2018 IEEE International Conference on Cloud Engineering

EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications

Thomas Rausch, Stefan Nastic, Schahram Dustdar

Distributed Systems Group
TU Wien, Vienna, Austria
{rausch, nastic, dustdar}@dsg.tuwien.ac.at

***Abstract*—Publish–subscribe middleware is a popular technology for facilitating device-to-device communication in large-scale distributed Internet of Things (IoT) scenarios. However, the stringent quality of service (QoS) requirements imposed by many applications cannot be met by cloud-based solutions alone. Edge computing is considered a key enabler for such applications. Client mobility and dynamic resource availability are prominent challenges in edge computing architectures. In this paper, we present EMMA, an edge-enabled publish–subscribe middleware that addresses these challenges. EMMA continuously monitors network QoS and orchestrates a network of MQTT protocol brokers. It transparently migrates MQTT clients to brokers in close proximity to optimize QoS. Experiments in a real-world testbed show that EMMA can significantly reduce end-to-end latencies that incur from network link usage, even in the face of client mobility and unpredictable resource availability.**

I. Introduction

Many modern Internet of Things (IoT) scenarios have stringent quality of service (QoS) requirements [1]. Typically, device communication in such environments is facilitated by cloud-based message-oriented middleware (MOM) based on the publish–subscribe model [2]. Edge computing is considered a key enabler for scenarios where such centralized cloud-based platforms are impractical [3], [4]. Edge computing aims to leverage the ever increasing amount of computational resources at the edge of the network and move data processing closer to where data are generated. To facilitate efficient device communication, publish–subscribe MOM can adhere to this principle. Instead of routing all messages to the cloud, brokers can be deployed on edge resources to reduce end-to-end latencies between devices in close proximity.

IoT and edge computing come with a set of challenges. The network topology is highly dynamic and subject to high churn. Clients as well as edge resources are mobile and may unexpectedly leave or enter the system. The dispersion of edge resources adds to the overall complexity of system management mechanisms [5]. Although the Message Queue Telemetry Transport (MQTT) protocol has proliferated as a standard pub/sub platform for IoT applications [6], state-of-the-art MOM solutions based on MQTT fall short of addressing these issues.

In this paper we propose EMMA, a distributed QoS-aware MQTT middleware for edge computing. The contributions of our system are (i) a continuous network-monitoring protocol that allows proximity detection based on network latency, (ii) a mechanism to orchestrate a network of distributed MQTT client gateways and brokers, (iii) a network reconfiguration scheme to optimize QoS during runtime based on node proximity. In an empirical evaluation using a real-world testbed, we show that EMMA can significantly reduce end-to-end latencies caused by message routing, even in the face of changing network topologies and client mobility.

II. Motivation

We briefly summarize two scenarios that motivate the need for edge computing middleware, one from the military domain described by Lewis et al. [7], and another from the mobile health (mHealth) domain described by Nastic et al. [8].

Tactical cloudlets: Mobile handheld devices are increasingly used by soldiers, field personnel and first responders. In tactical environments, such devices can help with tasks such as language translation, face recognition, mission planning and other on-premise decision making processes. In these tactical environments, edge resources such as cloudlets can be hosted on vehicles, drones, or other platforms in close proximity, and may be added on demand [7].

Edge analytics for mHealth: In mass emergencies and disasters, prompt paramedic attention is crucial to save peoples' lives. To guide and improve the decision making process of paramedics, patients are equipped with a wearable sensor that continuously reports vital parameters of the patient to mobile devices carried by paramedics. For on-premise decision making, data have to be processed locally in near real-time. On metropolitan area level, processed health data are useful for follow-up treatments in hospitals. For this type of analytics, data have to be transferred to, e.g., a local cloudlet. Any further long-term data processing, such as using Big Data techniques for complex data analytics, will require the data to be stored in a cloud storage system [8].

These scenarios demonstrate the need for edge computing communication middleware that can provide low end-to-end latencies between devices in close proximity, while still being able to distribute messages to the Cloud or other geographically dispersed locations. Furthermore, the scenarios highlight the characteristics and challenges of edge environments: dynamic network topologies, client mobility and changing resource availability.

0-7695-6371-6/18/31.00 ©2018 IEEE
DOI 10.1109/IC2E.2018.00043

Several questions and problems arise when designing such a middleware. (i) The ad-hoc distribution of brokers to resource constrained devices requires dynamic and efficient management of distributed subscription tables. (ii) Clients should be unaware of the dynamic broker network and should be transparently connected to brokers via gateways. (iii) To determine proximity, the network QoS between nodes needs to be monitored and reported efficiently. (iv) If proximity between clients and brokers changes, client–broker connections have to be reconfigured. (v) When brokers provide similar QoS to a set of clients, load needs to be distributed among these brokers.

III. Related Work

While publish–subscribe (pub/sub) is a well researched topic [9], cloud computing, the IoT, and edge computing, have introduced new challenges and opportunities. In particular, research has shown that current protocols and solutions can not adequately deal with the scalability and QoS requirements of modern IoT scenarios [1]. Also, many pub/sub solutions rely on their own protocols and models [10], [11], [12] that neglect established standards for IoT applications such as MQTT.

QoS awareness in pub/sub overlay networks has been addressed in different works spanning the past two decades [13], [14], [15]. Specifically, these approaches focus on techniques for managing complex networks of brokers using contextual QoS information, and efficient message routing within these networks. Contributions include optimal path selection for message routing based on QoS criteria and supporting mobility by providing efficient on-line re-configuration of overlays.

Scalability of pub/sub middleware under fluctuating load has primarily been addressed using cloud-based solutions. Centralized systems such as Amazon IoT [2], or Dynamoth [16] achieve scalability by providing load balancing in a broker cluster, and elasticity mechanisms for adding and removing broker nodes on demand. These cloud-based systems do not consider proximity of clients or latency incurring from link usage. Load balancing for edge computing is a fairly unexplored topic, and has only recently gained attention in the edge computing community [17].

Some open-source pub/sub brokers, such as JoramMQ [18], Mosquitto[1] and HiveMQ[2], provide basic mechanisms for enabling edge computing applications. For example, Mosquitto can be configured at deployment time to bridge topics, i.e., forward messages of a specific topic, to a centralized broker [19]. However, these mechanisms are all static in nature and do not address mobility or changing resource availability.

Few efforts have been made to engineer holistic solutions for QoS aware pub/sub systems that address the challenges of IoT and edge computing. An et al. [20] present PubSubCoord, a cloud-based coordination system for a distributed broker network. The overlay layer is strictly structured into *edge* brokers and *routing* brokers, coordinated via a layer of ZooKeeper nodes. In a non-peer-reviewed work, Abdelwahab and Hamdaoui present FogMQ [21] which supports migration of broker *clones* at runtime to the edge, thereby enabling low-latency data analytics.

IV. System Design

EMMA is built around MQTT, a pub/sub protocol that has gained newly found attention in the advent of IoT [6], [1]. Due to its lightweight design and minimal network overhead, MQTT is especially well suited for low-bandwidth and low-power environments [6]. We designed EMMA to act as a transparently distributed MQTT broker, enabling existing IoT software that uses MQTT to be seamlessly used. For this first prototype, we considered only part of the MQTT protocol.

A. Architecture

EMMA consists of four core components: gateways, brokers, the controller, and the network monitoring protocol. Figure 1 shows the general architecture of an EMMA deployment. MQTT clients connect to gateways which act as reverse proxies for dynamically connecting clients to brokers. Brokers implement the MQTT server protocol and our dynamic topic bridging approach. The controller acts as a registry, monitoring hub and system orchestrator. Next, we describe the role of each component.

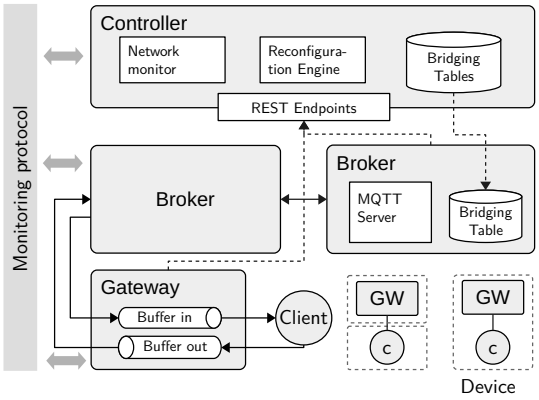


Fig. 1: Overview of the EMMA architecture

a) Gateways: are a key component in enabling mobility of clients and brokers by allowing reconfiguration of connections. Their purpose is to hide EMMA from the actual clients by tunneling MQTT traffic and providing a buffering mechanism during a reconnection process to a different broker. Gateways can be deployed on different physical locations than the clients themselves, and can also handle multiple client connections. They are lightweight, have no code dependencies, and have very little logic to them. A similar concept was proposed by Luzuriaga et al. [22], where intermediary buffers decouple message producers and MQTT publisher clients.

b) Brokers: implement the MQTT server protocol [23], i.e., manage topic subscriptions and disseminate published messages to subscribers. They also act as topic bridges to forward messages to other brokers that have subscribers to those topics. To that end, *bridging tables* are synchronized between brokers via the controller. Bridging tables specify which brokers have at least one subscriber to a specific topic.

c) Controller: The controller is the orchestration component of the system where gateways and brokers register when they enter or leave the network. It maintains the state of the network as a graph data structure. Formally, the network $N = (B, C, E)$ is a bipartite graph containing broker nodes B, client nodes C, and connections between clients and brokers as edges E (which we call a *link*). In this context, a client c represents a gateway. The reconfiguration engine of the controller detects proximity between gateways and brokers based on network latency and instructs gateways to reconnect to different brokers to optimize QoS. Furthermore, it balances load between brokers that provide similar QoS to clients.

d) Monitoring: The monitoring protocol is a lightweight binary protocol to allow distributed monitoring of network QoS. Each EMMA component implements the protocol. The monitoring protocol is also used to instruct gateways to reconnect to different brokers.

V. Coordination Mechanisms

A. Quality of Service Monitoring Protocol

To reason about the network at runtime, we implement a distributed asynchronous network monitoring protocol on top of UDP. Messages of the protocol are encoded in a lightweight binary format, summarized in table Table I.

The general process of the protocol is as follows. The controller sends a `QOSREQ` packet containing an ID, and the measurement target (a broker) to a gateway. The gateway then sends 10 `PINGREQ` packets to the broker in an interval of 250 ms. Each `PINGREQ` packet contains an ID, which is returned by the broker in a `PINGRESP` packet. The gateway records the sent and received timestamps for each packet and sends back the average latency as a `QOSRESP` packet to the controller. Calculating and using other metrics such as jitter and packet loss is part of our future work.

B. MQTT Broker and Gateway Network Orchestration

1) Dynamic Topic Bridging: Because we assume dynamic availability of brokers and mobility of clients, we cannot rely on static bridging configurations, e.g., like Mosquitto does [19]. We therefore extend the standard MQTT protocol with dynamic topic bridging. When a client publishes a message into a given topic, brokers first broadcast the message to all connected subscribers according to the MQTT protocol, and then forward the message to other brokers that have at least one subscriber to the respective topic. Brokers inform the controller about changes of their local subscription tables when necessary (e.g., if a client subscribes to a new topic), and the controller propagates relevant changes to the other brokers.

TABLE I: Packets of the monitoring protocol

Name	Description	Size
QOSREQ	Network QoS measurement request	13
QOSRESP	Network QoS measurement response	9
PINGREQ	Ping request	5
PINGRESP	Ping response	5
RECONNREQ	Request a gateway to reconnect to a broker	47
RECONNACK	Gateway acknowledges the reconnect	47

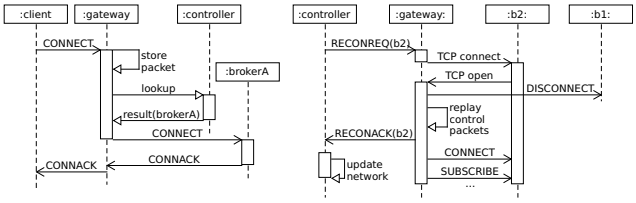


Fig. 2: Connection (left) and reconnection (right) procedures

2) Orchestrating Client Connections: Instead of connecting to a broker directly, MQTT client traffic is tunneled through a local gateway. When a client sends an MQTT `CONNECT` packet to a gateway, the gateway initially queries the controller for a broker to connect to. The gateway stores all MQTT control packets sent by the client related to establishing a broker connection and subscriptions (`CONNECT`, `SUBSCRIBE`, `UNSUBSCRIBE`). The controller may decide at runtime to migrate the connected clients to a different broker. To that end, the controller sends a `RECONNREQ` packet to the gateway via the monitoring protocol containing the host and port of the target broker. The gateway asynchronously opens a connection to the new broker, disconnects from the old broker once the new connection is established, informs the controller by sending a `RECONNACK` packet, and then places the stored control packets into the send buffer dequeue. To avoid packet loss during reconnection of clients to new brokers, a gateway maintains, for each client–broker tunnel, two buffers that buffer incoming messages from the client and broker respectively. Any messages sent from the client during a reconnection process are buffered into a dequeue. Once the connection to a new broker is established, the recorded MQTT control packets are placed into the head of the dequeue, the old connection is closed, and the buffer, which includes the control packets, is flushed. Figure 2 shows sequence diagrams of the connection and reconnection procedure.

C. Network Reconfiguration and QoS Optimization

The network is reconfigured by first examining the current QoS of the network, selecting potential broker candidates for each client in a way that will optimize QoS for those clients, and then migrating clients to their designated brokers. The reconfiguration engine of the controller is scheduled to run at a fixed interval of 15 seconds. Figure 3 shows the state of a network before and after a reconfiguration. Values of links indicate the proximity (latency in milliseconds). Arrows indicate message flow. Clients c_1 and c_2 are migrated from broker b_1 to b_2. A topic bridge between the two brokers is created automatically through the subsequent reconnection and subscription procedures described earlier.

When brokers provide similar QoS to clients, it is important to (i) avoid migrating clients due to slight variability of latency, and (ii) balance load between those brokers. To that end, we stratify brokers into latency groups,

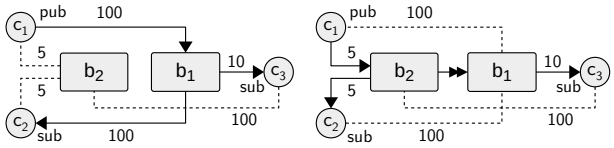


Fig. 3: Network before and after reconfiguration

based on the premise that connecting to any of the brokers in a group is acceptable in terms of QoS. We define the latency groups as millisecond-intervals I, and set $I = \{[0,2), [2,5), [5,10), [10,20), [20,50), [50,100), [100,200), [200,500), [500,1000), [1000,)\}$. That means, for example, if a broker b provides a client c a latency of 4, it would fall into the interval $I_2 = [2,5)$ and therefore into group 2. For balancing load in a group we implemented a simple strategy based on the amount of clients connected to a broker.

The basic algorithm for reconfiguring the network is outlined in Algorithm 1. The network $N = (B, C, E)$ is a bipartite graph of brokers B and clients C as described in Section IV-A. To avoid migrations when no significant load balancing would occur, we introduce the migration threshold θ, which defines the minimum percentage of connections that have to change within a group of candidate brokers in order to trigger a reconnect. By default, we set $\theta = 0.1$.

Algorithm 1 Network reconfiguration

In: Network $N = (B, C, E)$
In: Migration threshold θ

for all $c \in C$ **do**
  $b_c \leftarrow$ currentBroker(c)
  $B'_c \leftarrow$ brokersInLowestLatencyGroup(N, c)
  $b'_c \leftarrow b \in B'_c$ s.t. connCnt(b) is minimal
  if $b_c = b'_c$ **then**
    continue
  end if
  if $b_c \in B'_c$ **then**
    $\delta = \theta \cdot \sum_{b \in B'_c} \text{connCnt}(b)$
    if connCnt(b'_c) $+\delta \geq$ connCnt(b_c) **then**
      continue
    end if
  end if
  migrate(c, b'_c)
end for

VI. Evaluation & Implementation

To show the efficacy of our approach, we implement a prototype of the EMMA system and evaluate it in a real-world testbed. We run an experimental scenario that emulates the scenarios described in Section II to show that the system can (i) deal with clients and brokers unexpectedly entering and leaving the network, (ii) dynamically bridge topics only when required, (iii) reconfigure connections to optimize latency for clients in close proximity, and (iv) balance load between brokers that provide similar QoS to clients.

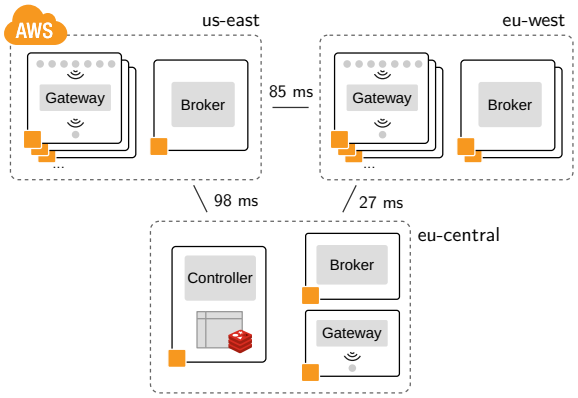


Fig. 4: Evaluation environment in AWS EC2

A. Prototype Implementation

We implement EMMA as a set of modular components written in the Java programming language. To date, the codebase comprises 217 classes and 10.3k effective LOC. The code is open source and published in our code repositories[3]. For distributing bridging tables we use Redis[4] and its keyspace notification feature to propagate updates only when necessary.

B. Experiment Setup

We now present the details of our evaluation environment, the experimental scenario, and deployment details.

1) Testbed: Our evaluation environment consists of multiple Amazon EC2 virtual machines that span three different AWS data centers. Figure 4 illustrates the setup and shows latencies between regions. Latencies within regions are in the sub-millisecond range. We use the following EC2 instances types for the components. The controller runs on a t2.large (2 vCPUs, 4 GiB RAM) instance. For broker nodes, we use t2.medium instances (2 vCPUs, 2 GiB RAM). Gateways and clients share t2.micro instances (1 vCPUs, 1 GiB RAM).

2) Scenario: The experimental scenario emulates the real-world edge computing scenarios presented in Section II. In this particular experiment, we spawn client groups across two regions. Each client group consists of 10 VMs, each hosting a gateway, 1 subscriber and 7 publishers that exchange messages in a topic named like the region they are deployed in, namely eu-west and us-east. To show that the system can dynamically create topic bridges when necessary, a publisher and subscriber pair is deployed in each region that communicate in the topic *global*. A single subscriber to this topic is also deployed in eu-east where the initial broker and controller reside. To demonstrate the orchestration mechanisms, we trigger the following events manually at runtime:

1) Clients appear that communicate in topic *global* (one publisher and subscriber in both us-east and eu-west. One subscriber in eu-central)
2) Client group appears in the us-east region
3) Broker spawns in eu-west (1)
4) Client group appears in the eu-west region

5) Broker spawns in us-east
6) Broker spawns in eu-west (2)
7) Subscriber to topic *global* in eu-central disappears
8) Broker shuts down in us-east

3) Clients & Load Generation: For the purpose of generating load and recording message statistics, we developed a general purpose framework for benchmarking publish–subscribe systems which is also open source[5]. Messages generated by the application contain a payload in the JSON format that has a total of 118 bytes (including JSON overhead). They contain a UUID, a timestamp the message was sent, and a dummy payload with 14 bytes as placeholder for a sensor reading. We configure each publisher to generate messages at a fixed rate of 10 messages per second.

C. Experiment Results

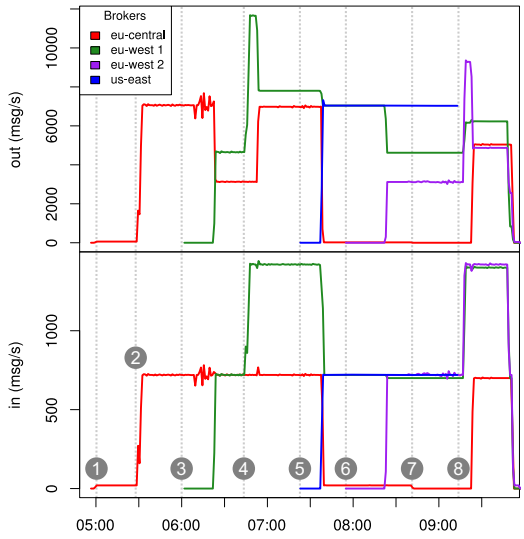
Figure 5 shows the main results from our experiment. Figure 5a shows the message throughput of the brokers during their lifetime. The circles and dotted lines indicate the events described previously. The x-axis indicates the time (in minutes) of the experiment, each labeled tick is 60 seconds apart. Figure 5b and Figure 5c show the average end-to-end latency of messages in the respective topics, aggregated every second over all subscribers of that topic.

1) Throughput & load balancing: Figure 5a shows how the rebalancing mechanism and the dynamic bridging approaches behave. At event 4, when the client group appears in eu-west, the output rate shows that it takes two rebalancing iterations to fully balance connections between the two active brokers. Because messages are bridged between the two brokers, the input rate does not change. However, as the output rate makes apparent, the balancing can significantly reduce the strain of multicast on a single broker. The input rate of the eu-central broker after event 7 shows that, once there are no subscriptions to a topic at a specific broker, the subscription tables are propagated and messages are no longer bridged to that broker.

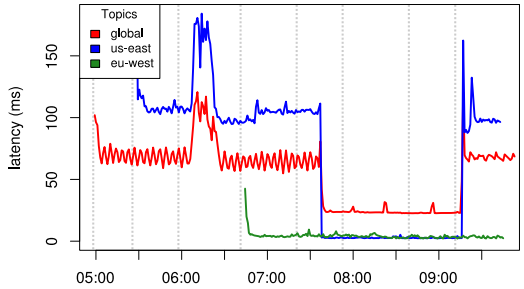
At event 8, when the broker in us-east shuts down, clients previously connected to that broker are immediately migrated to other available brokers. All three currently running brokers are in the same latency group for the clients deployed in the us-east region. As the output throughput graph shows, load is balanced between the three brokers.

2) Topic latencies: Figure 5b shows how rebalancing and migrations affect the end-to-end latencies between clients. In particular, the graphs reveal the cost of migrating clients. At event 3, when a new broker spawns and migration of clients begins, the message latencies spike. This is the result of a combination of the gateway buffering mechanism during reconnect, and the Java warm-up phase of newly spawned brokers. However, as the graph shows, the latency stabilizes a few seconds after the migration is complete, and the load is shared between the brokers eu-central and eu-west 1. The second iteration has a less drastic effect.

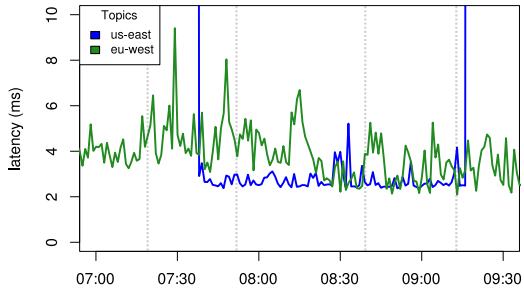
At event 4, the graph for the eu-west topic shows that clients in the region immediately connect to the broker closest to them. Consequently, the load balancing engine moves some



(a) Broker message throughput



(b) Aggregated topic latencies



(c) Latencies during intra-region communication

Fig. 5: Experiment results

of the clients connected from us-east back to the eu-central broker. In this case, it took two balancing iterations to fully balance the connections.

At event 5, when the broker in us-east is spawned, all clients in the region migrate to that broker. This includes the publisher and subscriber in that region communicating via the *global* topic, which is why the average latency also drops dramatically. A major source of latency in this topic is the round-trip time between the us-east region and the brokers in eu regions. Once clients within the region communicate via a

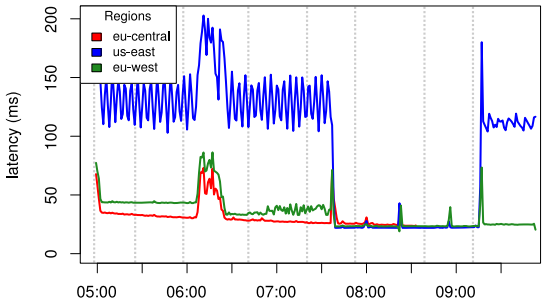


Fig. 6: Average latency for *global* topic subscribers

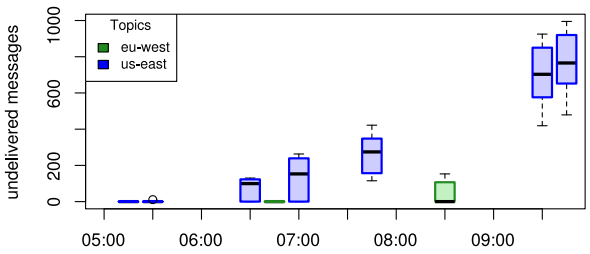


Fig. 7: Message loss for clients in the respective regions

local broker, only messages that have to be forwarded are sent over the us–eu link. At event 8, when the broker in us-east shuts down, latencies spike due to reconnection buffering, and then stabilize at their previous levels.

Figure 5c shows the time window between event 5 and 8, where both client groups have a broker in their respective region. As the graphs show, the latencies average at around 3-5 ms. The variance in the eu-west topic shows how load balancing affects latencies for devices in close proximity. These results also indicate that messages are bridged efficiently, meaning that the system can provide low end-to-end latencies for clients in close proximity, and forward messages to other brokers that provide similar QoS with minimal overhead.

3) Global message dissemination: In each region, one subscriber to the *global* topic is deployed. Figure 6 shows the average end-to-end latency for each subscriber in the respective region during the experiment. As the us-east graph shows, messages sent from the us-east region have, at first, the highest round trip time, causing the high fluctuation in that region. At the later stages after event 5, where a broker is present in each region, message latencies average at roughly the average link latencies between clients and their local brokers. Looking at the subscriber in eu-central, we also observe that, even when topics are bridged from other regions, the latencies do not significantly change. This shows that the overhead of bridging messages to geographically dispersed locations is minimal even under broker load.

4) Message loss: The gateway buffering approach avoids message loss for publishers during a reconnection procedure. However, messages published while subscribers are migrated may not be delivered to these subscribers. Figure 7 shows the total amount of undelivered messages in both client groups at given points in time during the experiment. The boxes show the distribution of undelivered messages across the 10 subscribers of the respective group. Boxes are drawn in points in time where the amount of undelivered messages changed. Each client group consists of 70 publishers that publish at a frequency of 10 msg/s, totaling at 700 msg/s. In the us-east topic, a total of 177,269 messages were published. As the last box for the topic shows, subscribers experienced a total message loss of about 765 messages, i.e., 0.43% per subscriber. For the *global* topic, message loss was minimal.

D. QoS Monitoring Network Usage

QoS monitoring comes at the cost of using the network's bandwidth. We present a sample calculation that shows that the use grows linearly with the amount of brokers and clients in the network. Measuring the QoS between two nodes involves sending a total of 22 UDP packets. A UDP packet in IPv4 Ethernet has 48 bytes of overhead. Hence the UDP overhead for each measurement totals at 1058 bytes. The request/response packets sent between gateway and controller are 13 byte and 9 byte respectively. Then, 10 ping messages are sent to the broker, and, ideally, 10 response messages are returned, totaling 100 bytes. Including the UDP overhead, this means a total of 1180 bytes per measurement.

Our prototype implements a fixed rate approach, where QoS is measured every 15 seconds. In a network of 100 brokers, this would mean roughly 7.9 kB/s network usage for a gateway. There is a lot of potential to optimizes these values, e.g., by adapting the update measurement frequency based on proximity. More inquiry on the requirements of network balancing algorithms on monitoring protocols is needed.

VII. Conclusion and Future Work

In this paper, we presented EMMA, a QoS aware MQTT middleware for edge computing. EMMA is a step towards a holistic message-oriented middleware for edge computing applications that aim to satisfy the stringent QoS requirements imposed by modern IoT scenarios. We have shown that EMMA can provide low-latency communication for devices in close proximity, while allowing message dissemination to geographically dispersed locations at minimal overhead costs. Our network reconfiguration mechanism enables client mobility, dynamic broker provisioning, and broker load balancing. Gateways allow existing MQTT client infrastructure to transparently connect to the system.

Future work includes a complete implementation of the MQTT protocol, including its message reliability guarantees (*at least once*, and *exactly once*), as well as automatic resource discovery and elasticity control mechanisms for autonomous broker deployment to the edge.

Acknowledgment

This work is partially supported by the Austrian Federal Ministry of Science within the CPS/IoT Ecosystem project and by TU Wien research funds.

References

[1] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting iot platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 41–52, feb 2017.

[2] J. Barr, "Aws iot – cloud services for connected devices," *AWS Blog*, 2015. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/

[3] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[4] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[5] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. June, pp. 30–39, Jan 2017.

[6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[7] G. Lewis, S. Echeverria, S. Simanta, B. Bradshaw, and J. Root, "Tactical cloudlets: Moving cloud computing to the edge," in *2014 IEEE Military Communications Conference*. IEEE, oct 2014, pp. 1440–1446.

[8] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, jun 2003.

[10] G. Siegemund, V. Turau, and K. Maâmra, "A self-stabilizing publish/subscribe middleware for wireless sensor networks," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, Mar 2015, pp. 1–8.

[11] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish–subscribe systems," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014.

[12] V. Turau and G. Siegemund, "Scalable routing for topic-based publish/-subscribe systems under fluctuations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 1608–1617.

[13] N. Carvalho, F. Araujo, and L. Rodrigues, "Scalable qos-based event routing in publish-subscribe systems," in *Fourth IEEE International Symposium on Network Computing and Applications*, jul 2005, pp. 101–108.

[14] Y. Chen and K. Schwan, "Opportunistic overlays: Efficient content delivery in mobile ad hoc networks," in *Middleware 2005: ACM/IFIP/USENIX 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005. Proceedings*. Springer Berlin Heidelberg, 2005, pp. 354–374.

[15] M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin, "Efficacy of techniques for responsiveness in a wide-area publish/subscribe system," in *Proceedings of the 11th International Middleware Conference Industrial Track*, ser. Middleware Industrial Track '10. ACM, 2010, pp. 40–45.

[16] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle, "Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, jun 2015, pp. 486–496.

[17] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue, "An approach to qos-based task distribution in edge computing networks for iot applications," in *2017 IEEE International Conference on Edge Computing (EDGE)*, June 2017, pp. 32–39.

[18] ScalAgent, "Jorammq, a distributed mqtt broker for the internet of things," 2014.

[19] M. Garcia, "How to bridge mosquitto mqtt broker to aws iot," *The Internet of Things on AWS – Official Blog*, 2016. [Online]. Available: https://aws.amazon.com/blogs/iot/how-to-bridge-mosquitto-mqtt-broker-to-aws-iot/

[20] K. An, S. Khare, A. Gokhale, and A. Hakiri, "An autonomous and dynamic coordination and discovery service for wide-area peer-to-peer publish/subscribe: Experience paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '17. New York, NY, USA: ACM, 2017, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/3093742.3093910

[21] S. Abdelwahab and B. Hamdaoui, "Fogmq: A message broker system for enabling distributed, internet-scale iot applications over heterogeneous cloud platforms," *CoRR*, vol. abs/1610.0, 2016. [Online]. Available: http://arxiv.org/abs/1610.00620

[22] J. E. Luzuriaga, J. C. Cano, C. Calafate, P. Manzoni, M. Perez, and P. Boronat, "Handling mobility in iot applications using the mqtt protocol," in *2015 Internet Technologies and Applications (ITA)*, Sep 2015, pp. 245–250.

[23] A. Banks and R. Gupta, "Mqtt version 3.1. 1," *OASIS standard*, vol. 29, 2014.

Notes

[1] https://mosquitto.org
[2] http://www.hivemq.com
[3] https://git.dsg.tuwien.ac.at/emma/emma
[4] https://redis.io
[5] https://git.dsg.tuwien.ac.at/emma/pubsub-benchmark