Cluster Computing
https://doi.org/10.1007/s10586-022-03636-w

Multi-level just-enough elasticity for MQTT brokers of Internet of Things applications

Linh Manh Pham[1] · Nguyen-Tuan-Thanh Le[2] · Xuan-Truong Nguyen[1,3]

Received: 4 May 2021 / Revised: 2 May 2022 / Accepted: 18 May 2022

Abstract
Applications for the Internet of Things (IoT) are rapidly having an impact on all areas of daily life. Every day, its embedded devices generate loads of data that requires efficient network infrastructure. The integration of lightweight communication protocols such as Message Queuing Telemetry Transport (MQTT) is to send millions of IoT messages back and forth with as few errors as possible. In practice, IoT big data analytic systems are often deployed with highly regarded MQTT solutions to handle huge amounts of dynamic data and achieve scalability. However, these solutions do not adapt well to fluctuations in workload, so they are not elastic yet. This article introduces a novel framework that provides just-enough elasticity for MQTT brokers with multiple levels of virtualization and its implementation using EMQX MQTT broker, Kubernetes container-orchestration system and OpenStack cloud environment. Various experiments based on a real life IoT application are conducted to validate our proposed framework and its elastic functionality.

Keywords Elasticity · MQTT broker · Internet of Things · Cloud computing · Smart homes

1 Introduction

In the Fourth Industrial Revolution or Industry 4.0 era, Internet of Things (IoT), where billions of devices connect and communicate with each other, has been changing all aspects, with different scales, of our life. IoT helps to deliver (big) data through the Internet either with or without human interventions. In 2020, 31 billion devices are connected as IoT devices and it is predicted that by 2050 this number will surge pass 170 billion limit [1]. In addition, an IoT network can hold up 50 to 100 trillion of connected objects, and it can track the movement of every single object. As an estimation, each person living in urban areas nowadays can be surrounded by 1000 to 5000 tracking devices. In the same context, currently, there are about 4 billion people connected, more than 25 million applications, more than 25 billion embedded and intelligent systems, which generate 50 trillion gigabytes of data [2]. The IoT market can bring up to 4 trillion USD in revenue for its service providers [2].

Maintaining the communication among such huge number of IoT devices and handling the data explosion are difficult tasks. Especially, when IoT applications cross the boundary of home-wide scale to reach the skyline of city or country-wide systems, the number of things can extremely increase at an unpredictable rate. Therefore, IoT service providers must deploy a robust and scalable network infrastructure. Nowadays, the modern IoT infrastructures contain an essential component called Message Queuing Telemetry Transport (MQTT) servers or brokers. These brokers implement MQTT protocol, an open Machine-to-Machine (M2M) protocol devised since 1999 and released by OASIS and ISO (ISO/IEC 20922) [3] as an industrial standard. Due to its advantages such as lightweight blue-print, bandwidth-efficient design, less consumed energy, or

✉ Linh Manh Pham
linhmp@vnu.edu.vn

Nguyen-Tuan-Thanh Le
thanhlnt@tlu.edu.vn

Xuan-Truong Nguyen
nguyenxuantruong@hpu2.edu.vn

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

[2] Thuyloi University, 175 Tay Son, Dong Da, Hanoi, Vietnam

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

Published online: 06 June 2022

spatial/temporal decoupling, MQTT brokers are dominating the IoT world indisputably.

Recent MQTT brokers implement both centralized and distributed approaches and have the capability of handling millions of connected clients in a short period of time. However, only few of them keep up with fluctuation of workload, caused by unpredictable increase or decrease of number of IoT devices at certain time. In reality, a city-wide IoT application often have to deal with the issue of dispersed and intermittent devices, that causes a change in the number of clients. For instance, smart cars often connect a vehicle network in the rush hours rather than regular ones, and therefore generate more data within these periods. This requirement leads to the need of developing novel MQTT systems that not only have the scalability but also able to keep pace with the change of workload, generated by billions of IoT devices. In other words, we need a new approach to make MQTT brokers elastic.

Elasticity is a native characteristic of cloud computing, according to NIST [4]. Thanks to it, cloud resources are not overused or underused, and therefore not only saves provider money but also improves customer experience. Nowadays, many IoT applications have been being deployed on cloud or ready to move on it. Therefore, IoT resources, such as MQTT brokers, can also be implemented on cloud to benefit its elasticity feature.

Traditionally, elasticity in the cloud often have to deal with "fatty" virtual machines (VM) that have elusive virtualization overhead and leave resource holes. Resource holes, as the result of scaling-out VMs with a fixed configuration (i.e., flavor), may result in overprovisioning resources that do not really match the expected amount of resources. However, the redundant resource is sometimes not usable because its capacity might be less than the amount needed for the target application. An effective algorithm, to reduce the resource holes, need to be proposed to obtain just-enough elasticity.

In this article, we propose a novel framework to make MQTT brokers elastic. As the result, our contributions are following:

- A novel framework providing multi-level just-enough elasticity while retains all features of MQTT protocol.
- A concrete implementation of our proposed framework using an open-source MQTT broker software (EMQX), a container-orchestration system (Kubernetes), and a private cloud platform (OpenStack).
- The experiments are conducted to validate the soundness of our proposed approach using the open-source implementation aforementioned.

The rest of this article is organized as follows. In Sect. 2, the background of MQTT is provided. In Sect. 3, we highlight related work. Then, we describe in detail the overall architecture of our proposed MQTT framework in Sect. 4. The multi-level just-enough elasticity is presented in Sect. 5. Next, the experiments and results are reported in Sect. 6. Finally, we conclude in Sect. 7.

2 Preliminary of MQTT

2.1 MQTT

MQTT is a lightweight messaging protocol designed for communication between devices and computer systems [5]. Due to its lightweight design, MQTT is suitable for a wide variety of IoT applications where different restriction requirements must be satisfied such as low bandwidth, low energy, or intermittent sensor nodes. MQTT follows client/server mechanism, where each device is a client and connects to a server, which can be understood as a broker, through TCP (i.e., Transmission Control Protocol). Broker is responsible for coordinating all messages from the senders to the correct receiving side. MQTT's high-level architecture consists of two main parts: Broker and Clients. In particular, the broker is considered as the center, it is the intersection of all connections coming from the clients. The broker's main task is to receive message from publishers, line messages in queues, and then transfer them to a specific address. The sub-task of the broker is that it can take on a few more features related to the communication process such as message security, message storage, logs. 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. Both types of decoupling are supported by MQTT.

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 using these wildcards for convenient.

In MQTT, the communication reliability can be obtained by specifying the levels of Quality of Service (QoS). The Quality of Service (QoS) level is an agreement

between the sender and the receiver of a message that defines the guarantee of delivery for a specific message. There are three QoS levels in MQTT. At level 0, the delivery is not acknowledged and the message is sent only once in any cases. The recipient does not acknowledge receipt of the message and the message is not stored and retransmitted by the sender. QoS level 0 is often called "fire and forget" and provides the same guarantee as the underlying TCP protocol. At level 1, if no acknowledgement is received by the publisher, it will retry to send the message multiple times. At level 2, exactly one copy of the message is received by the subscriber by a four-part handshake agreement between the publisher and subscriber. It is the safest and slowest quality of service level. To ensure that no data is lost, IoT applications clearly need a lightweight and powerful solution like the MQTT broker model. Some of the most widely used MQTT brokers by far are Mosquitto, HiveMQ, VerneMQ, Moquette, EMQX [6], etc.

In the last decade, many IoT applications have deployed MQTT brokers such as [7–12]. A common structure of an IoT application implementing MQTT brokers deployed with a remote data center, for example, in a cloud environment, described in Fig. 1. The goal of the application is to collect data from multiple IoT devices and sensors, then process and store these data, ultimately it sends notifications and reports to end users (using laptops, mobile devices, tablets, etc.). In some cases, the data collected does not require analysis but can still be published directly to topics that have been registered by end users. Terminal users can publish control commands to *commands* topics in the broker just like any other type of MQTT message. These notifications will be stored in the cloud repository and transmitted to IoT devices or sensors under a number of scheduled mechanisms. In the case of time-sensitive applications, *command* messages may not need to go through the Cloud but will be sent directly to IoT devices. We found that the end-user interface and data analysis

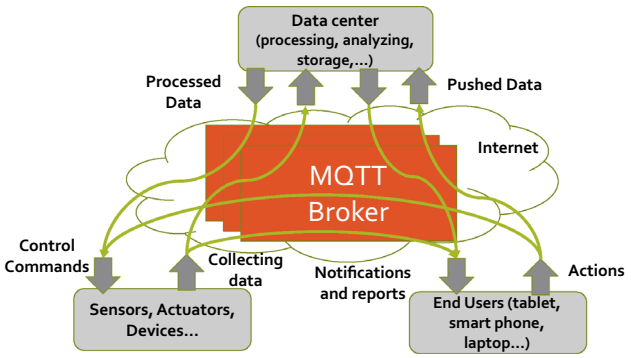


Fig. 1 A common structure of an IoT application implementing MQTT brokers

system, IoT devices are all MQTT clients producing and using remote measurement data.

2.2 Distributed MQTT Brokers

Some IoT applications often deploy a centralized MQTT broker to maintain all registered topics. However, brokers in this model are easy to become the bottlenecks of the whole system. In order not to encounter this, several solutions have been recommended, which can be divided into two types of distributing systems: the bridged brokers and the clustered brokers. With the former, two brokers could directly send more messages from clients whose locations remained segregated. Published notices are forwarded from a broker to a broker through its bridge under a specific access policy. A full network needs to be formed between brokers (i.e., a broker who can communicate with all other brokers) so that any MQTT client can connect with any broker it wants. Therefore, the use of the bridging model to get elasticity is very complicated. It is only suitable for networks that have a few MQTT brokers. Some MQTT brokers support the bridge model including HiveMQ, EMQX, JoramMQ, Moquette, Mosquitto, VerneMQ, etc. [13]. Some implementations of this model have been reported in the work of Schmitt et al. [14], and Zambrano et al. [15].

In the clustered model, one of the brokers (B0) keeps the original topic and the subtopics to which its subscriptions are associated. The other brokers (B1, B2, etc.) keep only related subtopics derived from the original topic at B0. Topic branches dynamically generated in a broker correspond to MQTT subscriptions for this broker. Consequently, the communication cost between brokers decreases relative to the bridging model. Furthermore, the knowledge of topic trie and routing table are transferred between brokers, so any MQTT client can connect/reconnect with any broker to set up/resume the session. There are very few MQTT brokers who fully support the clustering model including EMQX, HiveMQ, RabbitMQ, VerneMQ [16]. Some studies following this trend can be mentioned such as the work of Jutadhamakorn et al. [17], Thean et al. [18], and Detti et al. [19].

3 Related work

3.1 Elastic MQTT broker

The term elasticity are one of the essential characteristics of cloud computing [20]. With this special feature, cloud resources can be supplied or released corresponding to demand. Today, IoT applications are often deployed in the cloud to take advantage of this environment such as on-

demand measured services, broad network access, and rapid elasticity. Some solutions attempt to provide elasticity components of IoT applications, including Proliot [21], DOCKERANALYZER [22], ACD [23]. Nevertheless, very few elastic solutions for the MQTT broker have been proposed such as Brokel [24], E-SilboPS [25]. Brokel defines a multi-level elasticity model for Pub/Sub brokers (including MQTT) in general, so many edits dedicated to MQTT have been simplified or ignored. E-SilboP is a resilient content-based publishing/subscribing system specifically designed to support context-aware sensing and communication in IoT-based services. Therefore, many of the MQTT protocol's adjustable QoS parameters are ignored and it provides only content-based elasticity. These two solutions implement one of the distributed model mentioned in the Sect. 2.2. Our framework also follows the distributed model but focuses only on MQTT protocol, allowing customization of many MQTT-specific QoS parameters.

3.2 Multi-level elasticity for MQTT broker

A container is a standard unit of software that encapsulates source code of an application and its dependencies so that this application can operate quickly and reliably from one environment to another one. Instead of using hardware virtualization like virtual machines, containerization is a virtualization approach at the operating system level, which allows multiple containers to run directly on the operating system kernel. Containers are lightweight because they share the same operating system kernel, boot much faster, and use a portion of memory compared to loading the entire operating system. These advantages allow container-based applications to be deployed easily and consistently [26], regardless of the deployment environment as private data-center, public cloud, or even the developer's personal laptop.

Containerization provides a clear separation of production concerns. Now, developers focus on application logic and dependencies only, while IT operation teams can simply focus on deployment and management without concern for application details such as software versions and configurations specific to the application. Currently, there are some popular types of containers such as Docker, LXC, OpenVZ as well as some platforms, cluster management tools of these types such as Docker Swarm, Kubernetes, Apache Mesos, etc.

While a virtual machine is often considered as a coarse-grained resource [27], a virtualized container is seen as a more fine-grained one. The combination of these two types of virtualization reduces the possibility of creating resource holes, therefore it helps to realize just-enough elasticity. Although the multi-level elasticity is already implemented in some research projects involving data stream processing [28, 29] or cloud services [30], to the best of our knowledge, this is the first time it is mentioned for MQTT brokers.

4 Proposed elastic MQTT framework

In this section, we introduce a novel MQTT framework providing multi-level just-enough elasticity. 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 modules 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, earn enhanced performance, or 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 [31], we prefer combining the most pertinent open-source solutions into our framework. Figure 2 depicts the overall architecture of proposed framework composed of following modules.

4.1 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 broker *nodes*. 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 routes published messages across the cluster

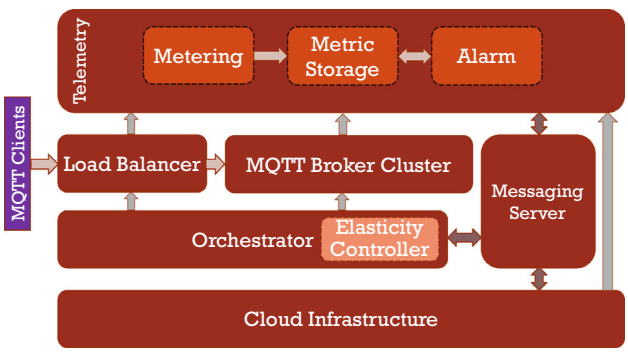


Fig. 2 Architecture of the multi-level elasticity framework for MQTT brokers

nodes 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 [32] need to be supported. The nodes can be installed directly on the VMs or on the containers inside the VMs.

We choose EMQX 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 [33]. 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. The cluster itself can be implemented on Kubernetes to benefit advantages of container virtualization.

4.2 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 [34] can serves as a LB for EMQX cluster and establishes/terminates the TCP connections. Many dynamic scheduling algorithms can be assigned by HAProxy such as round robin, least connection, or randomness.

4.3 Cloud infrastructure

It dynamically manages, provides, and releases 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 [35], 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.

4.4 Orchestrator

This component 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 may include resources such as virtual machines, containers, images, security groups, alarms, scaling policies, etc. To keep the thing simple and user-friendly, grammar of the DSL can be derived from XML, JSON, or YAML. The sub-module *Elasticity Controller* listens elasticity events to trigger corresponding actions. In the framework, the Orchestrator deploys and manages MQTT brokers as well as resources of the Telemetry module such as Metering, Metric Storage, and Alarm.

The main orchestrator supported by OpenStack is Heat service [36]. 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 to completely launch 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.

To do orchestration at container level, we implement Kubernetes as a container orchestrator. Kubernetes, a so-called container orchestration engine, is an open source platform that automates the management, scaling, and deployment of applications as containers. It eliminates lot of manual processes involved in deploying and extending containerized applications. Kubernetes orchestration allows users to build application services that span multiple containers. It schedules those containers on a cluster, expands containers, and manages the condition of the containers over time. With Kubernetes, we pack cluster nodes into Docker containers which are embraced by Kubernetes pods. Pods are the basic Kubernetes scheduling unit, representing a group of one or more application containers and a number of shared resources for those containers. Containers in the same pod share the same IP address and port space. They are always located at the

same location, scheduled together, and run in a shared context on the same node. Each pod is attached to the node where it is scheduled, and stays there until it is terminated.

To perform autoscaling, two techniques can be implemented on Kubernetes: Vertical Pod Autoscaler and Horizontal Pod Autoscaler.

Horizontal Pod Autoscaler (HPA) is a technique to automatically increase or decrease the number of pods by collecting and evaluating CPU usage metrics from the Kubernetes Metrics Server. The number of pods will be in the range min and max which are set when generating HPA. The HPA is implemented as a Kubernetes API resource and as a controller. Every 15 seconds, the controller periodically checks and adjusts the number of pods so that the observed average CPU usage matches the value specified by the user.

HPA calculates the number of pods based on a formula:

$$\#RequiredPods = I(\#CurrentPods * \frac{PresentValue}{ExpectedValue}) \quad (1)$$

where: I() is the rounding function.

For example, the user wants the CPU usage to stay at 60%, but the current demand increases it to 80% and the current number of pods is 2. Then, $\#RequiredPods = I(2 * (80\%)/(60\%)) = I(2.67) = 3$, so the system needs to create 1 more pod for the CPU usage to remain at no more than 60%.

Vertical Pod Autoscaler (VPA) is a technique that automatically increases and decreases resources such as CPU and memory for pods depending on the needs of the pods. Technically, VPA does not dynamically change resources for existing pods; instead, it checks the managed pods to see if the resources are set correctly and if incorrectly, removes them so that the controller can create other pods with updated configurations.

We see that VPA will increase service downtime when doing elasticity actions since this technique removes the pods instead of expanding or shrinking them dynamically. Hence we only use HPA as the main technique for elasticity at container level.

4.5 Telemetry

The telemetry module consists of three sub-modules, as follows: *Metering*: The goal is to collect, normalize, and transform efficiently data produced by orchestrated components. These data are then used to generate different views and help to solve telemetry use-cases, for example specific metrics for elasticity trigger. The subsequent modules (i.e., Alarm and Metric storage) will exploit directly on top of these metrics.

Alarm: It enables the ability of triggering responsive actions based on defined rules against samples or event data collected by Metering module. This module consists of two sub-modules: "Alarm Evaluator" and "Alarm Notifier". The former evaluates measures in Metric Storage module and decides whether they are over or under a given threshold. The latter then triggers a notification and sends a message to the Elasticity Controller of the Orchestrator, who in turn will perform corresponding elastic actions, such as scaling out/in or up/down.

Metric storage: Its goal is to mainly stores aggregated measures of cluster nodes, such as system performance. The metric is a collection of *(timestamp, value)* for a given managed resource, which can be anything from the node's temperature to the CPU usage of a VM. This database also stores events, that happen in Cloud infrastructure, for example an API request has been received, a VM has been started, an image has been uploaded, etc. Stored measures are retrieved for monitoring, billing, or alarming, whereas events are useful for auditing, performance analysis, debugging, etc.

In addition, OpenStack supports some official services for Telemetry module, including Ceilometer [37] for Metering, Aodh [38] for Alarm, and Gnocchi [39] for Metric Storage.

4.6 Messaging server

It is needed for communication between framework's modules based on 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 [40]. 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 may have to apply 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 occurs 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.

5 Multi-level just-enough elasticity

5.1 The method

We propose a method called just-enough elasticity and apply it to the two-level elasticity system using Kubernetes in the OpenStack cloud, specifically as follows.

Inner Level. The main virtualization resource of this level is Kubernetes pod containing containers. Here, the creation or retrieval of pods depends on the needs of the application using HPA technique. To implement HPA engineering, the Kubernetes cluster needs to have Metrics Server installed. Metrics Server is a resource aggregator of Kubernetes cluster. Metrics Server is not intended to be a relay service, it is a source of metrics for system monitoring solutions. The primary function of Metrics Server is to implement HPA and VPA.

Outer Level. The outside of pods are nodes where pods are scheduled to initialize and operate. So, elasticity in the outer level is the process of adding or revoking nodes (essentially the node corresponds to an OpenStack server) using Cluster Controller we have developed in Python language[1].

The just-enough elasticity method is implemented in both outer and inner layers. In essence, the user needs to define a resource area in advance and the increase or decrease will be within the upper and lower limit of that resource area. However, instead of dividing the resource area into servers with identical configuration, the article proposes how to divide that resource area into smaller and more fine-grained pods. Assume that the resource area is defined around 4 CPUs and 8 GB memory. Normally it can be divided into 4 servers, each with 1 CPU and 2 GB memory configuration, but we propose to divide it into 9 pods, each pod uses 0.44 CPU and 910 MB memory. Applying the Kubernetes cluster scaling principle in conjunction with HPA, this method will rely on the number of pods unscheduled due to the fact that the cluster runs out of resources to determine the configuration of the next node to be added to the cluster. Figure 3 depicts the results of the proposed method.

As assumed, a pod requires a resource amount of 0.44 CPU and 910 MB memory. If at the time the system checks, it detects that 4 pods are unscheduled due to a lack of resources, the system will immediately determine that at least 1.76 CPUs and 3640 MB memory are required. From determining the amount of resources needed, the system will suggest a configuration for node which is about to be created, it can be 2 CPUs and 3840 MB memory. CPU value will be rounded up to the nearest integer and Memory will be increased a bit more to make the server "more spacious".

Instead of turning on 2 servers, each with 1 CPU and 2048 MB memory running out a total of 2 CPUs and 4096 MB memory, applying the just-enough method saves 256 MB memory. Along with that, the resource is optimized when only having to run the operating system and other services once, instead of twice on each server. In cases where the amount of missing resources is too small, the system will return a minimal configuration required by the operating system for the new node to work.

5.2 Cluster controller

We have developed a Cluster Controller as a part of Elasticity Controller in the proposed framework to coordinate the scaling of the system at multiple levels. Basically, the Cluster Controller monitors the operation of the Kubernetes cluster and then decides whether to create a new node or remove an empty node from the cluster. To do that, the Cluster Controller will have to work with both OpenStack and Kubernetes through the APIs. We use the OpenStack SDK library to interact primarily with the Open Stack's compute and network APIs. With Kubernetes, we use the Kubernetes Python Client library and work directly with the Kubernetes API through methods GET, POST, PATCH, DELETE, etc. The Cluster Controller will work as shown in the diagram in Fig. 4.

Diagram explanation:

Step 1. The Cluster Controller will periodically check the system every 10 seconds to confirm if the pods are pending or not.

Step 2. Having the pending pods is a condition that determines whether a new node is needed. Since the pod is not only in the pending state when not scheduled, the pod can still be recorded in the pending status when switching from one mode to another, for example from running to terminating, at this step some conditions to select the right pod must be handled. A pending pod due to unscheduled must satisfy the following 3 conditions simultaneously: The pod has a phase of "Pending", has been in pending state for 20 seconds and has not been recorded start-up time.

Step 3. When the cluster has pending pods and all nodes are ready, this means that HPA has created more pods and that the cluster's resources are now insufficient. At this point, the Cluster Controller will accrue the resource

[1] https://github.com/fimocode/cluster_controller.

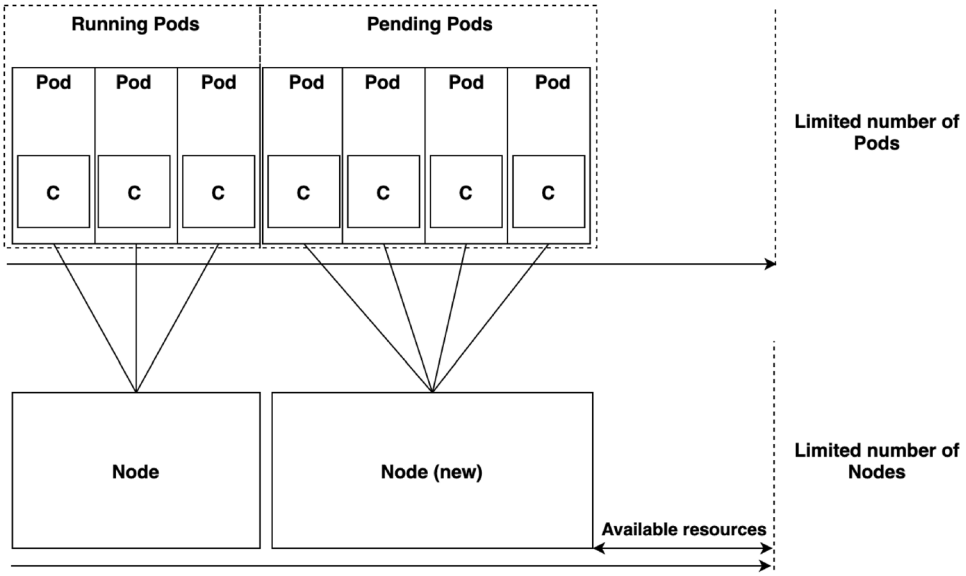


Fig. 3 Newly created nodes have configurations corresponding to demands

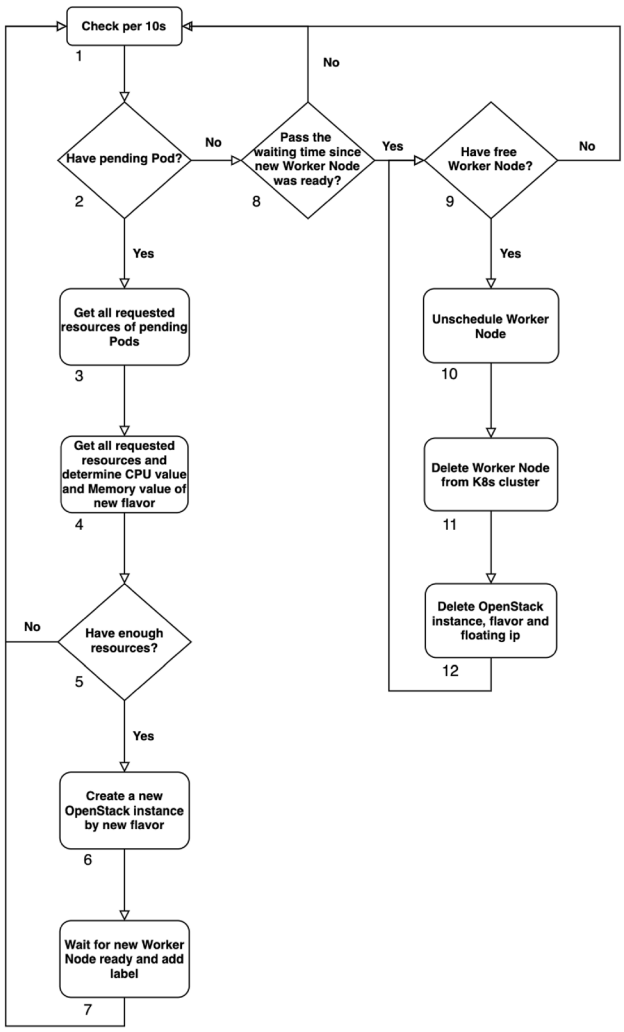


Fig. 4 Workflow of Cluster Controller

requirements of the pending pods with the conditions as mentioned in Step 2 to retrieve the total required resources and specifically the CPU and Memory.

Step 4. In this step, the Cluster Controller will process CPU and Memory values from Step 3. The CPU value will be rounded up (e.g., I(0.3) = 1) because OpenStack only receives the CPU as positive integers. Meanwhile, the required Memory value will be considered along with the operating system that will run on the new Node. Assuming that a Node needs more than 400 MB of memory to run the operating system and essential services when operating in the Kubernetes cluster. So considering the total of memory required and 500 MB, if the total is smaller than the minimum value required by the operating system, the memory value of the new node will be equal to that minimum value. If the total is greater, the Memory value of the new node will be equal to the required Memory value plus 500 MB.

Step 5. At this step, the Cluster Controller will compare the expected configuration of the new node with the rest of the infrastructure resources. If the infrastructure has enough resources, the Cluster Controller moves to Step 6 to create a node. If the infrastructure no longer has enough resources, the Cluster Controller will trigger a warning, refuse to create a new Node, and go back to Step 1.

Step 6. To create a new OpenStack server, the Cluster Controller will do the following:

– The Cluster Controller will generate 1 flavor based on the CPU and Memory values calculated in Step 4 and the values of root disk, ephemeral disk, swap disk, rxtx_factor will be set by default. The values of private network, key pair, image type will also be set with the default value by the Cluster Controller.

- The server after being successfully initialed will run the commands configured as user data, including installing packages and performing some tasks.
- After the system reports that the server has been successfully created, the Cluster Controller automatically creates and attaches a floating IP to that server from the pre-set provider network.

Step 7. When a new node is added to the cluster, it can be in either states: "NotReady" and "Ready". A node is considered "ready" i.e. when recognized by Master Node as "Ready" state. After confirming that the newly added Node is already in a "Ready" state, the Cluster Controller will proceed to label it to clearly separate the functionality of nodes in the Kubernetes cluster, for example "type:run-app". On the application side, when deploying, a "nodeSelector" field is added to the user data so that Kubernetes understands that the pods of this application will only be run on nodes labeled as "type:run-app", for instance.

Step 8. Waiting time is counted since the last node created and recorded as "Ready". If the waiting time has passed, it will go to the next step, if not, go back to Step 1.

Step 9. The operator will have to pre-identify the namespace containing the running applications for the Cluster Controller to check. A Worker Node is considered empty if no pod belongs to one of the namespaces of the application running on them. This process will repeat continuously to give a list of empty Worker nodes.

Step 10. The process of removing a node involves stopping scheduling a new pod on that node and removing the running pods. In the case of an empty node, it is enough to stop the schedule to isolate the node from the cluster.

Step 11. The Cluster Controller will send a request to the Server API to delete a node after it has been quarantined from the cluster.

Step 12. When deleting a server from OpenStack, the Cluster Controller will send a request to the OpenStack Compute API. The deletion process will include both the deletion of the floating IP and flavor of that server.

Steps 10, 11, and 12 repeat until the list of empty Worker nodes is empty. The MQTT cluster diagram implemented with Kubernetes is illustrated in Fig. 5.

6 Validating experiments

In order to validate functionalities of our proposed framework, we conducted the implementation mentioned in Sect. 4 in our homegrown infrastructure at VNU University of Engineering and Technology, Hanoi (VNU-UET). We also make some discussions after the results of the experiments.

6.1 Experiments with VM-level elasticity

6.1.1 Experiment testbed

The testbed is composed of two main parts: one implementation of our elasticity framework, and one load injector to simulate the MQTT clients and their workloads. Test plans of various scenarios are created using built-in functions of the load injector. The simulated publishers generate MQTT messages and send them to the Load Balancer (HAProxy). According to a specific scheduling algorithm, the LB distributes these messages to one EMQX broker of the cluster. The simulated subscribers also connect to the LB which distributes connecting requests to one member of the cluster. The result is MQTT subscriptions to specific topics in the cluster. The routing of messages from their sources to right destinations is conducted internally by the cluster as mentioned in Sect. 4.

Apache JMeter [41] is used as the load injector in our experiments. It is an open-source tool for load test and performance evaluation. Multiple protocols such as HTTP, HTTPS, SOAP, REST, FTP, JMS, etc. are supported by JMeter. Other protocols can be included into JMeter using plugins. To support the experimentation, a MQTT plugin for JMeter implementing some features of MQTT version 5.0 [42] has been developed. To do stress test, distributed testing paradigm with one JMeter master and a couple of slaves is used to ensure that there is no side-effect to the performance of simulated MQTT clients. In this experiment, EMQX brokers and JMeter load injectors are installed on VMs provisioned by the cloud. Each JMeter instance has 8 vCPU and 8 GB memory and each EMQX broker instance has 2 vCPU and 2 GB memory, all with Ubuntu 18.04.

6.1.2 Experiment scenarios

We conducted experiments with multi-publisher and multi-subscriber scenarios usually found in IoT applications using MQTT. The experiments based on these scenarios evaluate the effectiveness of proposed elasticity framework with clustered-broker model.

Multi-publisher scenario: This scenario simulates multiple IoT smart plugs sending data to multiple topics in the brokers. A smart-home center subscribing these topics has responsibility to process and analyze these data. We create a topic scheme with 40 houses which each one includes 90 smart plugs. In turn, each smart plug publishes 10 telemetry parameters. Therefore there are totally 3600 smart plugs playing the publisher roles in the scenario. The testbed for this scenario is illustrated in Fig. 6a.

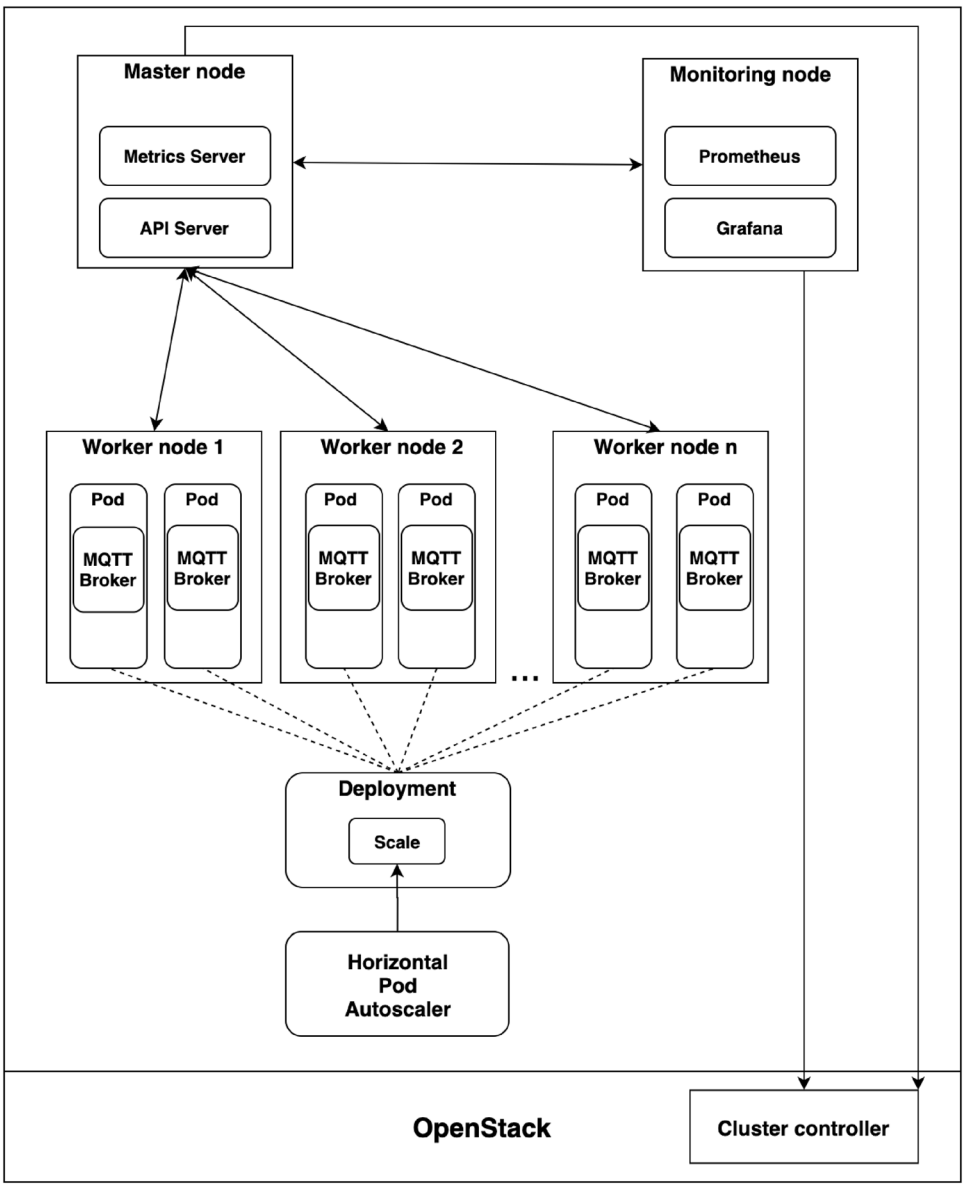


Fig. 5 An implementation of the framework with Kubernetes

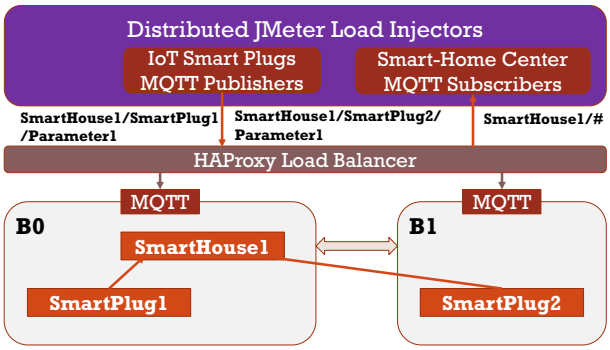
Multi-subscriber scenario: This scenario simulates the smart-home center sending control commands, for instance an indicator of an ON/OFF update, to multiple IoT smart plugs. In this context, the center is the only publisher and the smart plugs are the subscribers. Like the multi-publisher scenario, we also define a topic scheme representing 3600 smart plugs, but they play the subscriber roles in this scenario. The testbed for this scenario is depicted in Fig. 6b.

6.1.3 Results

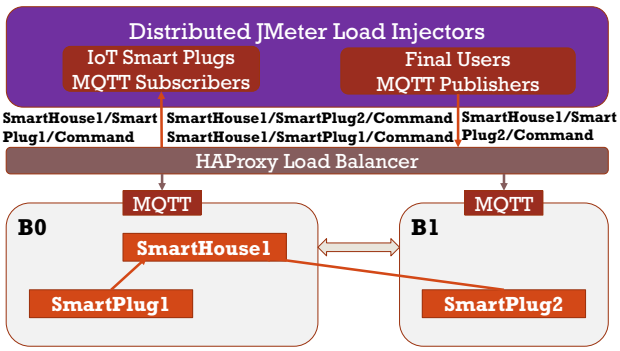
The MQTT workloads are prepared using JMeter test plan. The workload scheme starts with a short warm-up period and then drastically increases when MQTT clients join rapidly to the simulation and start to generate workload. EMQX servers are preconfigured following suggestions from EMQX documentation[2]. We chose IP multicast method for the node-discovery and autocluster mechanisms. The scheduling strategy for HAProxy was set to *balance*.

The Ceilometer, Aodh, and Gnocchi services were configured to measure and store measurements of average %CPU usage and number of virtual CPUs (vCPU) metrics. Upper and lower thresholds for average CPU usage are set to 80% and 25% respectively. It means that if average %CPU usage of clustered brokers breaks these thresholds and the events caught by Ceilometer and Aodh, a notification is sent to Heat service for conducting a

[2] https://www.emqx.io/docs/en/v3.0/tune.html.



(a) Multi-Publisher Scenario



(b) Multi-Subscriber Scenario

Fig. 6 Testbed of VM-level elasticity experiments with clustered brokers

corresponding elasticity action such as scaling in or out. Actually, Heat has to ask other OpenStack services such as Nova, Keystone, and Glance to get the elasticity actions done synchronously. Elasticity plan configured in Heat ensures the number of VMs always in range of 1 to 6.

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

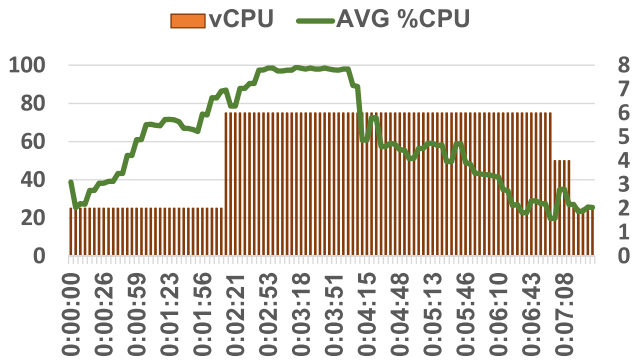
Multi-publisher scenario The multi-publisher scenario is tested with a cluster including one initial broker *B0* and other brokers who will be added dynamically when needed. In Fig. 7a, we see an elasticity effort to mitigate the pressure performed by our system. A group of two MQTT brokers is added to share the workload. These brokers automatically joins the cluster created beforehand by *B0* 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 servers and distributes messages to all load-balancing members. At the end, average %CPU usage of the clustered brokers reduces under the lower threshold in a period of time. Thus, we see another elasticity action (scale in) at this time of the simulation when MQTT clients are terminated or finished. At this point when the average CPU usage goes under 25%, number of broker VMs is decreased gradually to one for minimizing operating cost.

Multi-subscriber scenario The multi-subscriber scenario is tested with a cluster which is similar to the multi-publisher one. We also see the same elastic behaviours shown in Fig. 7b like in the case of multiple publishers. The scaling out actions with two groups of two brokers are triggered at the time later than the multi-publisher scenario. These two group of brokers are added sequentially by Heat. One gap of one minute is set between group 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* keeps retained messages at the brokers longer, thus the more the subscribers are, the busier the brokers are.

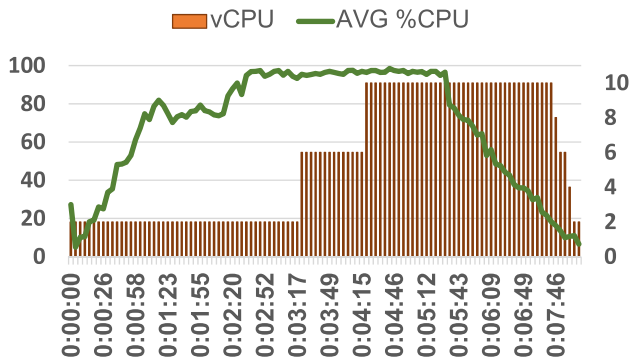
6.2 Experiments with multi-level elasticity

6.2.1 Experiment setup

In this part, we reuse the two scenarios discussed in Sect. 6.1.2: multi-publisher and multi-subscriber. Additionally, we describe another scenario used specifically for the experiment to evaluate multi-level just-enough elasticity method. MQTT workload also is composed and injected by the Apache JMeter plugin mentioned in Sect. 6.1.1. This time, the injector is installed on a OpenStack VM with the following configuration: 4 vCPUs, 4 GB memory, OS Ubuntu 18.04. Kubernetes cluster contains at least one Master node (VM, 2 vCPUs and 3 GB memory) and one Worker node (VM, 1 vCPU and 1 GB memory). Number of Worker nodes can be changed while the system is scaling. Each cluster node can host one or many Kubernetes pods. Each pod requires 0.1 CPU and 120 MB memory and contains one and only one container. This container provides runtime environment for one EMQX broker service and only hosts and exposes this service. MQTT QoS of both publishers and subscribers are set to 1. The lower and upper thresholds of average CPU usage are 10% and 90%, respectively. We use Prometheus [43] and Grafana [44] to monitor the Kubernetes system in real time. More details on the experiment's implementation is depicted in Fig. 5.



(a) Multi-Publisher Scenario



(b) Multi-Subscriber Scenario

Fig. 7 With VM-level elasticity: average %CPU usage of clustered brokers

6.2.2 Multi-publisher scenario

In this scenario, there is only one Worker node, called *wn1* for short, at the very beginning of the experiment. Elasticity activities of the Kubernetes pods and Worker nodes are summarized in Fig. 8. In the figure we see that there is only one pod running on the Worker node *wn1* in the initial phase of the experiment (up to the second minute) and average CPU usage of all brokers is about 13%. From the 3^{rd} to 6^{th} minute, the system triggers multiple scaling-out actions at the container level and the number of pods reaches 5 ones running all together. Some other pods are in the pending state since *wn1* does not have enough CPU resource. It means they must wait until supplemental resources are added. The workload is still high, therefore average CPU usage at this time is about 91%.

Around the 10^{th} minute, another Worker node *wn2* with the same configuration is added and ready. Since then, the pending and new coming pods can be scheduled on the new node. From the 18^{th} to 25^{th} minute, the entire system with 7 pods runs stably which average CPU usage is about 64%. After the 25^{th} minute, the injector stops publishing and subscribing MQTT messages. When average CPU usage is less than 10% around the 30^{th} minute, the scale-in actions are triggered and the idle pods are removed gradually by Kubernetes. Since there is no pod in Worker node *wn2* any more, this node is also released by OpenStack. Around the 35^{th} minute, there is only one pod and one Worker node *wn1* on all over the system.

We see that 7 pods are provisioned and released corresponding to Initial, Scale-out, Balanced, Scale-in phases of the elasticity system, thereby showing that our implementation has been working effectively.

6.2.3 Multi-subscriber scenario

In this scenario, there is also one Worker node *wn1*, at the very beginning of the experiment. Elasticity activities of the Kubernetes pods and Worker nodes are summarized in Fig. 9. In the figure we see that there is only one pod running on the Worker node *wn1* in the initial phase of the experiment (up to the third minute) and average CPU usage of all brokers is about 13%. Other events and activities of this experiment are similar to the multi-publisher one except that the multi-subscriber scenario needs more pods to keep the IoT application working correctly. We also see that 9 pods are provisioned and released corresponding to Initial, Scale-out, Balanced, Scale-in phases of the elasticity system, thereby showing that our implementation has been working effectively.

6.2.4 Multi-level just-enough elasticity

Just-enough elasticity method will work effectively when there are many pending pods at the same moment in the system. To simulate such the scenario with the same experiment setup, we decrease the upper threshold of average CPU usage from 90% to 30%. Each pod requires 0.1 CPU and 280 MB memory. The amount of memory for a new Worker node created by a scaling decision is calculated by applying Formula 1. Other setup is the same with the scenario in Sect. 6.2.2.

Similar to other scenarios, there is also one Worker node *wn1*, at the very beginning of the experiment. Elasticity activities of the Kubernetes pods and Worker nodes (according to amount of memory provisioned) are summarized in Fig. 10. In the figure we see that there is only one pod running on the worker node *wn1* in the initial phase of the experiment (up to the second minute) and average CPU usage of all brokers is about 12%. Soon after that, the system triggers one scaling-out action at the container level and the number of pods reaches 2 ones running all together. Besides, two pods are in the pending state since *wn1* does not have enough CPU resource. It means they must wait until supplemental resources are added. The workload is

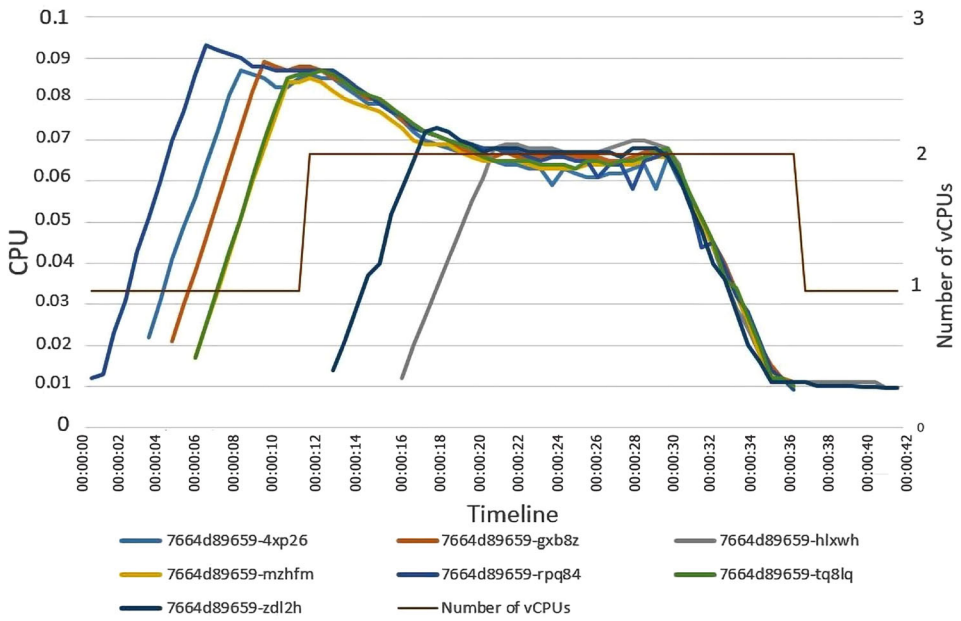


Fig. 8 Elasticity activities on the Pods and worker nodes - Multi-Publisher scenario

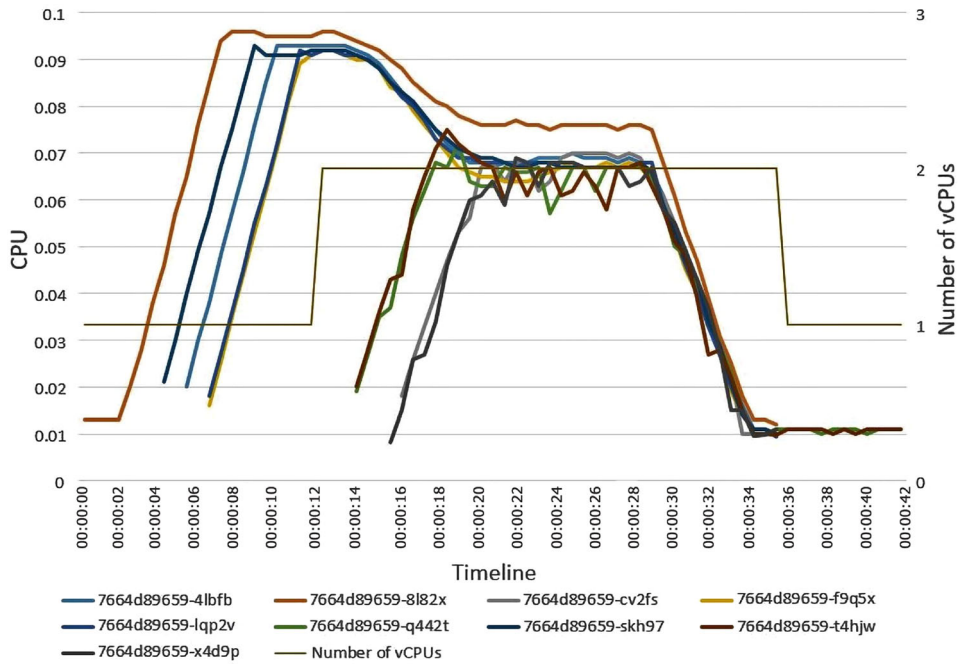


Fig. 9 Elasticity activities on the Pods and worker nodes - Multi-Subscriber scenario

still high, therefore average CPU usage at this time is about 84%. Since there are two pending pods, one more Worker node with minimal configuration (*wn2*, 1 vCPU and 1024 MB memory) is added to the system around the 7^{th} minute. The new Worker one is ready at the 8^{th} minute and the two pending pods are scheduled to run on the *wn2*.

At the next check, 4 pods are in the pending state, thus one more Worker node (*wn3*, 1 vCPU and 1340 MB memory) is created and ready around the 12^{th} minute. Since then the pending and new coming pods can be scheduled and run on the *wn3*. Again at the next check, the system recognizes that there are 6 pending pods, therefore one more Worker node (*wn4*, 1 vCPU and 1900 MB memory) is created and ready around the 18^{th} minute. Hence the pending and new coming pods can be scheduled and run on the *wn4*. Around the 19^{th} minute, the injector stops publishing and subscribing MQTT messages. Thereafter the system starts to trigger some scale-in actions and we see that there are 5 pods running with 10% CPU usage in average around the 26^{th} minute. Eleven minutes later, there is only one pod and one Worker node on all over the system.

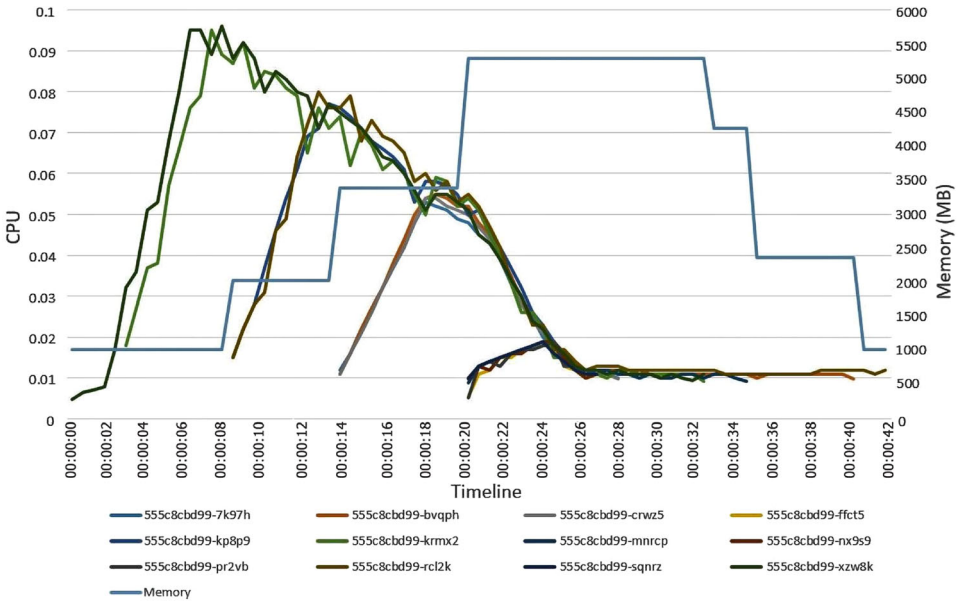


Fig. 10 Elasticity activities of the Pods and worker nodes in Just-enough scenario

Briefly, when number of pods increases very fast like in this scenario, multi-level just-enough elasticity method gives advantages in saving memory over creating individual Worker node with minimal configuration. This strategy is effective when the deployed application consumes a lot of memory and the elastic range is far from the limit of the originally established resource area.

7 Conclusion

This article presents a flexible framework enabling multi-level just-enough elasticity for MQTT brokers in IoT applications. Our framework delivers elasticity through the use of existing off-the-shelf components already in cloud ecosystems. At VM level, our elastic MQTT broker service has been successfully deployed using EMQX as MQTT broker solution and OpenStack as cloud environment. At container level, just-enough elasticity has been earned thanks to the implementation of Kubernetes cluster and EMQX brokers. Multiple comprehensive experiments are conducted by generating traffics to the service at various load levels to observe changes in number of broker instances at multiple levels. Our experiment results show that our elasticity MQTT broker service adapts well to user load changes, making the service fully accommodate incoming traffics as well as keep operating cost low. In the future, we could investigate our approach from an energy efficiency perspective and extend the experiments with large data sets and more scenarios to enhance the results used for evaluation. Furthermore, we also plan to develop a three-level scaling algorithm with software scaling inside the container to further increase resource efficiency.

Author contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by LMP, N-T-TL and X-TN. The first draft of the manuscript was written by LMP and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Data availibility The data used to support the findings of this study are available from the corresponding author upon request.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Sharma, N., Panwar, D.: 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, pp. 1127–1132 (2020)
2. Turner, V., Reinsel, D., Gantz, J.F., Minton, S.: The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. IDC Report (2014)
3. Message Queuing Telemetry Transport. http://mqtt.org. Accessed 30 April 2021
4. Mell, P., Grance, T.: The NIST definition of cloud computing. In: NIST special publication, vol. 800, pp. 145 (2011)
5. Eugster, P.Th., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. In: ACM Computing. Survey. 35, 2, 114-131 (2003)

6. EMQX Broker. https://docs.emqx.io/broker/latest/en/. Accessed 30 April 2021
7. Kawaguchi, R., Bandai, M.: Edge Based MQTT Broker Architecture for Geographical IoT Applications. In: International Conference on Information Networking (ICOIN), Barcelona, Spain, pp. 232-235 (2020)
8. Gupta, V., Khera, S., Turk, N.: MQTT protocol employing IOT based home safety system with ABE encryption. In: Multimedia Tools and Applications (2020)
9. Mukambikeshwari, A. Poojary: Smart Watering System Using MQTT Protocol in IoT. In: Advances in Artificial Intelligence and Data Engineering. Advances in Intelligent Systems and Computing, Vol. 1133. Springer, Singapore (2020)
10. See, Y.C., Ho, E.X.: IoT-Based Fire Safety System Using MQTT Communication Protocol. In: IJIE, Vol. 12(6), pp. 207–215 (2020)
11. Nazir, S., Kaleem, M.: Reliable Image Notifications for Smart Home Security with MQTT. In: International Conference on Information Science and Communication Technology (ICISCT), Karachi, Pakistan, pp. 1–5 (2019)
12. Alqinsi, P., Edward, I.J.M., Ismail, N., Darmalaksana, W.: IoT-Based UPS Monitoring System Using MQTT Protocols. In: 4th International Conference on Wireless and Telematics (ICWT), Nusa Dua, pp. 1–5 (2018)
13. Comparison of MQTT Brokers. https://tewarid.github.io/2019/03/21/comparison-of-mqtt-brokers.html. Accessed 30 April 2021
14. Schmitt, A., Carlier, F., Renault, V.: Data Exchange with the MQTT Protocol: Dynamic Bridge Approach. In: IEEE 89th Vehicular Technology Conference (VTC2019-Spring), Kuala Lumpur, Malaysia, pp. 1–5 (2019)
15. Zambrano A.M.V., Zambrano M.V., Mejía, E.L.O., Calderón X.H.: SIGPRO: A Real-Time Progressive Notification System Using MQTT Bridges and Topic Hierarchy for Rapid Location of Missing Persons. In: 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. Accessed 30 April 2021
17. Jutadhamakorn, P., Pillavas, T., Visoottiviseth, V., Takano, R., Haga, J., Kobayashi, D.: A scalable and low-cost MQTT broker clustering system. In: 2nd International Conference on Information Technology (INCIT), Nakhonpathom, pp. 1-5 (2017)
18. Thean, Z. Y., Voon Yap, V., Teh, P. C.: Container-based MQTT Broker Cluster for Edge Computing. In: 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE), Kedah, Malaysia, pp. 1–6 (2019)
19. Detti, A., Funari, L., Blefari-Melazzi, N.: Sub-linear scalability of MQTT clusters in topic-based publish-subscribe applications. IEEE Trans. Network Serv. Manag. **17**(3), 1954–1968 (2020)
20. Ullah, A., Li, J., Hussain, A.: Design and evaluation of a biologically-inspired cloud elasticity framework. Clust. Comput. **23**(4), 3095–3117 (2020)
21. Righi, R.R., Correa, E., Gomes, M.M., Costa, C.A.: Enhancing performance of IoT applications with load prediction and cloud elasticity. Future Gener. Comput. Syst. **109**, 689–701 (2020)
22. Fourati, M. H., Marzouk, S., Drira, K., Jmaiel, M.: DOCKERANALYZER : Towards Fine Grained Resource Elasticity for Microservices-Based Applications Deployed with Docker. In: 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Gold Coast, Australia, pp. 220–225 (2019)
23. Nardelli, M., Cardellini, V., Casalicchio, E.: Multi-Level Elastic Deployment of Containerized Applications in Geo-Distributed Environments. In: IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), Barcelona, pp. 1–8 (2018)
24. Rodrigues, V.F., Wendt, I.G., Righi, R.R., Costa, C.A., Barbosa, J.L.V., Alberti, A.M.: Brokel: Towards enabling multi-level cloud elasticity on publish/subscribe brokers. Int. J. Distrib. Sens. Networks (2017). https://doi.org/10.1177/1550147717728863
25. Vavassori, S., Soriano, J., Fernández, R.: Enabling large-scale IoT-based services through elastic publish/subscribe. Sensors **17**, 2148 (2017)
26. Yadav, M.P., Yadav, D.K.: Maintaining container sustainability through machine learning. Clust. Comput. **24**(4), 3725–3750 (2021)
27. Dawoud W., Takouna I., Meinel C.: Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning. In: Krishna P.V., Babu M.R., Ariwa E. (eds) Global Trends in Computing and Communication Systems. ObCom 2011. Communications in Computer and Information Science, vol 269. Springer, Berlin (2012)
28. Nardelli M., Russo Russo G., Cardellini V., Lo Presti F.: A multi-level elasticity framework for distributed data stream processing. In: Mencagli, G., et al. (Eds.), Euro-Par 2018: Parallel Processing Workshops. Euro-Par 2018. Lecture Notes in Computer Science, Vol. 11339. Springer, Cham (2019)
29. Russo Russo, G., Nardelli, M., Cardellini, V., Lo Presti, F.: Multi-level elasticity for wide-area data streaming systems: a reinforcement learning approach. Algorithms **11**(9), 134 (2018)
30. Linh Manh Pham, Truong-Thang Nguyen, Tien-Quang Hoang, "Towards an Elastic Fog-Computing Framework for IoT Big Data Analytics Applications", Wireless Communications and Mobile Computing, vol. 2021, Article ID 3833644, 16 pages, 2021
31. Roure, D., Goble, C.: Software design for empowering scientists. IEEE Softw. **26**(01), 88–95 (2009)
32. A distributed, reliable key-value store. https://etcd.io/docs/v3.4.0/. Accessed 30 April 2021
33. Kubernetes. https://kubernetes.io/. Accessed 30 April 2021
34. HAProxy. https://www.haproxy.com/solutions/load-balancing/. Accessed 30 April 2021
35. OpenStack: Open Source Cloud Computing Infrastructure. https://www.openstack.org/. Accessed 30 April 2021
36. OpenStack Heat. https://docs.openstack.org/heat/latest/. Accessed 30 April 2021
37. OpenStack Ceilometer. https://docs.openstack.org/ceilo-meter/latest/. Accessed 30 April 2021
38. OpenStack Aodh. https://docs.openstack.org/aodh/latest/. Accessed 30 April 2021
39. Gnocchi - Metric as a Service. https://gnocchi.xyz/. Accessed 30 April 2021
40. RabbitMQ. https://www.rabbitmq.com/. Accessed 30 April 2021
41. Apache Jmeter. https://jmeter.apache.org/. Accessed 30 April 2021
42. Pham, L.M., Nguyen, T.T., Tran, M.D.: A benchmarking tool for elastic MQTT brokers in IoT applications. Int. J. Inf. Commun. Sci. **4**(4), 59–67 (2019)
43. Prometheus. https://prometheus.io/. Accessed 30 April 2021
44. Grafana. https://grafana.com/. Accessed 30 April 2021

Linh Manh Pham is a lecturer at University of Engineering and Technology,Vietnam National University, Hanoi (VNU-UET). He was a postdoctoralresearcher at Inria, France. He earned an MSc. in Computer Science in theUSA and a Ph.D. in Cloud Computing at Grenoble Alpes University, France. Hisarea of research is Cloud/Fog Computing, and he intends to highlight thebenefits of this relatively novel field of research.

Nguyen-Tuan-Thanh Le is with Thuyloi University, Hanoi as a lecturer. He hasa Master degree in Software Engineering at University of Engineering andTechnology, Vietnam National University, Hanoi (VNU-UET). He earned a Ph.D.in Computer Science at Paul Sabatier University, Toulouse, France.

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 and currently a Ph.D. candidate at VNU-UET.