4th International Conference and Workshops on Recent Advances and Innovations in Engineering, 27-29 November 2019
(IEEE Conference Record # 47735)

Container-based MQTT Broker Cluster for Edge Computing

Zhong Ying Thean
Faculty of Enginnering and Green Technology
Universiti Tunku Abdul Rahman
Perak, Malaysia
theanzy1@gmail.com

Vooi Voon Yap
Faculty of Enginnering and Green Technology
Universiti Tunku Abdul Rahman
Perak, Malaysia
yapvv@utar.edu.my

Peh Chiong Teh
Faculty of Enginnering and Green Technology
Universiti Tunku Abdul Rahman
Perak, Malaysia
tehpc@utar.edu.my

Abstract— **This paper presents the implementation of an edge-based MQTT broker cluster, by using a lightweight container orchestration framework and a cluster of single board computers. The main goal is to deliver a low-cost, scalable and lightweight messaging solution to support communication between IoT devices in remote areas, where computational resources and network connectivity are limited. An intermediate layer of cluster server is developed to complement and support the collective processing of distributed MQTT brokers. This paper also presents the evaluations of the MQTT broker cluster in terms of message throughput, end-to-end latency and runtime performance of the system. The broker cluster is able to maintain an average latency of less than 10 milliseconds, and a worst-case latency bound of 52 milliseconds, showing that the microservice-based implementation is feasible for many latency demanding applications in the IoT edge-cloud environment.**

Keywords—MQTT, distributed computing, edge computing

I. Introduction

Traditional cloud-based to Internet of Things (IoT) becomes a bottleneck for IoT integration around the edge due to challenges such as bandwidth requirements, latency, and fault tolerance. Recently, edge and fog computing models are introduced to integrate cloud and sensor-based IoT environments into a variety of latency sensitive applications. Edge technology significantly reduces end-to-end latency, by locally processing data on the network edge, and immediately updates the results over the network [1].

Over the past years, many industry standards have adopted the publish/subscribe paradigm as part of their interfaces to accommodate applications of IoT. The MQTT (Message Queuing Telemetry Transport) protocol is a broker-based publish/subscribe IoT protocol widely used in cloud-based applications [2]. In publish/subscribe communication scenarios, centralized cloud model does not work well for IoT data because the data is usually concentrated and locally consumed around the edge broker [3]. The edge broker reduces unnecessary end-to-end latency by locally consumes and transmits IoT data at the edge of network. However, when the edge server is handling too many resources, it will encounter overloading issues, that may lead to failures and scalability problems. The centralized configuration of the MQTT broker presents a performance bottleneck and a single point of failure. To increase scalability and fault tolerance, publish/subscribe systems such as Siena and Hermes use a network of distributed brokers as intermediaries to disseminate messages between publishers and subscribers [4]. Moreover, the publish/subscribe paradigm attributed with the MQTT protocol is able to support better horizontal scalability due to its asynchronous nature of message communications [5].

Due to resource restrictions on edge servers, a lightweight software framework is required to facilitate deployment of distributed applications on the edge network. Container virtualization technology is able to fulfil the requirements of edge computing, because of its lightweight-ness, mobility, near-native performance, portability, support for heterogeneity, and ease of deployment [6], [7]. Edge-based container orchestration is also a viable lightweight virtualization solution for single board computers (SBCs). Past studies [8], [9] confirm the feasibility of container orchestration on cluster of SBCs, in terms of computational power, energy efficiency, and cost effectiveness. Docker Swarm is a container orchestration framework that is used to manage container packages on a clustered set of Docker hosts. This paper describes how the broker cluster is implemented, using Docker Swarm to deploy distributed applications on a Raspberry Pi SBCs cluster. System availability and fault tolerance can be achieved using Docker Swarm orchestration, by distributing application services through redundant microservice layers.

This paper presents an implementation of a distributed middleware layer to support clustering of MQTT brokers, in order to collectively disseminate messages between MQTT clients. The experiment is performed with a large number of MQTT clients, simultaneously communicating through the MQTT broker cluster. The experimental evaluations confirm the viability of the implemented system in terms of message throughput, latency, resiliency and lightweight-ness. The evaluations also show the performance penalty associated with horizontal scaling of the MQTT broker.

In the remainder of this paper, Section II presents related work. Section III describes the broker cluster architecture and the routing decisions made within the cluster components. Results and evaluations are presented in Section IV. Section V concludes the paper.

II. Related Work

MQTT is widely used in many internets and M2M applications. IoT platform such as Microsoft Azure Hub and Amazon IoT also offer services relied on MQTT. Several implementations of enterprise MQTT broker clusters such as HiveMQ, AWS IoT, and IBM Bluemix are available, but they are closed systems and only deployed in a central cloud data centers, thus their implementations are very limited. The implementation of an edge-based SBC broker cluster can accommodate use cases in area where high-end installations are not an option.

Unfortunately, a single MQTT broker does not scale well with large numbers of sensors. The central broker configuration presents a bottleneck that causes a broker queuing delay as the load increases. Open source libraries such as Mosquito suffers from single-point-of-failure. Many MQTT brokers including Mosquitto can be configured at deployment time to link parts of their topics tree structure to a remote bridge broker in order to publish to and receive messages through the bridge [10]. EMMA [10] uses dynamic bridging tables between MQTT brokers, to dynamically reroute the connection between clients and brokers, according to proximity and QoS index of the connection.

Jararweh et al. [11] present a mobile edge computing (MEC) framework, which uses an SDN-based system to deliver ubiquitous cloud services to mobile users. Their evaluations demonstrate how network overhead and resource consumptions can be reduced with the implementation of decentralized SDN local controllers in distributed MEC networks, as compared to a global centralized SDN controller. DM-MQTT [12] uses bi-directional SDN approach to establish multicast paths between distributed MQTT brokers that subscribe to the same topic, without going through the centralized broker. By doing so, the implementation of DM-MQTT is able to reduce transmission delay and network usage by a significant amount.

ILDM appropriates the implementation of MQTT brokers into a distributed edge environment [3]. An intermediary node is placed between a broker and a client to support data transfer between heterogeneous brokers located on multiple edge networks. However, the ILDM nodes cause of additional delays and network congestion, due to multiple routing hops and indirect data flow between broker nodes. The authors of ILDM also presents subscription and publication diffusion techniques to facilitate dissemination of messages across the distributed edge brokers. Nucleus [13] uses stateless MQTT brokers and a separate distributed storage to split the networking functionality of the broker from the state information. The authors address the resiliency of MQTT broker system by providing data redundancy to the state information maintained in a shared cache. This way the cost of fault tolerance is reduced because the broker can fail anytime while the data is not affected.

Shen and Tirthapura [14] present a self-stabilization algorithm for publish/subscribe system to maintain the distributed routing state and recover from failures. A node periodically exchanges its routing state with neighboring nodes and synchronizes its local state with the neighbor's routing state. Event retransmission is a common message loss-tolerance approach use in many messaging solutions. Modern messaging solutions such as Kafka and Flink implement publication resend to tolerate publication loss [15]. Publication lost can be detected by exchanging acknowledgement messages between message brokers [16].

III. Methodology

A. Broker Cluster Architecture

Fig. 1(a) illustrates the architecture of the broker cluster in an edge cloud environment. The mid-tier server, as depicted in Fig. 1(b), facilitates cooperative processing and communication within the network to make routing decisions based on routing state information about their subscriptions. It also decouples the MQTT broker's operations from the clients. It uses a packet filter module to filter and decode incoming MQTT packets before sending them to the worker threads, for the processing of MQTT state information. The system can use any standard MQTT broker application without recompilation or relinking of the broker software.

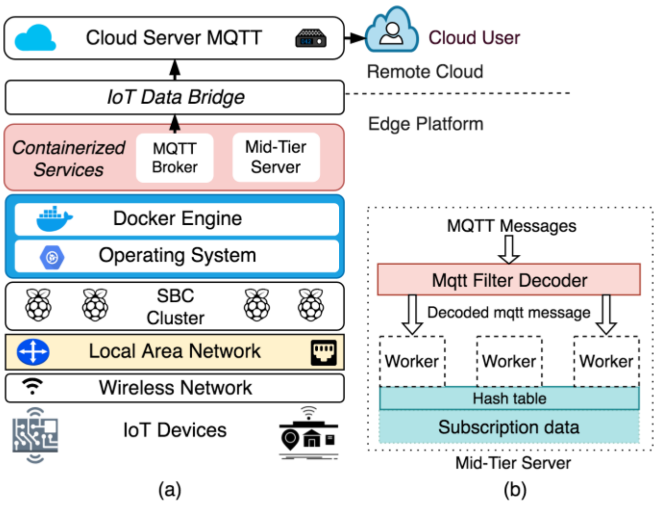

Fig. 1. Edge cloud MQTT broker cluster architecture

Docker Swarm is used to schedule and update Docker containers in the cluster nodes. The load balancer presents the frontend connection for clients and evenly distributes MQTT workload among backend MQTT broker nodes. To avoid unbalanced load among the brokers, the load balancer uses least connected algorithm so that new clients will be routed to the least loaded servers. The IoT data bridge implements MQTT bridging to integrate edge servers with the central cloud, where remote cloud users can obtain sensor data through a cloud broker.

B. Topology Construction and Routing Scheme

Subscription flooding approach is used for topology construction and event routing as the network size is small and only requires one hop to reach remote nodes. Types of routing state are the information about the neighbors in the overlay, the forwarding table, the neighbor subscription lists, recovery queues, and a queue of in-transit publish messages. The routing table structure maps the ip address of neighbor brokers, and their subscribed topics. The recovery table stores a buffered queue of lost publications to facilitate message retransmission. The neighbor subscriptions contain all topics subscribed by neighbor brokers.

The overlay information about the cluster is maintained by a gossip-based membership protocol. The membership protocol implements the SWIM protocol which maintains membership information of nodes in the overlay and detects node failures [17]. When a new broker node joins the group, existing broker nodes send a list of their locally subscribed topics to that new member so that the decentralized structure information is kept synchronized.

Fig. 2 depicts the sequence for subscription management and event routing when message of the same topic reaches two different broker nodes. A subscription that reaches a broker is advertised to all neighbor nodes. New publications are matched at each broker against its stored subscriptions in the routing table.

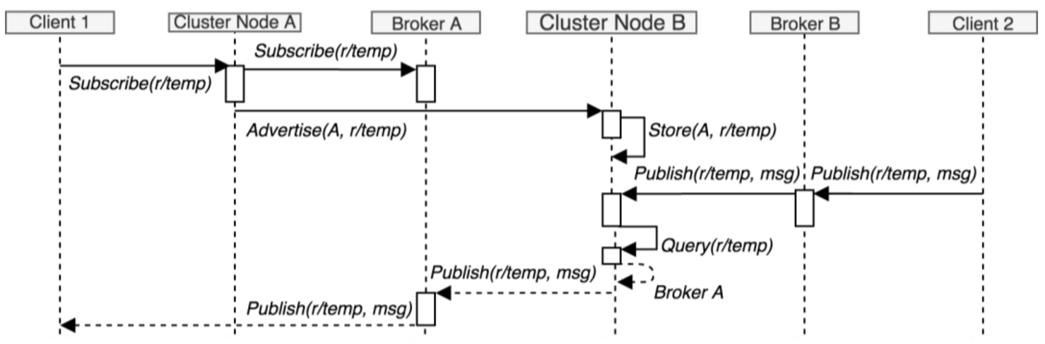

Fig. 2. Message Routing Sequence.

The fault mechanism in the system protects against node crashes. The system however does not handle transient or Byzantine faults. The system resynchronizes the information of global routing states after a node failure to maintain consistency across the cluster. Apart from that, the local subscription list is periodically advertised to all other broker nodes in every 10 seconds to improve consistency of the distributed state information.

C. *Publication Resend*

The client IDs, alongside with their subscribed topic, previously held by a failed broker are used for retransmission of publication. This information is advertised to all the other nodes in the cluster, so that they can update their local list of members. For each incoming subscription, the client id and topics will additionally be checked in a message recovery module. This module stores message queues of all publications failed to be forwarded to a remote broker. If the client is previously held by a failed broker and is not a resubscription by the client itself, the system will retransmit all of the publications in the corresponding message queue. The system also assumes that all publications forwarded to a non-faulty broker will be successfully delivered.

D. *Implementation*

In this research work, the implementation uses a cluster of Raspberry Pi 3 single board computers (SBCs) as the hardware infrastructure to run the container services. Each board runs Hypriot OS and uses Docker engine to deploy application services. The broker component uses Mosquitto MQTT broker and a customized middleware server for distributed coordination among brokers in each node. The cluster uses a star network topology, with one switch acting as the core of the star and other switches link the core to the RPIs. MQTT client devices access the broker service through Wi-Fi network. The load balancer is implemented using HAProxy which routes all requests at the transport layer. Scalability is achieved through scheduling Mosquitto replicas through Docker Swarm. A Docker compose file is written to describe and schedule the services for each Swarm node. The experimental setup uses five Raspberry Pi SBCs to deploy the Docker Swarm cluster. One Docker Swarm manager node runs the load balancer while the other are worker nodes runs the broker cluster.

IV. Results and Evaluation

The broker cluster is load-tested with up to 1000 pairs of publishers and subscribers, each pair corresponds to a unique topic string. The aim of measuring the latency and throughput is to understand how an increase in the number of sensors affects the broker's performance. Tested metrics are end-to-end latency (push-pull), and throughput (push), CPU time percentage, and RAM usage. Fig. 3 illustrates the Prometheus-based monitoring stack to obtain time series system metrics. Prometheus is used as an aggregator to collect the metrics from several target hosts. Grafana is used to visualize the time series data gathered in Prometheus.

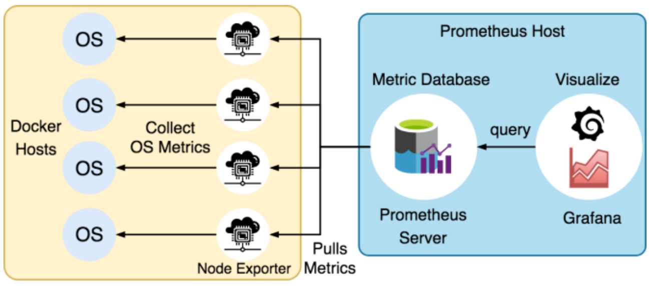

Fig. 3. System metrics monitor

A. *Throughput*

Publish throughput is the number of publication messages processed by the broker per second. To evaluate the performance of the clustered broker, the throughput is compared with a cloud broker and a single broker setting. With reference to Fig. 4, the throughput of the edge-based broker is significantly higher than the cloud broker. However, the clustered broker has less overall throughput compared to a single node broker. This is due to relay elements presented in the load balancer and in between broker components. The 4-way handshake of QoS 2 messages increases the total publication time, which decreases the message throughput.

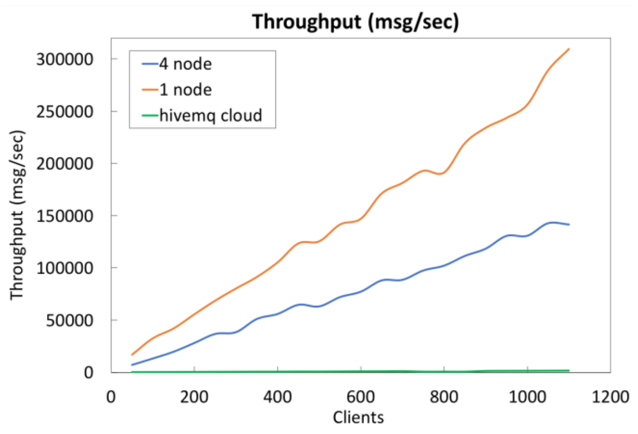

Fig. 4. Publish throughput with QoS 2

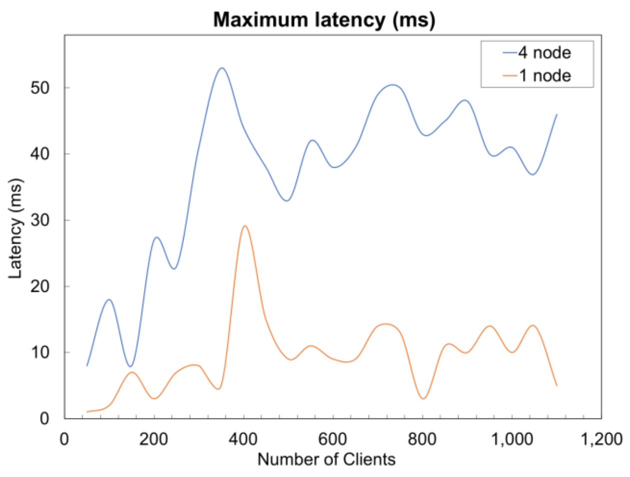

Fig. 5. Average end-to-end latency.

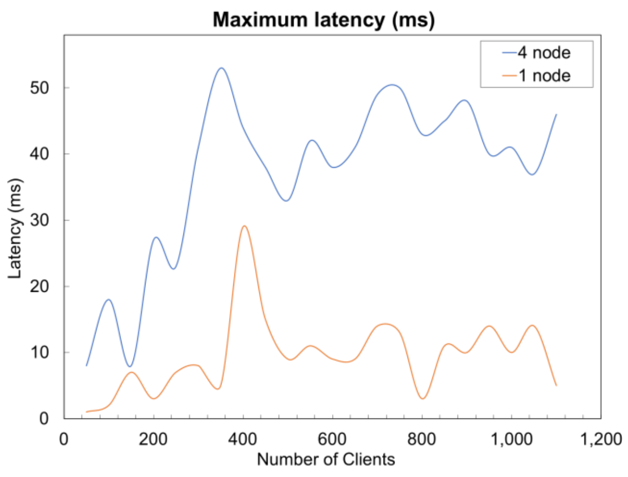

Fig. 6. Worst-case end-to-end latency

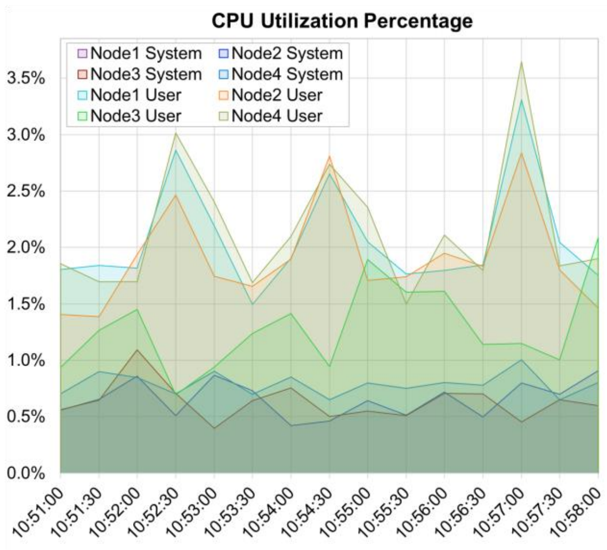

Fig. 7. Clustered broker loaded - CPU time percentage

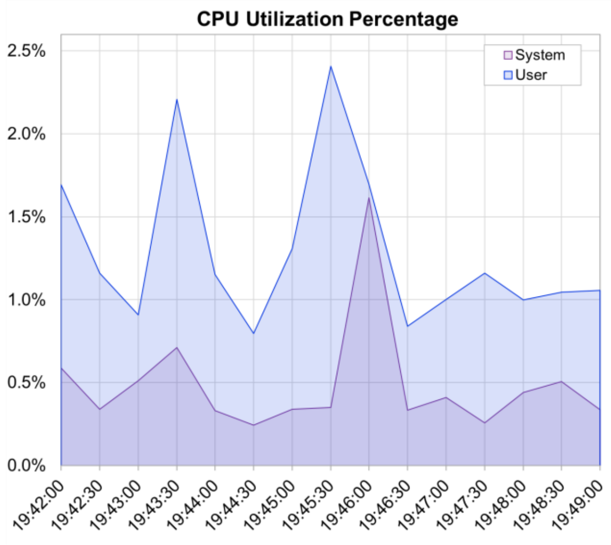

Fig. 8. Single broker loaded - CPU time percentage

B. *Latency*

The end-to-end latency is measured as the time taken for a MQTT message to reach from publisher to subscriber. The results from Fig. 5 show that the broker cluster has higher average end-to-end latencies compared to the single broker setup. As depicted in Fig. 6, the worst-case latencies remained consistent for the 4-node broker cluster. The maximum value of worst-case latency of the 4-node broker cluster is 52ms, which is 30ms higher than that of a single node broker. This is a consequence of relay elements present between broker nodes. Also, the cluster setup has lower degree of data locality than a single node broker. Data locality is present when a publication reaches a broker that also happens to subscribe to a similar topic [3].

C. *Publication Retransmission*

To evaluate the message recovery functionality of the MQTT broker cluster, one of the broker nodes is deliberately stopped during the test run. The test script runs with 8 subscribers and 8 publishers for 2 minutes so that the routing state has enough time to converge. After that, one of the broker nodes, which is currently hosting 2 MQTT subscriber clients, is deliberately failed. The broker cluster is able to recover the failed publications after the subscribers reconnect back to the system. From the observations, the client also receives no duplicate messages. The re-transmission process will filter and send the failed message to the relevant online brokers.

D. *Resource Usage*

From the CPU usage perspective, the both setups show the same scalability. The user's CPU usage peaks at 3.65% regardless of number of clients, as depicted in Fig. 7. The user's CPU usage percentage of the single node broker peaks at 2.47%, as depicted in Fig. 8. The result in Fig. 7 shows that the implemented cluster component costs less than 1.18% of CPU overhead. The graph in Fig. 7 shows irregular CPU time percentages throughout the whole test run. This is due to network I/O operations, which starved the running processes.

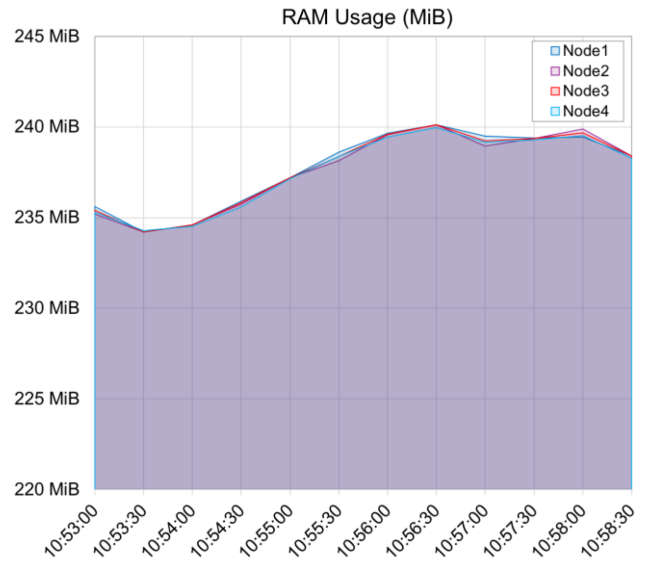

Fig. 9. Clustered broker loaded - memory usage

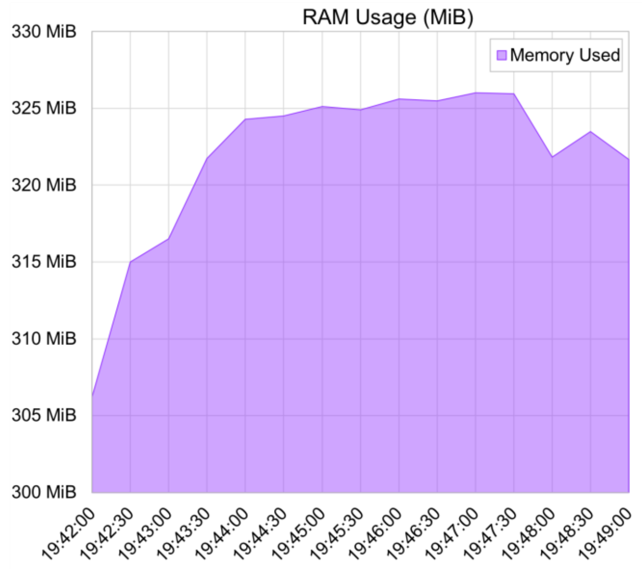

Fig. 10. Single broker loaded - memory usage

The RAM usage shown in Fig. 9 remains below 240 MiB during the entire test run, showing that higher number of sensors produces an almost negligible amount of RAM overhead. Hence, the implementation of the Docker-based MQTT cluster is considered lightweight. This also confirms the scalability of the developed broker cluster in terms RAM usage.

Results from Fig. 10 show that the RAM usage overhead of the single MQTT broker is 21MiB higher, as compared to the clustered broker setup. The observations suggest that replication of MQTT broker helps reduces the memory overhead.

V. Conclusion

In this paper, a microservice-based MQTT broker cluster in an edge-cloud setup is implemented and tested. From the observations in Fig. 6, the worst-case end-to-end latency for is higher for broker cluster setup as compared to the single node broker. This is a trade-off for resiliency as a result of the forwarding delay for the clustered setup when the degree of data locality is low.

The evaluated average end-to-end latency is below 10 milliseconds, which is suitable for most use cases that demand low latency [18]. Also, the cluster implementation will provide predictable latencies, as the latency bounds remained consistent, and variations between each average latency measurements is minor.

The software implementation is lightweight with CPU usage overhead of less than 1.18% and RAM usage overhead of less than 5 MiB for 1000 pairs of clients. Taking robustness into account, performance of the broker cluster is acceptable. The evaluations also show the resiliency of the MQTT cluster, as messages are able to be recovered during broker failure. According to the evaluations, the performance of the broker cluster is sufficient to support the proposed broker scheme.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” IEEE Internet Things J., vol. 3, no. 5, pp. 637–646, 2016.

[2] A. Popov, A. Proletarsky, S. Belov, and A. Sorokin, “Fast Prototyping of the Internet of Things solutions with IBM Bluemix,” in Proceedings of the 50th Hawaii International Conference on System Sciences (2017), 2017.

[3] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, “Dissemination of edge-heavy data on heterogeneous MQTT brokers,” in 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), 2017.

[4] P. R. Pietzuch and J. M. Bacon, “Hermes: a distributed event-based middleware architecture,” in Proceedings 22nd International Conference on Distributed Computing Systems Workshops, 2002, pp. 611–618.

[5] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, “Meeting IoT platform requirements with open pub/sub solutions,” Ann. Telecommun., vol. 72, no. 1–2, pp. 41–52, Jul. 2016.

[6] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, “Orchestration of Microservices for IoT Using Docker and Edge Computing,” IEEE Commun. Mag., vol. 56, no. 9, pp. 118–123, 2018.

[7] R. Morabito and N. Beijar, “Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies,” in 2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops), 2016, pp. 1–6.

[8] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, “A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters,” in 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), 2016, pp. 117–124.

[9] R. Scolati., I. Fronza., N. El Ioini., A. Samir., and C. Pahl., “A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-board Devices,” in Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER, 2019, pp. 68–80.

[10] T. Rausch, S. Nastic, and S. Dustdar, “EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications,” in 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 191–197.

[11] Y. Jararweh et al., “Software-Defined System Support for Enabling Ubiquitous Mobile Edge Computing,” Comput. J., vol. 60, no. 10, pp. 1443–1457, Feb. 2017.

[12] J.-H. Park, H.-S. Kim, and W.-T. Kim, “DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications,” Sensors , vol. 18, no. 9. 2018.

[13] S. Sen and A. Balasubramanian, “A highly resilient and scalable broker architecture for IoT applications,” in 2018 10th International Conference on Communication Systems & Networks (COMSNETS), 2018, pp. 336–341.

[14] Zhenhui Shen and Srikanta Tirthapura, “Self-stabilizing routing in publish-subscribe systems,” IET Conf. Proc., pp. 92-97(5), Jan. 2004.

[15] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, “Lightweight Asynchronous Snapshots for Distributed Dataflows,” CoRR, vol. abs/1506.0, 2015.

[16] R. Chand and P. Felber, “XNET: a reliable content-based publish/subscribe system,” in Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004., 2004, pp. 264–273.

[17] A. Das, I. Gupta, and A. Motivala, “SWIM: scalable weakly-consistent infection-style process group membership protocol,” in Proceedings International Conference on Dependable Systems and Networks, 2002, pp. 303–312.

[18] P. Schulz et al., “Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture,” IEEE Commun. Mag., vol. 55, no. 2, pp. 70–78, 2017.