Client-Transparent and Self-Managed MQTT Broker Federation at the Application Layer

José Fernando de Lacerda Machado Jr.*, Marco Aurélio Spohn†, Lisandro Zambenedetti Granville*
*Institute of Informatics - Federal University of Rio Grande do Sul - Porto Alegre, Brazil
†Federal University of Fronteira Sul - Chapecó, Brazil
lacerda.machado@ufrgs.br, marco.spohn@uffs.edu.br, granville@inf.ufrgs.br

***Abstract*—The use of IoT devices for monitoring and automation became very disseminated. Also, and as a consequence, IoT scalability issues evolved into one of the main challenges on large deployments. One of the most adopted architectures for the communication between IoT devices and other information systems is based on the Publish/Subscribe paradigm, mainly embraced by MQTT-capable devices. Some implementations aimed to solve scalability challenges on those environments, mainly using clustering solutions, while a few considered federation approaches. Existing solutions are predominantly proprietary, lacking public documentation, and may be considered incipient. In the present work, we propose a client-transparent and self-managed solution for scaling MQTT brokers using federation approach through a python-written wrapper, providing federation functionalities and message routing without customization of regular brokers. While clustering solutions usually target throughput improvement, the federation approach explores higher availability through distributed architecture. We present a validation to expose our solution's flexible availability and its capability to deal with topology change issues.**

I. Introduction

The MQ Telemetry Transport (MQTT) protocol has been widely adopted on Internet of Things (IoT) devices communication [1] [2], not only because of its low data overhead but also because of its reliability and strong standardization. That is due not only to MQTT's maturity but also to its ease of implementation. As a result, the adoption and deployment of MQTT-enabled devices and the solutions based on the platform became popular [3].

MQTT employs the Publish/Subscribe (P/S) paradigm where a broker intermediates the communication between publishers (*e.g.*, IoT devices) and subscribers (*e.g.*, other devices or applications that consume the publishers' offered information). Because IoT devices (playing the role of either publishers or subscribers) usually suffer from limited resources (battery capacity, processing power, and communication support), MQTT-based solutions aim at dealing with the limitations of IoT deployed setups [4]. Scaling up such systems by supporting an increasing number of publishers and subscribers also has to consider scaling up MQTT brokers to avoid them becoming bottlenecks in environments with varying amounts of flowing information. There exist several solutions for MQTT-based setups that employ clustering [5] [6] [7] as a technique to deal with the brokers' scalability issue. Unfortunately, most of those solutions are only available in commercial products. As an alternative to the clustering techniques, the use of federation techniques [8] are being considered, but still less mature than clustering.

Although attempts to solve the scalability problem in MQTT are in place, as mentioned above, the central fact is that MQTT implementations with improved scalability are scarce and often limited to proprietary products. As such, scaling up MQTT environments by employing a public, freely available solution is still a need. Implementing such functionality is thus an opportunity and potential game-changer, providing elastic capabilities for a broad set of application scenarios.

In a previous work [9], we proposed a self-managed MQTT federation that offers the first movement toward building and maintaining an overlay mesh network of autonomous brokers. Clients (*i.e.*, publishers, and subscribers) communicate over this self-organizing mesh network with low control overhead. The materialization of the above solution can range from a more intrusive one (*i.e.*, requiring changes to the broker) to the one in which the federation mechanism, which internally also relies on the P/S paradigm, stays exclusively at the application layer (*i.e.*, brokers remain unchanged) [10].

In this paper, we advance our previous research by presenting the design, implementation, and case study of a federation module for MQTT as a wrapper. Our solution, written in Python 3.8 and attached to a Mosquitto 3.1.1 MQTT broker, is flexible, easy-to-implement, and better scales up because of our self-managed federation approach. Our implementation seeks to be the least intrusive possible to the ordinary MQTT environment.

The remainder of this paper has the following organization. In Section II, we review related work. In Section III, we introduce our MQTT wrapper and detail how a set of wrappers in the mesh network federate. We show our implementation and case study in Section IV. In Section V, we conclude this paper with final remarks and future work.

II. Related Work

Several research efforts have addressed scaling up P/S systems, some based on clustering and others based on federation strategies. Bakker and Pattenier [11] present federation strategies on networked systems. They focus on two leading solutions: federation for connection-oriented networks and federation for connection-less networks, as those based on TINA-C NRA (Telecommunications Information Networking

Architecture Consortium - Network Resource Architecture), mainly employed in telecommunications environments.

Uramoto and Maruyama [12] present the InfoBus Repeater application, conveying a unique approach for scaling up MQTT environments throughout a bus. The application, written in Java, acts as a middleware that allows intercommunication between group members. When a member joins the bus, it informs its status as a publisher, subscriber, or both. The bus is a single point of failure and a bottleneck, providing limited scalability.

Bass [13] carried out a structured analysis of a P/S federated network approach for critical infrastructure environments. The work evaluates assembling models, the compliance of existing solutions, and the security aspects and characteristics of possible topologies. Although it is primarily theoretical, the work delivers a broad view of possible scenarios and solutions for assembling federated networks of sensors.

Thean *et al.* [14] presents a work based on clustering MQTT brokers to deal with edge computing demands. They propose an architecture for scaling a cluster of container-based MQTT brokers on the edge of the environment, each acting as a bridge to the brokers placed on a cloud infrastructure. Even though their solution provides enhanced scalability results, the container orchestrator remains a single point of failure.

In previous work, we [8] presented a generic approach for federating MQTT brokers following a mesh-assembling mechanism over an overlay network. The solution includes an overlay mesh network that provides the primary communication system, allowing the arrangement of topic meshes, reaching all clients for any federated topic, regardless of where the client connects. A new mesh forms whenever the first subscriber connects to any broker. All brokers with subscribers for the same topic, and any broker that interconnects them, integrate the topic mesh. A mesh has a core broker that coordinates the mesh construction and maintenance. The routing of topic messages begins by forwarding them toward the corresponding topic core, but as soon as the message reaches a mesh member, the message spreads throughout the mesh.

Afterward, we [10] proposed an implementation based on an endogenous approach [9]. The solution employs the native P/S mechanism for managing the meshes and routing of messages. Subscribers must send a beacon message to advertise themselves, but the federation of brokers is primarily transparent to the clients.

III. Federation proposal

We present a self-managed wrapper-based federation solution coded in Python and integrated into the Eclipse Mosquitto 3.1.1 MQTT broker. A wrapper is software capable of interacting with the MQTT broker transparently. For that, the wrapper monitors the broker's log, which has its data redirected to a topic called `$SYS`. The wrapper gathers data from all topics, being possible to monitor specific topics and subtopics for later forwarding its data to all federated members. This strategy makes it possible to adapt this solution to other known MQTT broker implementations capable of redirecting log data to a specific topic. Each broker to be federated shall have a wrapper attached and be responsible for communicating with the neighboring set of brokers and wrappers.

The wrapper follows the principles of the federation mechanism proposed by [8], on which the brokers federate through their neighbors, building a mesh. Each federated broker forwards messages through the mesh, allowing communication between brokers that are not neighbors. Information needed to manage the federation includes a broker identification number (*BrokerID*), the distance in hops to the *core broker* (*i.e.*, the broker that coordinates the mesh), the *mesh membership flag* (signaling whether the broker is in a given mesh), a list of neighboring brokers, and the desired mesh redundancy (achievable when the overlay topology allows).

A. Architecture

The main element of our architecture is the wrapper that interacts with the broker and allows communication with neighboring wrappers and their related brokers. The wrapper interacts with the broker monitoring its logs and all the topics and subtopics, so we assume that the wrapper can access all data generated on the broker.

The wrappers interact with other wrappers through network sockets, allowing data exchange. We chose UDP for transporting data between brokers, as a missing packet may arrive on a wrapper through different paths, so delivery confirmation is not critical. The main goal is to deliver data published on a given topic on a broker with a federated wrapper by forwarding it to another broker through its wrapper where there are connected subscribers to that given topic. Figure 1 presents the schematic architecture.

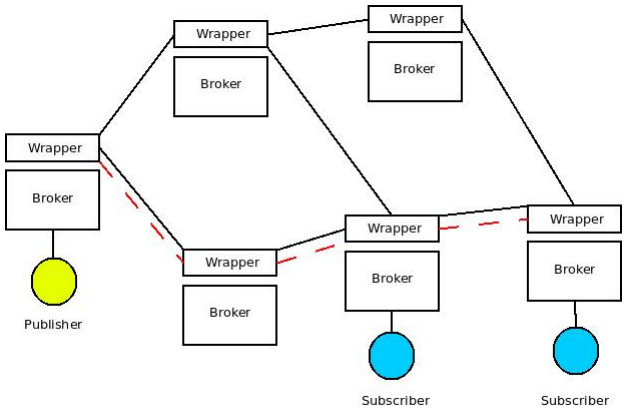

Fig. 1. System's architecture overview

B. Behavior

The wrappers' federation process encompasses two main phases. First, the federation assemblage, regarding the overlay infrastructure that provides communication among federated wrappers. Second, the mesh building and maintenance provide the means for transporting data between wrappers connected to brokers with publishers and subscribers. Regular topics with subscribers will be called federated topics.

1) Federation Assemblage: Federation assemblage assumes that every wrapper knows its neighbors' IP addresses. Periodical *hello* messages are exchanged between neighbor wrappers to keep track of their online status. Announcement messages with an identification number (BrokerID) for the wrapper are also sent periodically, which are forwarded through their neighbors to the other interconnected wrappers, allowing the assemblage of the overlay network (i.e., nodes learn about all federated brokers and their distances). As the announcements propagate, the wrapper nodes settle collisions by randomly selecting a new ID.

As the federation starts, there is a need to elect a core wrapper to coordinate the updates on the federation topology and other management matters. The election is based on the BrokerID number, winning the node with the lowest ID. This core wrapper will be called the management core.

2) Topic meshes: Topic meshes are the meshes of wrappers where there are subscribers to a given topic. They start along the federated network. When a wrapper detects a first subscriber for any topic for which there is no mesh, the wrapper advertises itself as the core for the new mesh. Through a *core announcement* message, wrappers learn how to reach any previously established core. Management and topic cores are similar, with the latter being the reference point for starting topic meshes, while the former orchestrates the overlay network. The node with the lowest ID wins the dispute in a core announcement contention.

If there is a topic core on a topic with a new subscriber, the wrapper sends a mesh membership announcement toward the topic core. If there are two or more paths towards the core with the same distance, it is possible to handle redundancy. The new membership may trigger intermediate wrappers to join the mesh to keep it connected. As for publications, the topic core is the reference target: the wrapper sends the message toward the core, and once reaching it or a mesh member first, the message spreads throughout the mesh. Wrappers keep a local cache to avoid sending the same message indefinitely, considering that the same publication may arrive through different paths.

IV. Implementation and Validation Tests

In our implementation, we have used Python version 3.8, supported by libraries providing MQTT functions such as threading, serialization, and socket connections. The Paho-MQTT library is also used and plays the most crucial role in the solution, allowing the interaction between the wrapper and the broker and entitling monitoring of subscriptions and messages on topics.

The wrapper has five main functional groups and the initial data setup, with this latter including information regarding IP addresses and ports, constants, initial broker identification, and the list of neighbors. By default, wrapper instances communicate through UDP port 10500.

The first functional group handles low-level network-related activities, such as packet sending and receiving. For performance purposes, we used UDP on the transport layer. There are two main functions - packet sending and packet receiving. Packets that need to be redirected are also handled over these functions.

The second functional group handles the operational functions. Packets have a function identifier and are handled according to their *message types*. A packet's structure comprises a *message type* field, followed by the data regarding that particular message.

```
('fd', (14, 1, ('10.81.180.217', 7535)))
```

There are ten different types of messages: hello (hl), hello back (hb), federate (fd), reconsider (rc), core announcement (ca), topic core (tc), reconsider topic core (rt), topic subscribe (ts) topic message (tm) and topology update (tu). Federate, core announcement, and topic core announcement have detailed information on message sequence numbering for control purposes. For instance, the structure of a federation message is as follows:

```
('fd', (14, 1, ('10.81.180.217', 7535)))
          (seq, dist, (ip, id)
```

The *fd* field indicates the message type, followed by a tuple that carries the announcement sequence number, the hop distance from the announcer, and its BrokerID - which also consists of a tuple holding the IP address and the node's virtual ID. The source broker defines the sequence number, while the distance field changes as the message moves farther from the source.

A group of processes runs as *daemons*. These processes are responsible for the hello, federation, core announcements, MQTT broker monitoring, and maintenance. By default, nodes transmit federation and core announcements every 20 s and hello messages every 5 s. The latter group comprises additional functions, such as log generating, cache flushing, and debugging.

The environment for validating our solution consisted of a regular desktop computer (4th generation quad-core i7 with 8GB of RAM) running Linux Mint 20.3 and Oracle VirtualBox 6.1.38 hypervisor with a Ubuntu 22.04 virtual machine guest, using 4GB of RAM and 35GB of disk space. LXD 5.0 was used to host LXC containers inside the virtual machine, using the mainstream Ubuntu 20.04 LTS image. Each container had Mosquitto 3.1.1 installed alongside Python 3.8.10, with Paho-MQTT 1.6.1 library installed through Pip.

For redirecting log messages generated by Mosquitto to the `$SYS` topic, `/etc/mosquitto/mosquitto.conf` needed to be configured with the lines below:

```
log_dest topic
log_type all
```

Each container had only one network interface installed, named 'eth0', connecting to a virtual bridge 'lxdbr0' on the Ubuntu 22.04 host (this is necessary because the solution uses the network interface name to gather its IP address). Each broker/wrapper set on the network corresponds to one

container instance. The configuration regarding the neighbors is in a list in the wrapper, as mentioned before. Example:

```
neigh = [('10.81.180.180'),
         ('10.81.180.217')]
```

A. Environment remarks and considerations

LXC delivers a handful of scenarios better than a Docker-based environment because it provides a complete operating system experience with minimal memory and processing footprint. Also, Docker needs a new image of the application to be generated each time the source code is changed. With LXC, creating new instances and cloning running instances is straightforward, making scaling the system a trivial task.

B. Validation and testing

For validation purposes, we used the scenario proposed in [8] (Figure 4). The validation aims to check the assemblage of the overlay network and identify whether the wrapper behaves as desired, handling federated topics and message routing.

After configuring and starting all the wrappers on network nodes, the nodes populate their federation list. Figure 2 shows the node's three federation lists (`n =` is the list of neighbors, `f =` is the list of federated nodes, followed by the broker IP and node's ID, then the management core).

```
core = (('10.81.180.150', 827), 38)
n = ['10.81.180.22', '10.81.180.230', '10.81.180.180', '10.81.180.217']
f =
('10.81.180.22', 6233, 0, [('10.81.180.22', 9)])
('10.81.180.230', 1276, 0, [('10.81.180.230', 3)])
('10.81.180.150', 827, 0, [('127.0.0.1', 0)])
('10.81.180.180', 1111, 0, [('10.81.180.180', 7)])
('10.81.180.217', 8611, 0, [('10.81.180.217', 7)])
('10.81.180.243', 4397, 1, [('10.81.180.217', 6)])
t =
broker = ('10.81.180.150', 827)
core = (('10.81.180.150', 827), 38)
```

Fig. 2. Node's 3 federation list

The next step consists of simulating a topic subscription and observing the federated topic evolution. We start with a subscriber at node zero. Figure 3 depicts the node's five debugging outputs, showing the federated topic `mytopic/example_subtopic`, and the core for that given topic is node zero - the node that we have the subscriber attached to. Publications for the related topic are forwarded toward node zero and flood the mesh once reaching it.

To test publication routing, we did a publication on node four, monitoring the wrapper, gathering the publication, and sending it toward the core through node one. Next, a publication starts on node five, which has route redundancy towards the core. In this case, we randomly choose one of the neighbors in the message-forwarding process. In all situations under consideration, the publications successfully reach the subscriber.

C. Performance analysis

When analyzing our solution's performance, it is imperative to differentiate the key performance indicators between

```
n = ['10.81.180.180', '10.81.180.22', '10.81.180.150']
f =
('10.81.180.243', 228, 1, [('10.81.180.180', 6)])
('10.81.180.217', 6785, 2, [('10.81.180.180', 6)])
('10.81.180.230', 4314, 0, [('127.0.0.1', 0)])
('10.81.180.22', 3480, 0, [('10.81.180.22', 5)])
('10.81.180.180', 7712, 0, [('10.81.180.180', 7)])
('10.81.180.150', 4154, 0, [('10.81.180.150', 4)])
t =
('mytopic/example_subtopic', [('10.81.180.243', 228)])
broker = ('10.81.180.230', 4314)
core = (('10.81.180.243', 228), 33)
```

Fig. 3. Example of a federated topic with a subscriber attached to node 0

clustering and federation. Clustering relies on aggregating computational resources as a single block, having the load balancer as the main bottleneck and single point of failure. On the other hand, the federation relies on orchestrating distributed resources with multiple access points. Therefore, one could argue that the federation first targets the service's high availability, while clustering aims at high throughput.

The performance analysis evaluates the message delivery reliability while dealing with changes to the overlay topology. For the case studies, we consider the analysis starting after an initial configuration for the federation of brokers is running.

The first test consists of running a subscriber for a non-existent topic. The corresponding node advertises itself as the core for the new topic. The topic mesh construction begins as new subscribers for the same topic connect to different nodes. To test the message routing, a client starts publishing from a node outside the mesh. Initially, messages are directed toward the core, spreading throughout the mesh once reaching the core or a mesh member.

We evaluate the core election process by simultaneously instantiating two subscribers to the same topic in separate nodes. For a while, two core announcements wander around the federation, but eventually, the core with the greater ID gives way to the one with the minor ID. Once nodes learn about the remaining core, the mesh construction process converges. In the case of network partitioning, there will be two different scenarios. In the first, the slice that kept the existing core will rely on it to identify disconnected nodes, and the remaining nodes will be informed to update their databases. In the second, the remaining brokers on the coreless slice will identify that they are not receiving core information updates and will orchestrate a new core election.

V. Conclusions

This work presents a new variant for the federation approach introduced in previous works [8]–[10]. Our solution differs by adopting a wrapper cooperating with the broker while communicating to other wrappers running on neighboring brokers. This proposal is ongoing work and lacks functionalities available on a regular Mosquitto MQTT broker, such as QoS controls and authentication, which are part of future work. However, we achieve our main objective: to provide and evaluate a self-managed federation of MQTT brokers.

We noticed that the initial federation process (*i.e.*, related to the overlay network) requires more time than anticipated. It

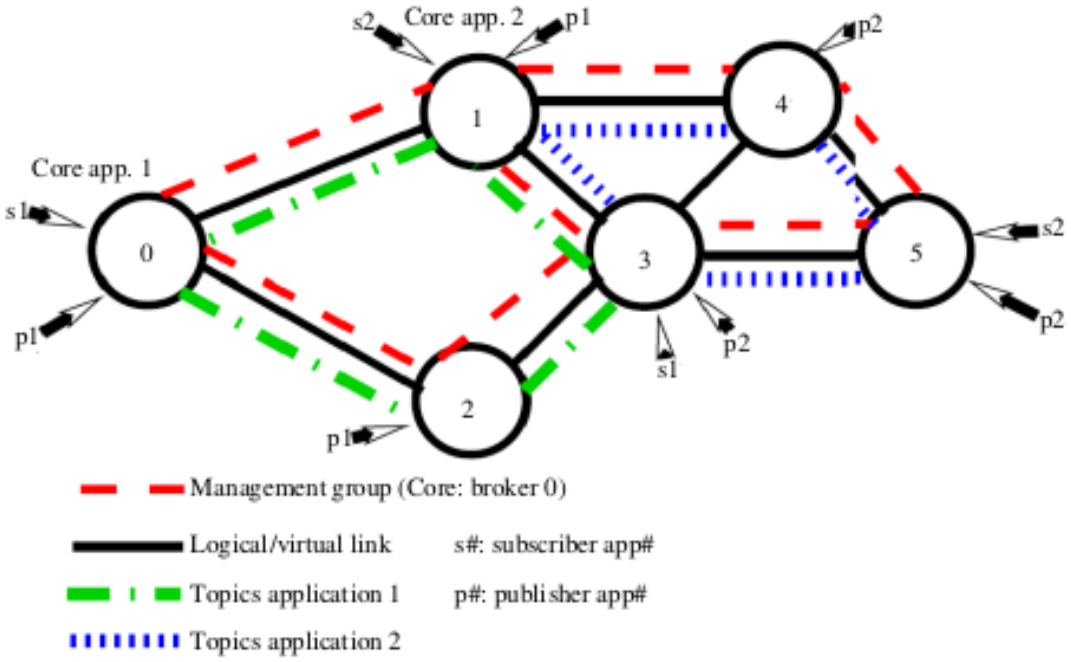

Fig. 4. Validation topology

shows it is worth improving all the topology-related services, from the initial overlay establishment to all the resulting maintenance. We are working on new mechanisms for better handling joining and leaving the federation network.

Monitoring active subscribers is still an open issue. For now, we assume the connection between a subscriber and the broker is stable and remains connected indefinitely. On the other hand, intermittent connecting clients might get new IDs when reconnecting, which becomes a broader monitoring burden.

The present work opens an extensive list of possibilities for improving the federation of MQTT brokers. Compared to other solutions, mainly proprietary market-driven solutions, our solution covers a particular domain, and the initial prototype shows our proposal's potential. The increased availability comes from the self-managing mesh approach, which is central to our work. In case of network partitioning or general connectivity problems, the service is still available for the clients in the same partition. As partitions merge again, the system converges quickly with reduced control overhead.

REFERENCES

[1] M. Houimli, L. Kahloul, and S. Benaoun, "Formal Specification, Verification and Evaluation of the MQTT Protocol in the Internet of Things," in *International Conference on Mathematics and Information Technology (ICMIT)*, Adrar, Algeria, Dec. 2017, pp. 214–221.

[2] N. Naik, "Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP," in *IEEE International Systems Engineering Symposium (ISSE)*, Vienna, Austria, Oct. 2017, pp. 1–7.

[3] M. Kashyap, V. Sharma, and N. Gupta, "Taking MQTT and NodeMcu to IOT: Communication in internet of things," *Procedia Computer Science*, vol. 132, pp. 1611 – 1618, 2018, international Conference on Computational Intelligence and Data Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050918308585

[4] I. Made Wirawan, I. Dwi Wahyono, G. Idfi, and G. Radityo Kusumo, "Iot communication system using publish-subscribe," in *2018 International Seminar on Application for Technology of Information and Communication*, 2018, pp. 61–65.

[5] HiveMQ, "HiveMQ website," https://hivemq.com, 2022, accessed: 2022-04-19.

[6] EMQX, "EMQX website," https://emqx.com, 2022, accessed: 2022-08-19.

[7] VerneMQ, "VerneMQ website," https://vernemq.com, 2022, accessed: 2022-08-19.

[8] M. A. Spohn, "Publish, subscribe and federate!" *Journal of Computer Science*, vol. 16, no. 7, pp. 863–870, Jul 2020. [Online]. Available: https://thescipub.com/abstract/jcssp.2020.863.870

[9] M. Spohn., "An endogenous and self-organizing approach for the federation of autonomous mqtt brokers," in *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, INSTICC. SciTePress, 2021, pp. 834–841.

[10] N. K. Ribas and M. A. Spohn, "A new approach to a self-organizing federation of mqtt brokers," *Journal of Computer Science*, vol. 18, no. 7, pp. 687–694, Jul 2022. [Online]. Available: https://thescipub.com/abstract/jcssp.2022.687.694

[11] J. H. L. Bakker and F. J. Pattenier, "The layer network federation reference point-definition and implementation," in *TINA '99. 1999 Telecommunications Information Networking Architecture Conference Proceedings (Cat. No.99EX368)*, 1999, pp. 125–127.

[12] N. Uramoto and H. Maruyama, "Infobus repeater: a secure and distributed publish/subscribe middleware," in *Proceedings of the 1999 ICPP Workshops on Collaboration and Mobile Computing (CMC'99). Group Communications (IWGC). Internet '99 (IWI'99). Industrial Applications on Network Computing (INDAP). Multime*, 1999, pp. 260–265.

[13] T. Bass, "The federation of critical infrastructure information via publish-subscribe enabled multisensor data fusion," in *Proceedings of the Fifth International Conference on Information Fusion. FUSION 2002. (IEEE Cat.No.02EX5997)*, vol. 2, 2002, pp. 1076–1083 vol.2.

[14] Z. Y. Thean, V. Voon Yap, and P. C. Teh, "Container-based mqtt broker cluster for edge computing," in *2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, 2019, pp. 1–6.