a load balancer is illustrated in Figure 3.

- **Orchestrator**: This module parses descriptions of system components in their domain-specific high-level language (i.e., Domain-Specific Language - DSL) and then deploys, manages, and monitors the full life cycle of all components involved. Those components include resources such as virtual machines, containers, images, security groups, alerts, extension policies, etc. DSL grammar can be derived from XML, JSON, or YAML with the primary motivation to keep things simple and user-friendly. Within the architecture framework, the Orchestrator deploys and manages the MQTT brokers as well as resources of elastic decision-making block such as Metering, Metric Storage, and Alarming.
- **Telemetry**: This module includes three sub-modules: Metering, Metric Storage, and Alarming.

  Metering: The goal of this module is to collect, standardize, and transform data generated by coordinated components effectively. These data will be used to gen-

erate multidimensional views and help resolve different telemetry cases. Among them, data of specific metrics (i.e., measurements) are collected and analyzed for the purpose of decision-making on elasticity. Alarming and Metric Storage are two modules that directly exploit these measures.

Alarming: Its goal is to provide the ability to trigger response actions based on predefined rules relied on sample data or events collected by the Metering module. It consists of two main sub-modules: "Alarm evaluator" and "Alarm notifier". The first module evaluates the measurements of a given metric stored in the Metric Storage module to find out whether they are over or below a predetermined threshold or not. The second module activates the notifications and sends them to the Orchestrator to order the corresponding elastic actions such as increasing/ decreasing the number of MQTT brokers.

Metric storage: A database that mainly stores aggregate measures of cluster nodes such as system performance metrics. A measurement is a list of the form *(timestamp, value)* for a given managed resource. Resources can be anything from the temperature of the nodes to the CPU usage of a virtual machine. Besides, the database also stores events, which are lists of things happening in the Cloud infrastructure such as a request to an API received, a virtual machine started, a photo uploaded, or anything else. Stored measurements are retrieved for monitoring, billing, or alerting purposes, in which saved events are useful for testing, performance analysis, debugging, etc.

- **Cloud infrastructure**: It manages, provides, and dynamically releases virtual resources for scalability. For "unlimited" resources, all private, public, or hybrid clouds may be required to deploy.
- **Messaging server**: It is essential to carry out communication between the modules of the architecture framework by exchanging messages. It creates channels to be connected using popular communication protocols like CoAP, AMQP or even MQTT.

All the modules of the framework are detachable. That means the boot order of the components is not entirely crucial. Even so, some modules working independently usually do not make much sense, so there are still prerequisites for the start-up of other modules beforehand. Likewise, the components and resources are described and managed by *the Orchestrator*, which can be instantiated at any point in time. The Orchestrator must be able to resolve dependencies between the components and therefore come up with a deployment plan that contains the correct order of installation. From the system description to the deployment plan, the Orchestrator needs to use a series of resolvers such as Learning Automata based allocators, Constrained Programming-based solvers, and Heuristics and Meta solvers. When an event or combination of events and conditions occurs during execution, the Orchestrator generates the corresponding scaling plan and makes the necessary modifications to transform the existing implementation model to the proposed deployment model describing in the elastic plan. Modifications include actions that comply with ECA (Event-Condition-Action) laws such as horizontal or vertical scaling of a resource when measurements of the resource violate the thresholds beforehand. Figure 4 illustrates a simplified coordination diagram between the components mentioned in the MQTT architectural framework during the resource elastic event.

IV. THE IMPLEMENTATION

In this section, we present a specific deployment of the architectural framework proposed in the previous section. It mainly supports cloud-based IoT applications that require scaling as an essential feature. These applications include, but are not limited to, applications for big data analysis or applications that are sensitive to latency. With the principle of developing software for the open science community [27], we prioritize the combination of open source solutions that are most suitable for our architectural framework where they are being used universally. Figure 5 depicts a deployment of the architectural framework whose details are described as following.

- **MQTT broker cluster using EMQX**: EMQX is based on the distributed Erlang / OTP programming platform and the Mnesia database [28]. It provides broker nodes that run in parallel and are highly fault-tolerant and distributed. It is one of the few open-source MQTT solutions that support the clustered model. Furthermore, EMQX is the only solution that supports all three levels of MQTT QoS, as well as both MQTT protocols for conventional networks and MQTT-SN for sensor networks. EMQX supports node autodetection and cluster auto-joining with various strategies such as IP multicast, ETCD, dynamic DNS and K8s [29]. In that way, when a broker node comes in or out corresponding to the scaling action, the cluster will automatically recognize the changes and update its configuration to reflect the number of new nodes.
- **Load Balancer**: Several commercial LB solutions that are supported by EMQX such as AWS, Aliyun, or QingCloud. On open-source software, HAProxy [30] can act as an LB for the EMQX cluster and allocate TCP connections. Many dynamic scheduling

algorithms can be assigned on HAProxy such as round robin, least connections, or random.

- **Cloud infrastructure**: To serve the open science community, OpenStack [31], an open-source private Cloud, is chosen to provide and release virtual resources. With a supported worldwide community of users and well-maintained mighty services, OpenStack fits our goals well. Some of the specific OpenStack services used for our deployment are Nova, Keystone, Glance, Horizon, Swift, and Neutron. Because OpenStack cloud is selected, the following modules are officially supported by OpenStack services.
- **Orchestrator**: The official orchestrator supported by OpenStack is Heat service [32]. The infrastructure for a cloud application is depicted in the legibly Heat template file and edited by humans. Infrastructure resources that can be described include servers, storage, users, security groups, floating IP, etc. Heat also provides an automatic elasticity service that integrates with *the Telemetry* sub-modules, so the scaling server group can be included as a resource in the template file. This is a perfect fit for achieving elasticity goals. Sample files can also describe dependencies between resources (for example, this floating IP is assigned to this virtual machine). This helps Heat generate all managed components in the correct order to fully launch the application. Heat manages the entire life cycle of an application and knows how to make the necessary dynamic changes. Finally, it also handles the deletion of all deployed resources when the application completes.
- **Telemetry**: OpenStack has several officially supported services for the Telemetry. These include the Ceilometer service for Metering, Aodh for Alarming, and Gnocchi for Metric Storage.

  Metering: The Ceilometer service [33] provides the following functions: (1) Exploration of measurement data of the OpenStack service, (2) Collecting events and measuring data by tracking notifications sent from the service, (3) Exporting data collected to various defined targets including data warehouse and queue containing record.

  Alarming: When the measurement data or the collected event breaks a given rule, the Aodh service will trigger an alert [34]. The service consists of the following components: (1) An API that provides access to the warning information stored in the metric storage. (2) A alarm evaluator that determines when warnings are triggered due to measurements over a threshold for a period of time. (3) A alarm notifier that allows setting warnings based on an evaluation over a threshold for a collection of patterns.

  The Metric Storage: Gnocchi is an open-source time series database [35]. It attempts to solve storing problems and index time series measurements of OpenStack resources on a large scale. Gnocchi deploys an unique approach to time series storage: instead of storing raw data points, it aggregates (average, minimum, etc.) them before storing them. Since Gnocchi calculates all measurements as aggregate ones when collecting, importing, and processing data, data retrieval is done quickly by re-reading calculated results from before.
- **Messaging server**: In the OpenStack cloud, internal communication between OpenStack services can be performed by RabbitMQ [36]. RabbitMQ is an open source message-oriented middleware that supports popular communication protocols such as STOMP, AMQP and MQTT.

V. EVALUATION

To evaluate and validate the functions of the proposed framework, we performed the implementation described in Section 4 in the data center of VNU University of Engineering and Technology (VNU-UET). We also give some discussions on the results of the experiments.

1. Installing experiment system

The experimental system consists of two main parts: a deployment of our proposed architecture framework and a load generator to simulate multiple MQTT clients and load their work. We create test plans with different scenarios using the available functionality of the load generator. The publishers are facsimiled to create MQTT messages and send them to the LB (HAProxy). In LB turn, it distributes these messages to a cluster of EMQX brokers according to a predefined scheduling algorithm. The simulated subscribers also perform MQTT subscriptions to specific topics in the cluster by connecting to LB. LB also distributes connection requests from subscribers to one of the cluster members. Messages routed from the source to the correct destination are conducted internally in the cluster as mentioned in Section 3. Table I shows the configuration values of the main parameters for HAProxy version 1.8.8 used in the experiment.

We used Apache JMeter [37] version 5.3 as a load generator in our tests. It is an open-source tool to measure the effects of simulation load and evaluate performance. It supports the experiment of various types of protocols like REST, HTTP, HTTPS, JMS, FTP, SOAP, etc. Other protocols can be added to JMeter using plugins. To support the experiments, we developed an MQTT plugin for JMeter

that implements some features of the MQTT 5.0 version [38]. To perform the stress tests, we used a distributed experiment model with a primary JMeter server and a couple of passive JMeter servers. This ensures that there were no side effects on the performance of the MQTT clients simulated.

Our OpenStack private cloud is installed in our data center for research of VNU-UET. EMQX brokers and JMeter load generators are installed in cloud-powered virtual machines. Each EMQX broker instance has 2 vCPU and 2 GB memory. Each JMeter virtual machine has 8 vCPU and 8 GB memory. We use OpenStack Train, which was released in 2019. Our OpenStack Cloud is built on 3 physical servers using an Intel processor. Each physical server has 80 CPU with 2.4 GHz, 256 GB memory and 1.5 TB storage. CentOS 7 is installed on all physical machines as the host operating system. On top of that, KVM is used as a base virtualization solution. For better resource isolation, we install OpenStack controller services (e.g. Nova, Neutron, Keystone, etc.) on the dedicated virtual host instead of running them directly on physical servers. Only the hypervisor service (also known as OpenStack Nova Compute) manages the running of virtual machines, which will be installed directly on the physical servers. In this way, the system services of the OpenStack cloud itself are completely separate from the virtual server instances created by the cloud users. In short, the stress tests performed in our experiments run on OpenStack virtual servers, which will not impact cloud performance and vice versa.

2. Experimental scenarios

We conducted experiments with two common scenarios often found in IoT applications using MQTT: Multi-publisher and Multi- subscriber. Each scenario assesses the effectiveness of the elasticity performance with two models: MQTT centralized broker and MQTT clustered broker.

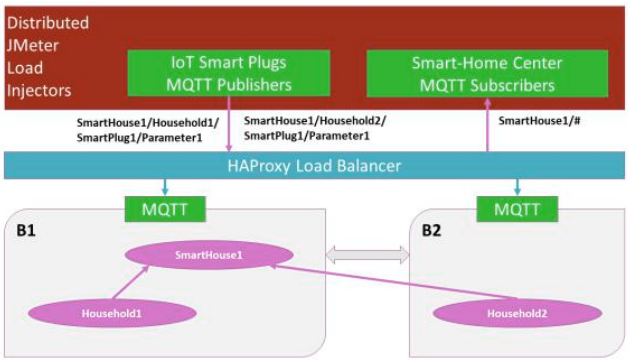

Figure 6. Scenario Multi-publishers on MQTT brokers cluster

TABLE I
MAIN PARAMETERS FOR HAPROXY ARE USED IN THE EXPERIMENTS

Parameters	Value	Meaning
listen	mqtt	Create process "listening" with the responsibility to wait and process MQTT packets.
bind	*:1883	The "listen" process will wait on port 1883 from every network.
mode	tcp	The "listen" process operates in TCP mode.
maxconn	50000	The maximum number of connections per "listen" accepted.
default_backend	emqx_back	The name of the rear broker cluster.
option	clitcpka	Turn on sending TCP keepalive packets on the client-side.
option	srvtcpka	Turn on sending TCP keepalive packets on the server-side.
backend	emq_back	Create a rear broker cluster.
balance	roundrobin	Scheduling mode in turns with round.
timeout check	5000	Set the time to wait for additional check after the connection has been established.
server	emqx_node[i]	Identifies the "i" node that participates in the cluster.

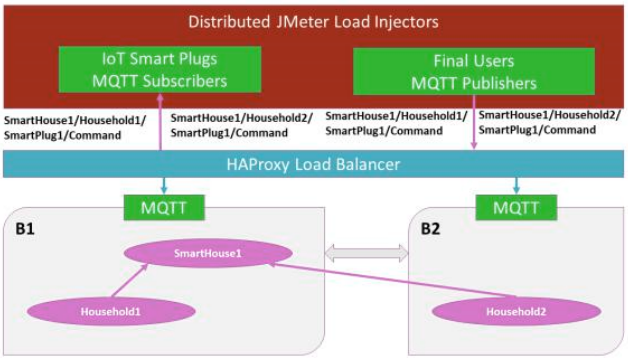

Figure 7. Multi-Subscriber clustered brokers scenario

a) Multi-publisher scenario

This scenario simulates a large number of IoT devices, such as smart plugs, publishing telemetry data to a central smart home system. The plugs are the publishers and the central smart home system is the subscriber. The threads are structured like a tree with three levels. The top-level represents smart houses in a district. The middle level is the households in a smart house. The wireless access point in every household allows smart plugs to connect to the Internet and publish data to a central smart home system.

The last level of the tree is smart plugs that send energy

TABLE II
THE CONFIGURATION PARAMETERS OF THE EMQX BROKER CLUSTER

Parameters	Value	Meaning
cluster.discovery	mcast	Automatically discovers and creates clusters based on UDP multicast.
cluster.mcast.addr	239.192.0.1	EMQX multicast address.
cluster.mcast.ports	4369, 4370	EMQX multicast ports.
cluster.mcast.iface	0.0.0.0	Indicates which discovery service the local IP address should link to.
cluster.mcast.ttl	255	Indicates the Time-To-Live value of multicast.
cluster.mcast.loop	on	Indicates if multicast packets have been sent to local loopback address.
cluster.autoclean	5m	Indicates the time to wait before removing inactive nodes from the cluster.

measurements of devices to Wireless access points. A lower topic level added below is the measure level that represents telemetry parameters such as power consumption (kWh). The experiment defines 40 partitions of topics including:

- 1 root topic for smart house "SmartHouse"
- 3 topics "Household" for each smart house
- 30 topics "SmartPlug" for each household
- 10 topics "Parameter" for each smart plug

Thus, a partition of the topic represents 90 smart plugs and each plug publishes 10 telemetry parameters. We have 3600 smart plugs totals in the whole scenario. The experiment for this scenario is depicted in Figure 6.

b) Multi-subscriber scenario

This scenario also simulates a large number of smart plugs controlled by a central smart home system. These smart plugs are subscribers and the central smart home system is the publisher. "SmartPlug" provides bidirectional communication. The end users and the smart house center can send commands to the plugs. Besides, the smart plugs can also respond to these commands, an indicator of an ON/OFF update, for example. The topics are divided into a couple of levels like Multi-publisher scenario. We also define topic partitions presenting 3600 smart plugs providing a control interface, each partition includes:

- 1 root topic for smart house "SmartHouse"
- 3 topics "Household" for each smart house
- 30 topics "SmartPlug" for each household
- 1 topic "Command" for each smart plug

The experiment for Multi-subscriber is depicted in Figure 7.

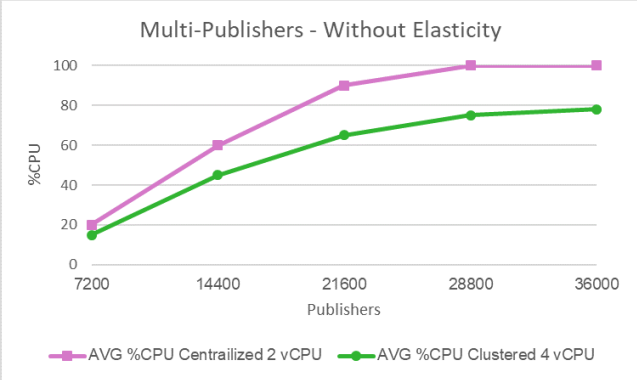

Figure 8. Without Elasticity: Average %CPU Usage of the Multi-Publisher Scenario

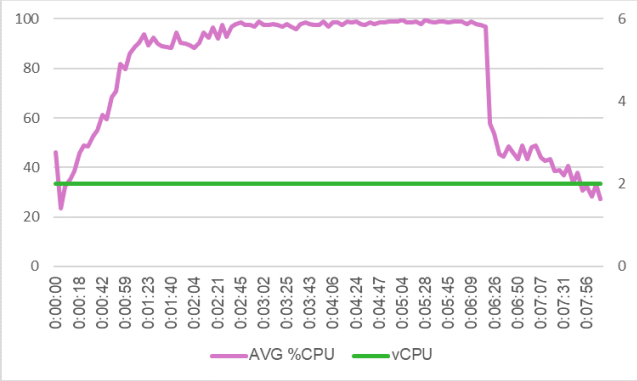

Figure 9. Average %CPU Usage of Multi-Publisher Scenario with the Centralized Broker

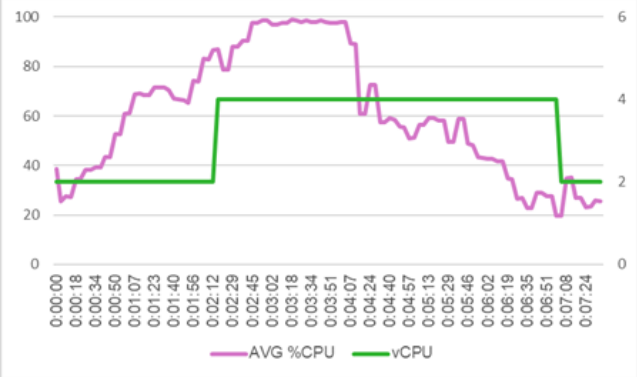

Figure 10. Average %CPU Usage of the Multi-Publisher Scenario with the Clustered Brokers with Elasticity

3. The results

MQTT workloads are created by editing JMeter scripts. The workload starts with a short warm-up and then increases significantly as MQTT clients quickly join the simulation. The EMQX broker is already configured following suggestions from EMQX documentation. We chose IP multicast methods for automatic node-discovery and auto-

join cluster mechanisms. Table II shows the configuration parameters of the EMQX broker cluster. The scheduling algorithm for HAProxy is installed as *roundrobin* (delivered sequentially in turn).

Ceilometer, Aodh, and Gnocchi were configured to measure and store measurements of average %CPU utilization and the number of virtual CPUs (vCPU). Upper and lower thresholds for average %CPU utilization are set to 80% and 25% respectively. It means that if average %CPU utilization breaks the thresholds and the events caught by Ceilometer and Aodh, a notification is sent to Heat for conducting a corresponding elasticity action such as scaling in or out. Actually, Heat has to ask other OpenStack services such as Nova, Keystone or Glance to get the elasticity actions done synchronously. Elasticity plan configured in Heat ensures the number of VMs always in range of 1 to 3.

We used two JMeter client machines for distributed tests. In each client machine, a maximum of 5 JVM processes are allowed to initiate. According to the test scenarios, each process is responsible for running 3600 MQTT clients. Therefore, the maximum 36000 MQTT clients can be started and run in two client machines. To increase saturated probability of the brokers, QoS level of publishing and subscribing MQTT messages is fixed to 2 and "clean session" flag set to FALSE in all experiments.

a) Multi-Publisher scenario

In the case of using a centralized MQTT broker, there is only one subscriber per topic partition. This subscriber listens to all the topics of the partition by subscribing to "SmartHouse/#" with wildcard mask "#" denoting all subtopics of the root topic "SmartHouse". One publisher is created for every topic "SmartPlug" sending messages to the topics "Parameter" below the topic "SmartPlug". In total, 3600 publishers send messages to 36000 topics "Parameter" at a steady rate which is one message/second. In the clustered case, the multi-publisher scenario is tested with a cluster of two brokers B_0 and B_1 (B_1 will be added dynamically when needed). Publishers (90 each partition) and subscribers (one each partition) are equally load-balanced across the two brokers.

Figure 8 shows the average %CPU utilization in both centralized and clustered cases without elasticity. We see that the MQTT system with one broker (2vCPU) is easy to be saturated. Adding one more broker (4vCPU totally) to form the cluster can help resolving the problem. On the other hand, to compare service quality improvement of centralized cases with inelastic clustering using 2 brokers, we performed the experiment 20 times with the average time to complete a scenario which is 6 minutes in the centralized case and 4 minutes 30 seconds in the inelastic clustering case, respectively (down 25%). In both cases, the packet publication rate is halved to avoid brokers saturation which is very likely in centralized cases. In the centralized case, we see in Figure 9 that the average %CPU utilization of the broker reaches saturation after a couple of minutes (at the 1st minute). At this point, the dropped message rate starts to increase. With elasticity, operating cost reduces since we do not have to always maintain multiple brokers (clustered brokers). In Figure 10, we see an elasticity effort to mitigate the pressure performed by our system. One virtual machine of MQTT broker B_1 is created to share the workload. This broker automatically joins the cluster created beforehand by B_0 using the multicast method. The change in the topology is announced to HAProxy for reloading its configuration. The reloading process needs to be used instead of restarting one in order to lower the server downtime as much as possible. After reloading, HAProxy recognizes the new server and distributes messages to all load-balancing members. At the end, the average %CPU utilization of the broker reduces under the lower threshold after a period of time. Thus, we see another elasticity action (scaling in) at this time of the simulation when MQTT clients are finished or terminated. At this point when the workload goes under 25%, the number of VMs is decreased to one for minimizing operating cost.

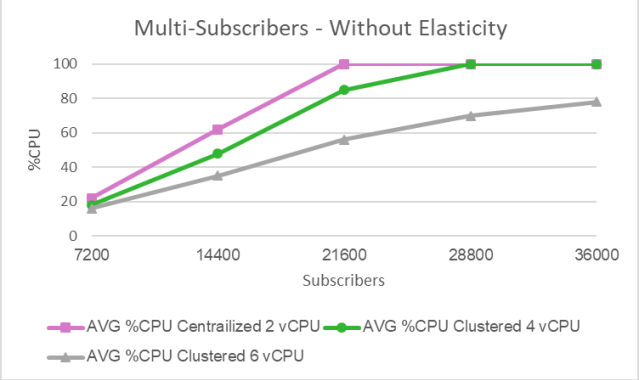

Figure 11. Without Elasticity: Average %CPU Usage of the Multi-Subscriber Scenario

b) Multi-Subscriber scenario

In the case of centralized brokers, there is only one publisher per topic partition. This publisher sends messages to all topics of the partition at a steady speed. A publisher is created for each "SmartPlug" topic. Each publisher receives a message from the "Command" topic under the "SmartPlug" topic. On all the partitions, 3600 publisher received messages from 3600 "Command" topics at a steady speed. In the case of the cluster brokers, the multi-publisher scenario is tested with a group of two MQTT brokers B_0 and B_1 (B_1 dynamically added as needed). The subscribers (90 per partition) and the publisher (each partition has only

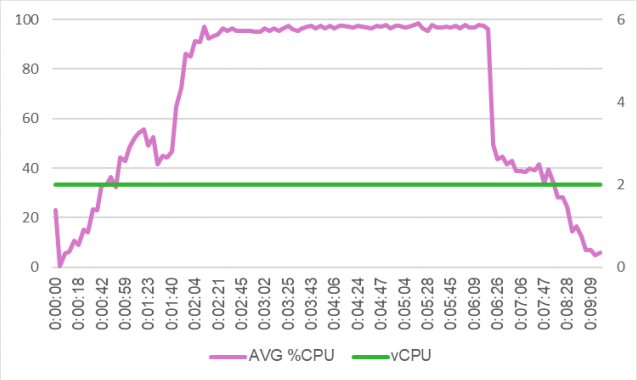

Figure 12. Average %CPU Usage of Multi-Subscriber Scenario with the Centralized Broker

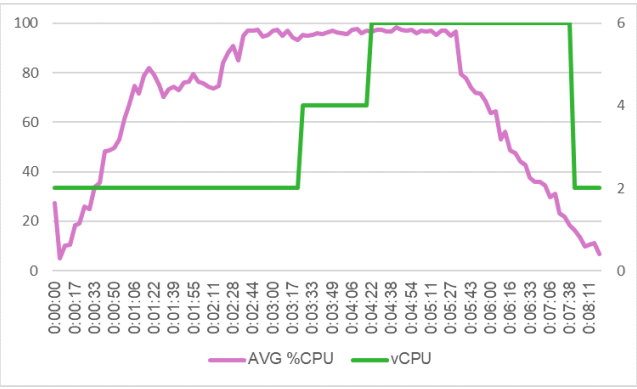

Figure 13. Average %CPU Usage of the Multi-Subscriber Scenario with the Clustered Brokers with Elasticity

one publisher) are load balanced and distributed evenly to the two brokers B_0 and B_1.

Figure 11 shows the average %CPU usage in both centralized and cluster cases which do not have elasticity. We saw the same behavior as that of the multi-publisher scenario. Adding one more broker to the cluster is not really helpful, but two brokers (6 vCPU) might solve the problem. On the other hand, for the comparison of service quality improvement between centralized and inelastic clustering cases using 3 brokers, we performed the experiment 20 times with the average execution time to complete the scenario which is 6 minutes 30 seconds in the case of centralization and 5 minutes 50 seconds in the case of inelastic clustering, respectively (down 10%). In both cases, the message publication rate is halved to avoid brokers saturation which is very easy to occur in centralized case.

In the centralized case, we also see clearly in Figure 12 that the broker's average %CPU usage will saturate after a few minutes (at the second minute). At this point, the proportion of unsolicited discarded messages also begins to increase.

With elasticity, we also see the same behavior shown in Figure 13, which is the same for many publishers. The scaling action with two brokers is triggered later than the multi-publisher scenario. These two brokers are added sequentially by Heat service. A one-minute gap is set between broker additions to avoid rapid fluctuations of resources. Average CPU usage stayed above the threshold for a longer time than the multi-publishers scenario. The reason is that the combination of the QoS 2 level and the "clean session" flag set to FALSE will keep the messages at the brokers for a longer time, so the more the publishers run in parallel, the busier the brokers are.

VI. Conclusion

We have presented a flexible architectural framework that can support scaling functionality for distributed MQTT brokerage services in IoT applications. Our architectural framework provides service elasticity by using open source software that is currently commonly used in the cloud computing environment. Our elasticity MQTT brokerage service has been successfully deployed using EMQX as the MQTT brokers and OpenStack as a private cloud environment. Experiments are conducted by generating the IoT traffic for the service at different load levels to observe changes in the number of broker instances. Our experimental results show that the proposed MQTT brokerage service adapts well to the end-user load changes while maintaining low operating costs.

Acknowledgement

This research is funded by Graduate University of Science and Technology under grant number GUST.STS.ĐT2019-TT02.

References

[1] N. Sharma and D. Panwar, "Green IoT: Advancements and Sustainability with Environment by 2050," *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp. 1127-1132, 2020.

[2] V. Turner, D. Reinsel, J.F. Gantz, S. Minton, "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things," *Journal of Telecommunications and the Digital Economy IDC Report Apr*, 2014.

[3] MQ Telemetry Transport, "http://mqtt.org," visited on Oct, 2020.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)," *NIST special publication*, vol. 800, p. 145, (2011).

[5] P. Th. Eugster, P. A. Felber, R. Guerraoui, A. Kermarrec, "The many faces of publish/subscribe". *ACM ACM Computing Surveys*, Vol. 35, No. 2, pp. 114-131, 2003

[6] R. Kawaguchi, M. Bandai, "Edge Based MQTT Broker Architecture for Geographical IoT Applications," *2020 International Conference on Information Networking (ICOIN)*, pp. 232-235, 2020.

[7] V. Gupta, S. Khera, N. Turk, "MQTT protocol employing IOT based home safety system with ABE encryption," *Multimed Tools Appl*, vol. 80, pp. 1-19, 2020.

[8] Mukambikeshwari, A. Poojary, "Smart Watering System Using MQTT Protocol in IoT," *Advances in Artificial Intelligence and Data Engineering. Advances in Intelligent Systems and Computing*, vol. 1133, pp. 1415-1424, 2020.

[9] Y. C. See, E. X. Ho, "IoT-Based Fire Safety System Using MQTT Communication Protocol," *ijie*, vol. 12, no. 6, pp. 207-215, 2020.

[10] S. Nazir, M. Kaleem, "Reliable Image Notifications for Smart Home Security with MQTT," *International Conference on Information Science and Communication Technology (ICISCT)*, pp. 1-5, 2019).

[11] P. Alqinsi, I. J. M. Edward, N. Ismail, W. Darmalaksana, "IoT-Based UPS Monitoring System Using MQTT Protocols," *4th International Conference on Wireless and Telematics (ICWT)*, pp. 1-5, 2018.

[12] Comparison of MQTT Brokers, "https://tewarid.github.io/2019/03/21/comparison-of-mqtt-brokers.html," visited on Oct, 2020.

[13] M. Collina, G. E. Corazza, A. Vanelli-Coralli, "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST," *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*, pp. 36-41, 2012.

[14] A. Schmitt, F. Carlier, V. Renault, "Data Exchange with the MQTT Protocol: Dynamic Bridge Approach," *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pp. 1-5, 2019.

[15] A. M. Zambrano V, M. Zambrano V, E.L.O. Mejía, X. Calderón H, "SIGPRO: A Real-Time Progressive Notification System Using MQTT Bridges and Topic Hierarchy for Rapid Location of Missing Persons," i*n IEEE Access*, vol. 8, pp. 149190-149198, 2020.

[16] The features that various MQTT servers (brokers) support, "https://github.com/mqtt/mqtt.github.io/wiki/server-support," visited on Oct, 2020.

[17] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, D. Kobayashi, "A scalable and low-cost MQTT broker clustering system," *2017 2nd International Conference on Information Technology (INCIT)*, pp. 1-5, 2017.

[18] Z. Y. Thean, V. Voon Yap, P. C. Teh, "Container-based MQTT Broker Cluster for Edge Computing," *2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, pp. 1-6, 2019.

[19] A. Detti, L. Funari, N. Blefari-Melazzi, "Sub-Linear Scalability of MQTT Clusters in Topic-Based Publish-Subscribe Applications," *in IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1954-1968, 2020.

[20] M. H. Fourati, S. Marzouk, K. Drira and M. Jmaiel, "DOCKERANALYZER : Towards Fine Grained Resource Elasticity for Microservices-Based Applications Deployed with Docker," *20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 220-225, 2019.

[21] R.R. Righi, E, Correa, M.M. Gomes, C.A. Costa, "Enhancing performance of IoT applications with load prediction and cloud elasticity," *Future Generation Computer Systems*, Volume 109,P. 689-701, 2019.

[22] L.M. Pham, "Autonomic fine-grained migration and replication of component-based applications across multi-clouds," *in Proc. of 2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, pp. 5-10, 2015.

[23] M. Nardelli, V. Cardellini, E. Casalicchio, "Multi-Level Elastic Deployment of Containerized Applications in Geo-Distributed Environments," *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 1-8, 2018.

[24] V.F. Rodrigues, I.G. Wendt, R.R. Righi, C.A. Costa, J.L.V. Barbosa, A.M. Alberti, "Brokel: Towards enabling multi-level cloud elasticity on publish/subscribe brokers," *International Journal of Distributed Sensor Networks*, vol. 13, no. 8, 2017.

[25] S. Vavassori, J. Soriano, R. Fernández, "Enabling Large-Scale IoT-Based Services through Elastic Publish/Subscribe". *Sensors*, pp. 17-2148, 2017.

[26] A distributed, reliable key-value store, "https://etcd.io/docs/v3.4.0/," visited on Oct, 2020.

[27] D. Roure, C. Goble, "Software Design for Empowering Scientists," *IEEE Software*, vol. 26, no. 01, pp. 88-95, 2009.

[28] EMQX Broker, "https://docs.emqx.io/broker/latest/en/,' visited on Oct, 2020.

[29] Kubernetes, "https://kubernetes.io/," visited on Oct, 2020.

[30] HAProxy, "https://www.haproxy.com/solutions/load-balancing/," visited on Oct, 2020.

[31] OpenStack: Open Source Cloud Computing Infrastructure, "https://www.openstack.org/," visited on Oct, 2020.

[32] OpenStack Heat, "https://docs.openstack.org/heat/latest/," visited on Oct, 2020.

[33] OpenStack Ceilometer, " https://docs.openstack.org/ceilometer/latest/," visited on Oct, 2020.

[34] OpenStack Aodh, " https://docs.openstack.org/aodh/latest/," visited on Oct, 2020.

[35] Gnocchi - Metric as a Service, "https://gnocchi.xyz/," visited on Oct, 2020.

[36] RabbitMQ, "https://www.rabbitmq.com/," visited on Oct, 2020.

[37] Apache JMeter, " https://jmeter.apache.org/," visited on Oct, 2020.

[38] L.M. Pham, T.T. Nguyen, M.D. Tran, "A Benchmarking Tool for Elastic MQTT Brokers in IoT Applications," *International Journal of Information and Communication Sciences*. Vol. 4, No. 4, pp. 59-67, 2019

Linh Manh Pham is a lecturer at University of Engineering and Technology, Vietnam National University, Hanoi (VNU). He was a postdoctoral researcher at Inria, France. He earned an MSc. in Computer Science in the USA and a Ph.D. in Cloud Computing in France. His area of research is Cloud/Fog Computing, and he intends to highlight the benefits of this relatively novel field of research.
Email: linhmp@vnu.edu.vn.

Tien-Quang Hoang is with Hanoi Pedagogical University 2. He is also a researcher of Center of Multidisciplinary Integrated Technologies for Field Monitoring, University of Engineering and Technology, Vietnam National University, Hanoi (VNU-UET). He has a Master degree in Computer Networks and Data Communication at VNU-UET.
Email:hoangtienquang@hpu2.edu.vn.

Xuan-Truong Nguyen is with Hanoi Pedagogical University 2 as a lecturer. He is also a researcher of Center of Multidisciplinary Integrated Technologies for Field Monitoring, University of Engineering and Technology, Vietnam National University, Hanoi (VNU-UET). He has a Master degree in Software Engineering at VNU-UET.
Email: nguyenxuantruong@hpu2.edu.vn.