VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

# An Elasticity Framework for Distributed Message Queuing Telemetry Transport Brokers

Linh Manh Pham[*], Xuan Tung Hoang

*VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam*

Received 23 October 2020
Revised 25 February 2021; Accepted 27 February 2021

**Abstract:** Internet of Things (IoT) applications are increasingly making impact in all areas of human life. Day by day, its chatty embedded devices have been generating tons of data requiring effective network infrastructure. To deliver millions of IoT messages back and forth with as few faults as possible, participation of communication protocols like Message Queuing Telemetry Transport (i.e., MQTT) is a must. Lightweight blueprint and battery friendly design are just two of many advantages of this protocol making it become a dominant in IoT world. In real application scenarios, distributed MQTT solutions are usually required since centralized MQTT approach is incapable of dealing with huge amount of data. Although distributed MQTT solutions are scalable, they do not adapt to fluctuations of traffic workload. This might cost IoT service providers because of redundant computation resources. This leads to the need of a novel approach that can adapt its volume changes in workload. This article proposes such an elastic solution by proposing a flexible MQTT framework. Our MQTT framework uses off-the-shelf components to obtain server's elasticity while keeping IoT applications intact. Experiments are conducted to validate elasticity function provided by an implementation of our framework.

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

## 1. Introduction

It is a fact that Internet of Things (IoT) has been widely spreading in many domains with different scales from homes, enterprises, institutions to industries. Behind IoT application scenarios are millions of connected devices trying to communicate and deliver data throughout the Internet either with or without human interventions. At present, 31 billion devices are connected as IoT devices and it is predicted that by 2050 this number will surge pass 170 billion limit [1]. Also, an IoT network can hold up 50 to 100 trillion connected objects, and this network can track the movement of every single of objects. Each people living in urban areas can be surrounded by 1000 to 5000 tracking IoT things. In the same context, currently, there are about 4 billion people connected, more than 25 million applications,

[*] Corresponding author.
*Email address:* linhmp@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.267

more than 25 billion embedded and intelligent systems, which generate 50 trillion gigabytes of data. The IoT market can bring up to 4 trillion USD in revenue for its service providers [2].

To support the communication of billions of IoT devices and delivery of its huge generated data, the IoT service providers need to implement and maintain robust and scalable network infrastructures. Especially, when IoT applications cross the boundary of home-wide to reach the skyline of city- or country-wide systems, the number of IoT devices can increase extremely at an unpredictable rate. That is a point for development of not only brawny but also scalable IoT infrastructures. Such modern IoT infrastructures nowadays contain an essential component called Message Queuing Telemetry Transport (i.e., MQTT) servers (a.k.a. brokers). These brokers are implemented on MQTT protocol devised since 1999, which is an open Machine-to-Machine protocol (M2M) originally. It is also an openly industrial standard released by OASIS and ISO (ISO/IEC 20922) [3]. With its advantages such as lightweight blueprint, bandwidth-efficient design, low power consumption, or spatial/temporal decoupling, MQTT brokers are dominating the IoT world indisputably.

Although recent MQTT brokers implement both centralized and distributed approaches to be able to handle millions of connected clients in a short period of time, only a few one proposed solutions to keep up with fluctuation of workload generated by IoT clients. This might happen when number of IoT devices increase or decrease unpredictably during certain times. In reality, the city-wide IoT applications often deal with disperse and intermittent devices causing a change in the number of involving clients. For instance, smart cars often join given connected vehicle networks in the rush hours rather than regular hours, generating more IoT data within specific periods. This leads to the need for development of new MQTT systems that know not only how to scale but also keep pace with changes of the workload generated by the clients. In other words, an approach makes MQTT brokers elastic.

Elasticity is a native characteristic of Cloud computing according to NIST [4]. Thanks to elasticity, cloud resources are not overused or underused, which not only saves cloud providers' money but also improves customer experience. Many IoT applications have been being implemented on Cloud or ready to be moved on it. It is worth mentioning that IoT resources like MQTT brokers implemented on Cloud can also benefit from cloud elasticity.

In this article, we propose a framework making MQTT brokers elastic. To obtain this goal, the following contributions are made:

i) We propose a novel framework that can flexibly support elasticity while retain all features of MQTT protocol;

ii) We make a concrete implementation of the proposed framework using an open-source MQTT broker software (EMQX) and a private cloud platform (OpenStack);

iii) We conduct experiments to validate the soundness of the proposed approach using the open-source implementation aforementioned.

The rest of the article is organized as follows. After highlighting various related work in Section 2, we describe in detail the overall architecture of our proposed MQTT framework in Section 3. The validating experiments and results are reported in Section 4. Finally, we conclude in Section 5.

## 2. Related Work

### *2.1. Message Queuing Telemetry Transport Brokers*

MQTT is a Message Oriented Middleware (MOM), which follows publish/subscribe pattern (pub/sub for short) [5]. With lightweight design in mind, MQTT is suitable for many IoT applications where various restricting requirements must be satisfied such as low bandwidth, low energy, or intermittent sensor nodes. The pub/sub model brings to the decoupling in both space and time, where MQTT clients (i.e. publishers and subscribers) do not have knowledge about locations of each other and do not need to be present at the same time while sending or receiving the messages, respectively. This is possible because the protocol uses an intermediate service called MQTT brokers to mediate messages among participants (publishers and subscribers). While space decoupling characteristic helps IoT application separate its high volume of available data from the origin of data, time decoupling one is a necessity for IoT applications because of its distributed nature.

The communication mechanism of MQTT is relatively simple. First, the MQTT clients need to establish a connection to MQTT broker by sending a CONNECT message. A CONNACK message from the broker sent back to the client is to confirm for a successful connection. After that, the operations of publishing/subscribing messages to/from the broker can be done. The publisher needs to send a PUBLISH message containing a topic name. A topic (i.e. subject of interest) is a string used by broker, where subscribers register to it for getting copies of needed messages. To do that, the subscriber must send a SUBSCRIBE message containing its interesting topic to the broker. Topics can be organized in a hierarchical way (i.e. topic trie) to take advantage of wildcard filters such as "#" or "?". In general, the clients can publish/subscribe to more than one topic not using or using these wildcards for convenient.

In MQTT, the communication reliability can be obtained by specifying the levels of Quality of Service (QoS). There are three levels of QoS including "At most one" (0), "At least one" (1), and "Exactly one" (2). At level 0, the delivery is not acknowledged and the message is sent only once in any case. At level 1, if no acknowledgement is received by the publisher, it will try to resend the message multiple times. At level 2, exactly one copy of the message is received by the subscriber by a two-way handshake agreement between the publisher and subscriber. To guarantee no data is lost, IoT applications obviously need a lightweight and vigorous solution like the MQTT broker model. Some of the most widely used MQTT brokers so far are Mosquitto, HiveMQ, moquette, VerneMQ, EMQX, etc.

In recent decade, many IoT applications have implemented MQTT brokers such as [6-11]. A typical structure of an IoT application using MQTT brokers deployed with centralized datacenter remotely (like in the Cloud environment) is depicted in Figure 1. The goal of the application is to collect data from many IoT devices and sensors, then process and store these data, and at last send notifications and reports to the final users (using laptop, mobile, tablet, etc.). In some cases, the gathered data can be published directly to the topics subscribed by final users without any data analysis. Control commands can be published to the command topic in the broker like any other type of MQTT messages by the final users. These messages will be archived in the cloud storage and transmitted to the IoT devices or sensors by some scheduling mechanism. In the case of time-sensitive application, the command messages can also be sent directly to the IoT devices without travelling to the Cloud. We see that IoT devices, end-user interfaces, and data analyzing system are all MQTT clients producing and consuming telemetry data.

Many IoT applications often implement a centralized MQTT broker keeping all subscribed topics. However, the broker in this topology is easy to become a bottle neck of the entire system. To avoid this, some solutions have been proposed, which can be categorized into two types of distributed system: bridged brokers and clustered brokers. In the first model, two brokers can be bridged to be able to serve more messages from clients while keeping separation of both broker's locations. Published messages are forwarded from a broker to its bridged one according to specific access policy. A full-mesh network needs to be formed among brokers (i.e. each one pairs with all others) in order that any MQTT client can connect to any broker it wishes to. Therefore, using bridged model to obtain elasticity is too complex. It is only suitable for networks having a few MQTT brokers. MQTT brokers support the bridging method including HiveMQ, EMQX, JoramMQ, moquette, mosquitto, VerneMQ, etc. [12]. Some implementations of this model are reported in research of Collina et al., [13], Schmitt et al., [14], and Zambrano et al., [15]. MQTT brokers following the clustered model take advantage of subtopics in the hierarchical trie. One of the brokers (B0) keeps root topic and subtopics which its subscriptions involve. Other ones (B1, B2, etc.) only keep their involved subtopics originating from the root topic located at B0. The topic branches are dynamically created in a broker corresponding to MQTT subscriptions to this broker. Therefore, the overhead among the brokers is reduced significantly in comparison to the bridged model. Moreover, the knowledge of topic trie and route table are transferred among brokers, thus any MQTT client can connect/reconnect to any broker it wants to establish/resume its sessions. Not many MQTT brokers support full features of the clustered topology including EMQX, HiveMQ, RabbitMQ, VerneMQ [16]. Some research projects in this trend that can be mentioned are the work of Jutadhamakorn et al. [17], Thean et al., [18] and Detti et al., [19].

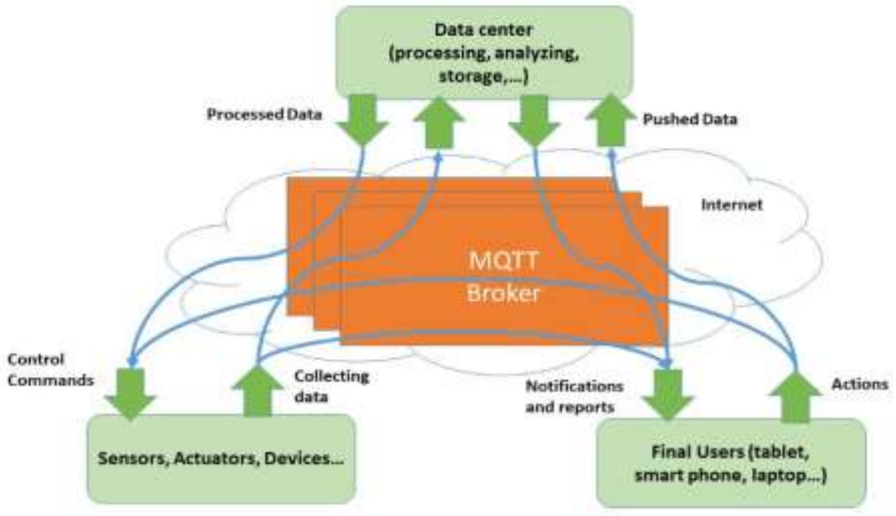

Data center (processing, analyzing, storage,...)
Processed Data
Pushed Data
Internet
MQTT Broker
Control Commands
Collecting data
Notifications and reports
Actions
Sensors, Actuators, Devices...
Final Users (tablet, smart phone, laptop...)

Figure 1. A typical structure of an IoT application using MQTT brokers.

## *2.2. Elastic Message Queuing Telemetry Transport Broker*

Elasticity defined as one of essential characteristic of Cloud Computing. Thanks to this special feature, cloud resources can be provisioned or released corresponding to demand. Nowadays, IoT applications are often installed in Cloud to take advantage of this environment like on-demand measured service, broad network access as well as rapid elasticity. Multiple solutions try to provide elasticity for other components of IoT applications rather than MQTT broker, some of which are Proliot [20], DOCKERANALYZER [21], ACD [22], BDAaaS [23].

Very few elastic solutions mentioning MQTT broker among various types of pub/sub servers have been proposed such as Brokel [24] and E-SilboPS [25]. Brokel defines a multi-level elasticity model for Pub/Sub brokers (including MQTT) in general, thus many MQTT-specific tweaks have been simplified or omitted. E-SilboP is an elastic content-based publish/subscribe system specifically designed to support context-aware sensing and communication in IoT-based services. Therefore, it also omits many adjustable QoS parameters of MQTT protocol and only provides content-based elasticity. It is worth mentioning that these both research works are solutions for pub/sub servers in general, which do not focus only on MQTT broker. One of the prerequisite to holistically obtain elastic MQTT is that the solution must implement one of the distributed topologies mentioned in the subsection 2.1.

## 3. Elastic Message Queuing Telemetry Transport Framework

This section introduces our new elastic MQTT framework. The framework is designed to have flexible architecture containing a set of representative modules. When an implementation of the framework is deployed on the Cloud, each of these representative ones will be specialized into a concrete component-off-the-shelf (COTS) one. Therefore, the modules of framework can be substituted flexibly to obtain new features, to earn enhanced performance, or to lower software licensing fees. We also present a concrete implementation of each of the modules constituting the framework. The implementation mainly targets for cloud-based IoT applications which require elasticity as an essential feature. These applications include, but not limited to, big data analytics, latency-sensitive ones. With the principle of software development serving the e-science community [27], we prefer combining the most pertinent open-source solutions into our framework. The overall novel architecture of the framework is depicted in Figure 2 composing of the following modules:

i) MQTT broker cluster: A cluster of MQTT brokers implementing distributed pub/sub model with customizable QoS parameters. The cluster consists of a number of runtime systems called node. Nodes connect to each other using TCP/IP sockets and communicate by message passing. Each node keeps its parts of topic tries and current subscriptions. This mechanism helps published messages be routed across the cluster from the first node receiving the messages to the last one delivering the messages to the subscribers. The nodes can join cluster manually or automatically. With automatic way, node discovery and autocluster mechanisms such as IP multicast, dynamic DNS, or ETCD [26] need to be supported. The nodes can be deployed on both public or private cloud networks. Public Cloud providers, such as AWS, Azure, or private Cloud platforms, such as OpenStack, CloudStack could be the good candidates.

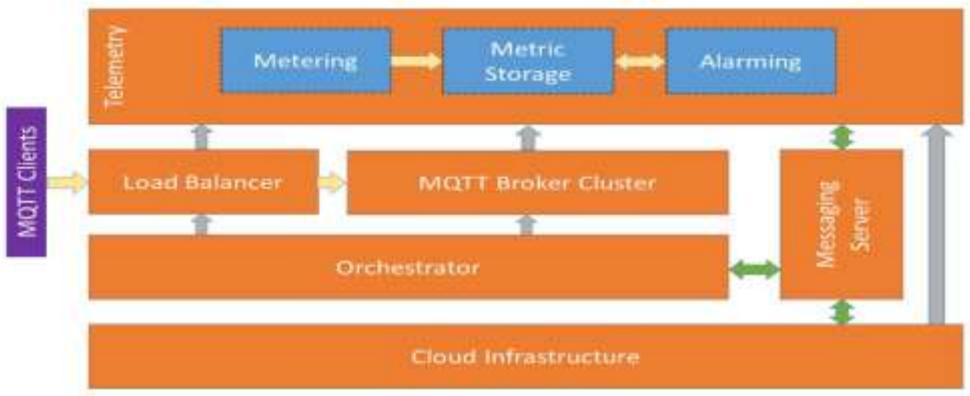


Figure 2. Architecture of Elastic MQTT Framework.

We choose EMQX [28] for our MQTT brokers. EMQX provides concurrent, fault-tolerant, and distributed broker nodes. It is one of few open-source MQTT solutions which offer clustered brokers. Moreover, EMQX is the only one implementing all three levels of MQTT QoS, MQTT protocol for regular networks, and MQTT-SN protocol for sensor ones. EMQX supports node discovery and autocluster with various strategies as in the case of IP multicast, dynamic DNS, etcd, and Kubernetes [29]. By that when a broker node arrives or leaves according to elastic actions, the cluster automatically recognizes the changes and updates its configuration to reflect new number of nodes;

ii) Load balancer: A Load Balancer (LB) is often deployed in front of a MQTT cluster to distribute MQTT connections and traffic from devices across the MQTT clusters. LB also enhances the high availability of the clusters, balances the loads among the cluster nodes, and makes the dynamic expansion possible. The links between the LB and cluster nodes are plain TCP connections. By this setup, a single MQTT cluster could serve millions of clients. Thanks to LB, MQTT clients only need to know one point of connection instead of maintaining a list of MQTT brokers;

Some commercial LB solutions are supported by EMQX such as AWS, Aliyun, or QingCloud. In the terms of open-source software, HAProxy [30] can serve as a LB for EMQX cluster and establish/terminate the TCP connections. Many dynamic scheduling algorithms can be assigned by HAProxy such as round robin, least connection, or randomness;

iii) Cloud infrastructure: It manages, provides, and releases dynamically virtual resources for gaining elasticity. To obtain "unlimited" resources, an implementation of private, public, or hybrid cloud may need to be carried out. To serve e-science community, OpenStack [31], an open-source private Cloud is chosen to provision and release virtual resources. With world-wide supported user community and large well-maintained services, OpenStack is a fit for our goal. Some specific OpenStack services deployed for our implementation are Nova, Keystone, Glance, Horizon, Swift, and Neutron. Since we chose OpenStack cloud, the following modules should deploy services supported officially by OpenStack;

iv) Orchestrator: This module parses a system-component description in its own high-level domain specific language (i.e. DSL) and then deploys, manages, and monitors the entire life cycle of all involving components. Those components include resources such as virtual machines, containers, images, security groups, alarms, scaling policies, etc. The grammar of the DSL can be derived from XML, JSON, or YAML. The main motivation is to keep the thing simple and user-friendly. In the framework, the Orchestrator deploys and manages MQTT brokers as well as resources of the elastic decision-making block such as Metering, Metric Storage, and Alarming.

The main orchestrator supported by OpenStack is Heat service [32] The infrastructure for a cloud application is described in a Heat template file. Infrastructure resources that can be described including servers, volumes, users, security groups, floating IPs, etc. Heat also provides an autoscaling service integrating with sub-modules of Telemetry, so a scaling group can be included as a resource in the template. This is a perfect fit for our elasticity goal. Templates can also delineate the dependencies between resources (e.g., this floating IP is assigned to this VM). This helps Heat to create all of managed components in the correct order for completely launching application. Heat manages the entire life cycle of the application and it knows how to make the necessarily dynamic changes. Finally, it also takes care of deletion of all the deployed resources when the application accomplishes;

v) Telemetry: This module consists of three sub-modules

Metering: This module's goal is to efficiently collect, normalize, and transform data produced by orchestrated components. These data are intended to be used to create different views and help solve various telemetry use cases. Among them, data of specific metrics (i.e., measures) is collected and analyzed for elasticity triggering goal. Alarming and Metric storage are two modules which directly exploit these measures.

Alarming: Its goal is to enable the ability of triggering responsive actions based on defined rules against sample or event data collected by Metering module. It consists of two main sub-modules: "Alarm Evaluator" and "Alarm

Notifier". The former evaluates measures of a given metric stored in Metric storage module whether they are over or under a threshold. The latter then triggers a notification and sends to the Orchestrator who will perform corresponding elastic actions such as scaling out or in.

Metric storage: This database mainly stores aggregated measures of cluster nodes such as system performance metrics. The metric is a list of (timestamp, value) for a given managed resource. The resource can be anything from the temperature of the nodes to the CPU usage of a VM. Besides, the database also stores events, that is a list of things that happens in Cloud infrastructure: an API request has been received, a VM has been started, an image has been uploaded, whatever. Stored measures are retrieved for monitoring, billing, or alarming, where events are useful to do audit, performance analysis, debugging, etc.

Correspondingly, OpenStack supports some official services for Telemetry module, including Ceilometer [33] for Metering, Aodh [34] for Alarm, and Gnocchi [35] for Metric Storage;

vi) Messaging server: It is needed for communication between framework's modules by exchanging messages. It creates connected channels using favoured communicating protocols such as AMQP, CoAP, or even MQTT. In OpenStack cloud, internal communication among OpenStack services may be conducted by RabbitMQ [36]. RabbitMQ is an open-source message-oriented middleware supporting popularly communicating protocols such as AMQP, STOMP, and MQTT.

All modules of the framework are decoupling. It means the startup order is not quite important. In spite of that, it makes no sense for some modules to work independently, thus requires the power-up of other ones as prerequisites. Similarly, components and resources described and managed by the Orchestrator should be initiated at any given moment. The Orchestrator must have ability to resolve dependencies between components and from there come up with a deployment plan containing the appropriate order of installment. From the system description to the deployment plan, the Orchestrator needs to use a chain of solvers such as Learning Automata based allocator, Constraint Programming based solvers, Heuristics based solvers, and Meta-Solvers. When an event or combination of events and conditions occur at runtime, the Orchestrator generates the corresponding elasticity plan and conducts the necessary modifications to convert the current topology to the expected one described in the elasticity plan. The modifications include actions following ECA (event-condition-action) rule such as resources' scaling in/out or up/down when measures of a resource trespass the given thresholds. Figure 3 depicts one possible implementation of our framework using the aforementioned open-source solutions.

## 4. Validating Experiments

In order to validate functionalities of our proposed framework, we conducted the implementation mentioned in Section 3 in our homegrown infrastructure at VNU University of Engineering and Technology (VNU-UET). We also make some discussions after the results of the experiments.

### *4.1. Experiment Testbed*

The testbed consists of two main parts: one implementation of our proposed framework, and one load injector for simulating multiple MQTT clients and their workloads. The tester creates test plans with different scenarios using built-in functions of the load injector. The simulated publishers generate MQTT messages and send them to the Load Balancer (HAProxy). In turn, the LB distributes these messages to one EMQX broker of the cluster according to scheduling algorithm. The simulated subscribers make MQTT subscriptions to specific topics in the cluster by connecting to the LB, too. The LB also distributes connecting requests of the subscribers to one of the members of the cluster. The routing messages from the source to the right destinations is conducted internally by the cluster as mentioned in Section 3.

We used Apache JMeter [37] as the load injector in our experiments. It is an open-source tools for load test and performance evaluation. It supports testing of many different protocol types such as HTTP, HTTPS, SOAP, REST, FTP, JMS, etc. Other protocols are included into JMeter using plugins. To support the experimentation, we have developed a MQTT plugin for JMeter implementing some features of MQTT version 5.0 [38]. To do stress test, we used distributed testing paradigm with one JMeter master and a couple of slaves to ensure that there is no side-effect to the performance of simulated MQTT clients.

Our private OpenStack cloud is installed in the data center for research at VNU-UET. EMQX brokers and JMeter load injectors are virtual machines provisioned by the cloud. Each EMQX broker instance has 2 vCPU and 2 GB memory, and each JMeter virtual machines has 8 vCPU and 8 GB memory. We use OpenStack Train, which was released on 2019. Our OpenStack cloud is built on 3 physical servers using Intel processors. Each physical server has 80 CPUs at 2.4 GHz, 256 GB memory and a storage pool of 1.5 TB. CentOS 7 are installed on all physical machines as host operating system. On top of that, KVM is used as a base virtualization solution. For better resource isolation, instead of running OpenStack controller services (e.g. NovaAPI, NeutronServers, Keystone, etc.) directly on physical servers, we install those components on dedicated virtual server instances. Only hypervisor service (a.k.a. OpenStack NovaCompute) that takes care of running virtual instances, will be installed directly on physical servers. By this way, system services of OpenStack cloud itself will be completely separated from virtual server instances created by users of the cloud. In short, stress tests conducted by our experiments, which are running on OpenStack's virtual servers will not affect performance of the cloud and vice versa.

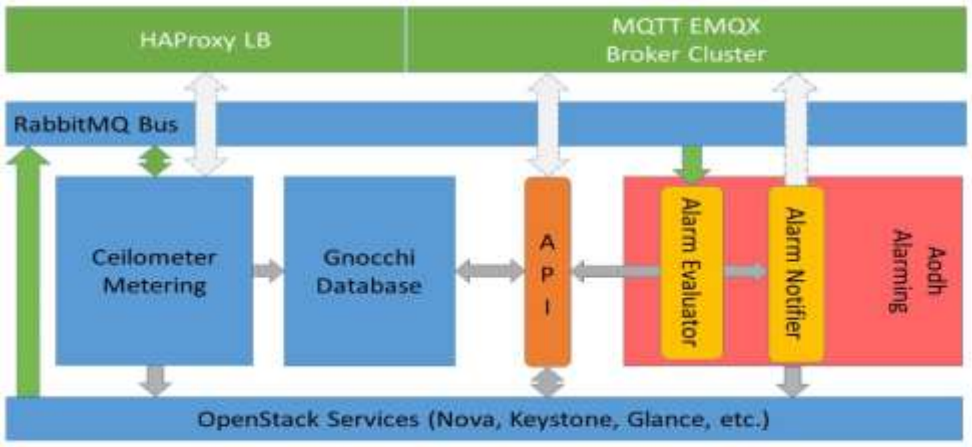


Figure 3. An implementation of Elastic MQTT Framework.

### *4.2. Experiment Scenarios*

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

#### 4.2.1. Multi-publisher scenario

This scenario simulates a huge number of IoT devices, for example smart plugs, publishing telemetry data to a central smart-home system. The devices are the publishers and the central smart-home system is the subscriber. Devices are structured as a three-level tree. The top level represents the smart houses in a district. The middle level is called households in a smart house. The access point in each household enables the smart plugs to reach Internet and publish data to the smart-home center. The bottom level is the smart plugs who send to the access point measures of devices plugged into them. The three-level tree is mapped to MQTT topics. A topic level is added below the device to represent the telemetry parameters, for instance the power consumption (kWh). The test scenario defines 40 topic partitions made of:

i) 1 root topic “SmartHouse”;

ii) 3 topics “Household” each smart house;

iii) 30 topics “SmartPlug” each household 10 topics “Parameter” each smart plug.

Therefore, a topic partition represents 90 smart plugs and each one publishes 10 telemetry parameters. We have 3600 smart plugs totally in the scenario. The testbed for this scenario is depicted in Figure 4.

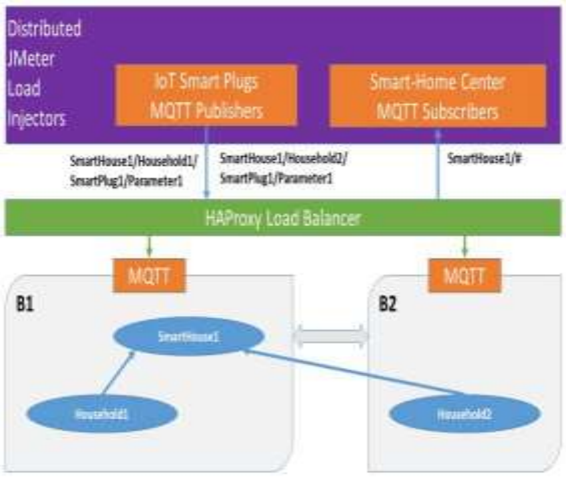

Distributed JMeter Load Injectors
IoT Smart Plugs MQTT Publishers
Smart-Home Center MQTT Subscribers
SmartHouse1/Household1/SmartPlug1/Parameter1
SmartHouse1/Household2/SmartPlug1/Parameter1
SmartHouse1/#
HAProxy Load Balancer
MQTT
MQTT
B1
B2
SmartHouse1
Household1
Household2

Figure 4. Multi-Publisher Clustered Brokers Scenario.

4.2.2. Multi-subscriber scenario

This scenario also simulates a huge number of smart plugs, controlled by a central smart-home system. The smart plugs are the subscribers and the central system is the publisher. Smart plugs provide two-way communications. The final users and smart-home center can send commands to the plugs. Besides, the smart-plugs can also respond to these commands, for example an indicator of an ON/OFF update. Topics are partitioned in a couple of levels in the same way as in the multi-publisher scenario. We also define topic partitions representing 3600 smart plugs providing a control interface, which each partition is composed of:

i) 1 root topic “SmartHouse”;

ii) 3 topics “Household” each smart house;

iii) 30 topics “SmartPlug” each household;

iv) 1 topic “Command” each smart plug.

The testbed for this scenario is depicted in Figure 5.

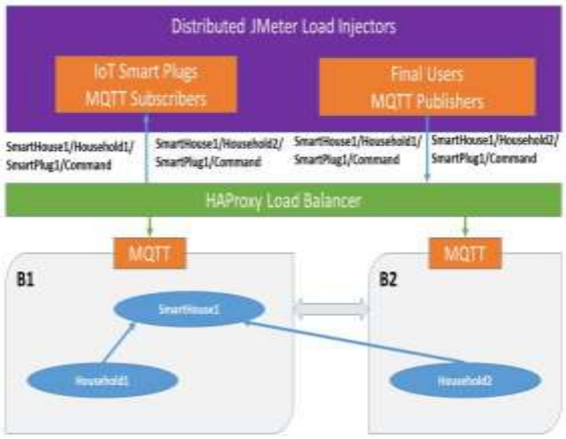

Distributed JMeter Load Injectors
IoT Smart Plugs MQTT Subscribers
Final Users MQTT Publishers
SmartHouse1/Household1/SmartPlug1/Command
SmartHouse1/Household2/SmartPlug1/Command
SmartHouse1/Household1/SmartPlug1/Command
SmartHouse1/Household2/SmartPlug1/Command
HAProxy Load Balancer
MQTT
MQTT
B1
B2
SmartHouse1
Household1
Household2

Figure 5. Multi-Subscriber Clustered Brokers Scenario.

## *4.3. Results*

The MQTT workloads are prepared using JMeter test plan. The workload starts with a short warm-up period and then dramatically increases when MQTT clients joins steadily to the simulation. EMQX servers are preconfigured following suggestions from EMQX documentation[1]. We chose IP multicast method for the node-discovery and autocluster mechanisms. The scheduling strategy for HAProxy was set to roundrobin.

The Ceilometer, Aodh, and Gnocchi were configured to measure and store measurements of average %CPU utilization and number of virtual CPUs (vCPU) metrics. Upper and lower thresholds for average %CPU utilization are set

[1] https://docs.emqx.io/en/broker/v3.0/tune.html.

to 80% and 25% respectively. It means that if average %CPU utilization breaks these 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, maximum of 5 JVM processes are allowed to initiate. According to the test scenarios, each process is responsible for running 3600 MQTT clients. Therefore, 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.

### 4.3.1. 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". Totally, 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 B0 and B1 (B1 will be added dynamically when needed). Publishers (90 each partition) and subscribers (one each partition) are equally load-balanced across the two brokers.

Figure 6a shows 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 to resolve the problem. In the centralized case, we see obviously in Figure 7a that average %CPU utilization of the broker gets saturation after a couple of minutes (at the 1st minute). At this point, 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 8a, we see an elasticity effort to mitigate the pressure performed by our system. One virtual machine of MQTT broker B1 is created to share the workload. This broker automatically joins the cluster created beforehand by B0 using 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, 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%, number of VMs is decreased to one for minimizing operating cost.

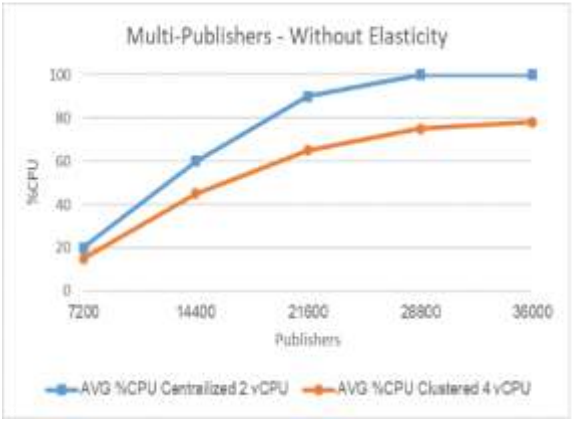
### 4.3.2. Multi-subscriber scenario

In centralized case, there is only one publisher each topic partition. This publisher sends messages to all the topics of the partition at a steady rate. One subscriber is created for every topic "SmartPlug". Each subscriber receives messages from the topic "Command" under the topic "SmartPlug". In all partitions, 3600 subscribers receive messages from 3600 topics "Command" at a steady rate.

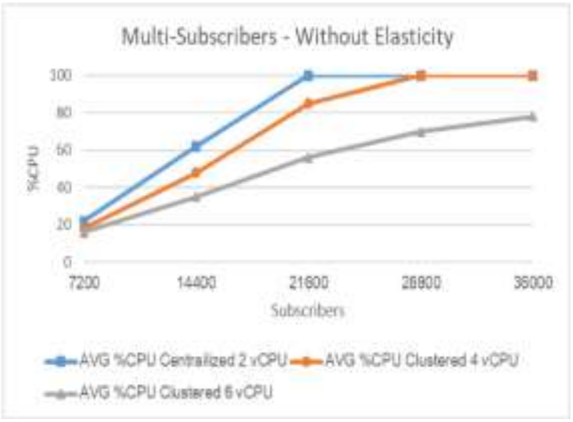
In the clustered case, the multi-subscriber scenario is tested with a cluster of two MQTT brokers B0 and B1 (B1 is added dynamically when needed). Subscribers (90 each partition) and publishers (one each partition) are equally load-balanced across the two brokers B0 and B1.

Figure 6b shows average % CPU utilization in both centralized and clustered cases without elasticity. We see the same behaviors like the case of multiple publishers. Adding one more broker to the cluster does not really help, but two more brokers (6 vCPU) can resolve the problem. In the centralized case, we also see obviously in Figure 7b that average %CPU utilization of the broker gets saturation after a couple of minutes (at the 2nd minute). At this point, dropped message rate also starts to increase.

With elasticity, we also see the same behaviours shown in Figure 8b like in the case of multiple publishers. The scaling out action with two more brokers is triggered later than the multi-publisher scenario. These two brokers are added sequentially by Heat. One gap of one minute is set between broker additions to avoid elastic oscillation. The average %CPU utilization stays above the upper threshold during the time longer than in the multi-publisher scenario. The reason is that the combination of QoS level set to 2 and "clean session" flag set to FALSE keep retained messages at the brokers longer, thus the more the subscribers are, the busier the brokers are.

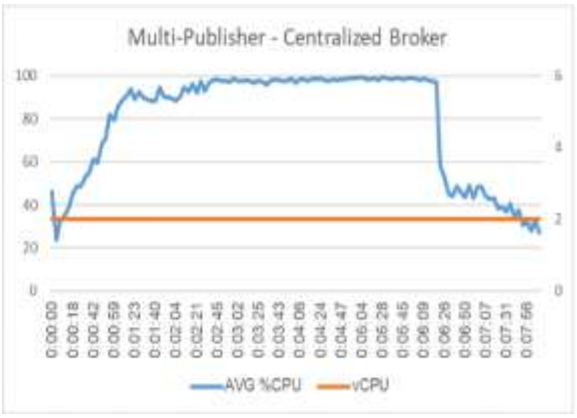


(a) Multi-Publisher Scenario.

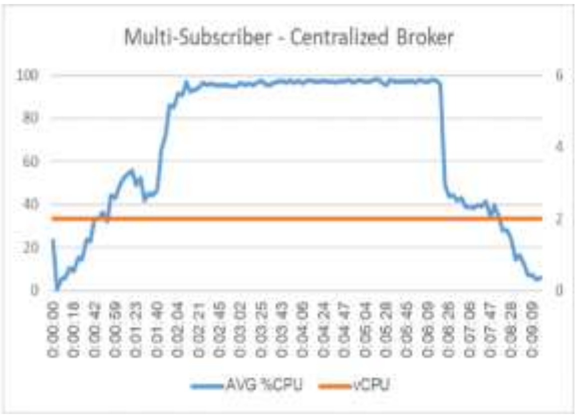


(b) Multi-Subscriber Scenario.

Figure 6. Without Elasticity: Average %CPU Usage of the Centralized and Clustered Brokers.

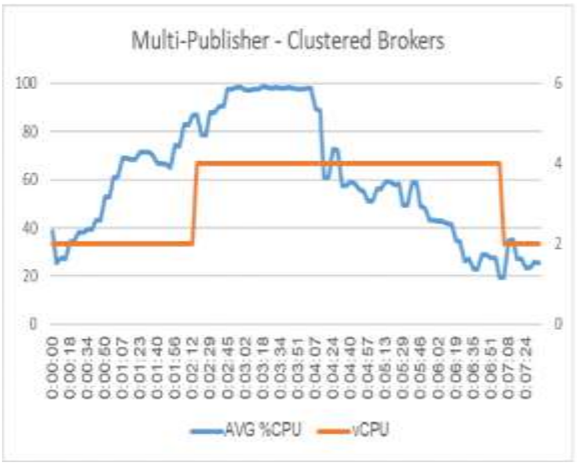


(a) Multi-Publisher Scenario.

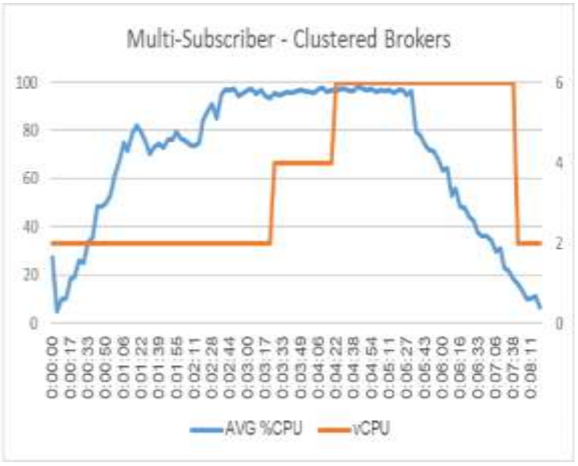


(b) Multi-Subscriber Scenario.

Figure 7. Average %CPU Usage of the Centralized Broker during Experimental Time.



(a) Multi-Publisher Scenario.



(b) Multi-Subscriber Scenario.

Figure 8. Average %CPU Usage of the Clustered Brokers with Elasticity.

## 5. Conclusion

We have presented a flexible framework that can support elasticity for MQTT broker service in IoT applications. Our framework brings elasticity to the service by leveraging existing off-the-shelf components that are currently used in cloud environments. Our elastic MQTT broker service has been successfully implemented using EMQX as MQTT broker solution and OpenStack as cloud environment. Experiments are conducted by generating traffics to the service at varying load level to observe changes in number of broker instances. Our experiment results show that our MQTT broker service adapts relative well to user load changes making the service fully accommodate incoming traffics as well as keep operating cost low.

**Acknowledgements**

This work has been supported by VNU University of Engineering and Technology under project number CN19.09. We also send sincere thanks to the staff of Center for Computer Network and eLearning, VNU-UET for supporting the implementation of project's infrastructure.

## References

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

[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, IDC Report Apr, 2014.

[3] MQ Telemetry Transport, http://mqtt.org/, 2020 (accessed on: October 30th, 2020).

[4] P. Mell, T. Grance, The NIST Definition of Cloud Computing (Draft), NIST Special Publication, Vol. 800, No. 145, 2011, pp. 1-3.

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, A. Kermarrec, The many Faces of Publish/subscribe, ACM Comput, Surv, Vol. 35, No. 2, 2003, pp. 114-131.

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

[7] V. Gupta, S. Khera, N. Turk, MQTT Protocol Employing IOT Based Home Safety System with ABE Encryption, Multimed Tools Appl, 2020.

[8] A. Mukambikeshwari, Poojary, Smart Watering System Using MQTT Protocol in IoT, Advances

in Artificial Intelligence and Data Engineering, Advances in Intelligent Systems and Computing, Springer, Singapore, 2020.

[9] Y. C. See, E. X. Ho, IoT-Based Fire Safety System Using MQTT Communication Protocol, International Journal of Integrated Engineering, Vol. 12, No. 6, 2020, pp. 207-215.

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

[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), Nusa Dua, 2018, pp. 1-5.

[12] Comparison of MQTT Brokers, https://tewarid.github.io/2019/03/21/comparison-of-mqtt-brokers.html"/, 2020 (accessed on: October 30th, 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), Sydney, NSW, 2012, pp. 36-41.

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

[15] A. M. V. Zambrano, M. V. Zambrano, E. L. O. Mej´ıa, X. H. Calderon, SIGPRO: A Real-Time Progressive Notification System Using MQTT Bridges and Topic Hierarchy for Rapid Location of Missing Persons, in IEEE Access, Vol. 8, 2020, pp. 149190-149198.

[16] The features that Various MQTT Servers (Brokers) Support, https://github.com/mqtt/mqtt.github.io/wiki/server-support"/, 2020 (accessed on: October 30th, 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), Nakhonpathom, 2017, pp. 1-5.

[18] Z. Y. Thean, V. V. 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), Kedah, Malaysia, 2019, pp. 1-6.

[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, 2020, pp. 1954-1968.

[20] 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, Vol. 109, 2020, pp. 689-701.

[21] M. H. Fourati, S. Marzouk, K. Drira, 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), Gold Coast, Australia, 2019, pp. 220-225.

[22] 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), Barcelona, 2018, pp. 1-8.

[23] L. M. Pham, A Big Data Analytics Framework for IoT Applications in the Cloud, VNU Journal of Science: Computer Science and Communication Engineering, Vol. 31, No. 2, 2015, pp. 44-55.

[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, pp. 1-20.

[25] S. Vavassori, J. Soriano, R. Fernandez, Enabling Large-Scale IoT-Based Services Through Elastic Publish/Subscribe, Sensors, 2017.

[26] A Distributed, Reliable Key-value Store, https://etcd.io/docs/v3.4.0/, 2020 (accessed on: October 30th, 2020).

[27] D. Roure, C. Goble, Software Design for Empowering Scientists, IEEE Software, Vol. 26, No. 1, 2009, pp. 88-95.

[28] EMQX Broker, https://docs.emqx.io/broker/latest/en/, 2020 (accessed on: October 30th, 2020).

[29] Kubernetes, https://kubernetes.io/, 2020 (accessed on: October 30th, 2020).

[30] HAProxy, https://www.haproxy.com/solutions/load-balancing/, 2020 (accessed on: October 30th, 2020).

[31] OpenStack: Open Source Cloud Computing Infrastructure, https://www.openstack.org/, 2020 (accessed on: October 30th, 2020).

[32] OpenStack Heat, https://docs.openstack.org/heat/latest/, 2020 (accessed on: October 30th, 2020).

[33] OpenStack Ceilometer, https://docs.openstack.org/ceilometer/latest/, 2020 (accessed on: October 30th, 2020).

[34] OpenStack Aodh, https://docs.openstack.org/aodh/latest/, 2020 (accessed on: October 30th, 2020).

[35] Gnocchi - Metric as a Service, https://gnocchi.xyz/, 2020 (accessed on: October 30th, 2020).

[36] RabbitMQ, https://www.rabbitmq.com/, 2020 (accessed on: October 30th, 2020).

[37] Apache Jmeter, https://jmeter.apache.org/, 2020 (accessed on: October 30th, 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 Sciencesm, Vol. 4, No. 4, 2019, pp. 59-67.