PAPER Special Section on Parallel and Distributed Computing and Networking

Interworking Layer of Distributed MQTT Brokers*

Ryohei BANNO[†a)], ***Member***, **Jingyu SUN**[††], ***Nonmember***, **Susumu TAKEUCHI**[††], ***and*** **Kazuyuki SHUDO**[†], ***Members***

SUMMARY MQTT is one of the promising protocols for various data exchange in IoT environments. Typically, those environments have a characteristic called "edge-heavy", which means that things at the network edge generate a massive volume of data with high locality. For handling such edge-heavy data, an architecture of placing multiple MQTT brokers at the network edges and making them cooperate with each other is quite effective. It can provide higher throughput and lower latency, as well as reducing consumption of cloud resources. However, under this kind of architecture, heterogeneity could be a vital issue. Namely, an appropriate product of MQTT broker could vary according to the different environment of each network edge, even though different products are hard to cooperate due to the MQTT specification providing no interoperability between brokers. In this paper, we propose Interworking Layer of Distributed MQTT brokers (ILDM), which enables arbitrary kinds of MQTT brokers to cooperate with each other. ILDM, designed as a generic mechanism independent of any specific cooperation algorithm, provides APIs to facilitate development of a variety of algorithms. By using the APIs, we also present two basic cooperation algorithms. To evaluate the usefulness of ILDM, we introduce a benchmark system which can be used for both a single broker and multiple brokers. Experimental results show that the throughput of five brokers running together by ILDM is improved 4.3 times at maximum than that of a single broker.

key words: *MQTT, publish/subscribe, distributed systems, IoT, edge computing*

1. Introduction

MQTT has attracted much academic and industrial interest in recent years as one of the key technologies of IoT services [1]. It is a protocol of topic-based pub/sub messaging, in which messages are exchanged through logical channels called "topics", as shown in Fig. 1. MQTT uses a server called "broker" to manage topics and mediate between publishers and subscribers. This paradigm provides decoupling between clients, e.g., each publisher has no concern with the location of subscribers that will receive its message [2].

Although typical IoT systems place an MQTT broker in the cloud [3] as shown in Fig. 2 (a), this centralized architecture can cause some issues due to the following characteristics of IoT data:

- A massive volume of data is generated at the network edge, rather than in the cloud.
- Data have high locality; data generated in an area are often utilized in the same area.
- Data are much utilized for event-driven services, so that high real-time performance is indispensable.

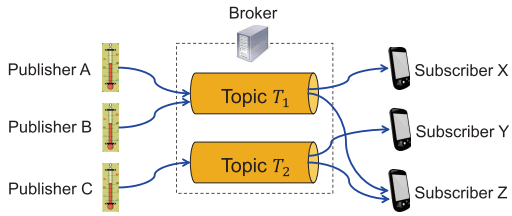
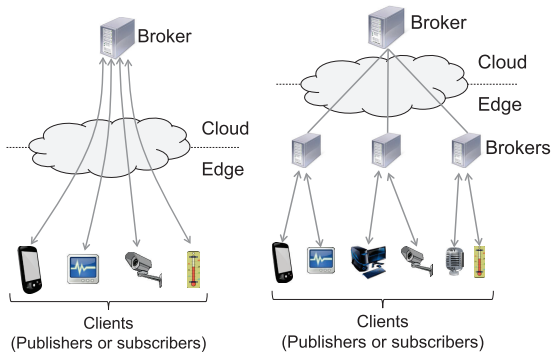


Fig. 1 Message flows of topic-based pub/sub.



(a) Cloud-based. (b) Edge-based.

Fig. 2 Architectures for handling edge-heavy data.

Such characteristics are called "edge-heavy" [4]. The problems of managing edge-heavy data with the above cloud-based architecture are resource consumption and latency. That is, heavy load is concentrated with oppressing cloud resources such as the network bandwidth, as well as latency tends to be long due to the distance between devices and cloud data centers.

Cooperation of multiple MQTT brokers placed at the edges is a solution to the problems. In this architecture, shown in Fig. 2 (b), the cloud broker does not need to communicate with all clients directly, so that the consumption of cloud resources is reduced and consequently overall throughput is increased. Furthermore, it makes latency lower for locally consumed data, because the edge brokers are closer to IoT devices. There could be several variations of this architecture; cascaded edge brokers for han-

Manuscript received January 7, 2019.
Manuscript revised May 28, 2019.
Manuscript publicized July 30, 2019.
[†]The authors are with Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
[††]The authors are with NTT Network Innovation Laboratories, Tokyo, Musashino-shi, 180–8585 Japan.
*This is a paper on system development.
a) E-mail: banno@computer.org
DOI: 10.1587/transinf.2019PAK0001

Copyright © 2019 The Institute of Electronics, Information and Communication Engineers

dling densely placed devices, cooperation without the cloud broker if there is no client in the cloud, etc.

Assuming the edge-based architecture, especially assuming the exchange of IoT data among various fields such as homes, factories, and offices, there is an issue of heterogeneity. Namely, an appropriate product of an MQTT broker is different according to an environment of each network edge or an organization/individual responsible for its operation[†]. For example, industrial computers in factories and set-top boxes in small offices are both possible to be edge servers for placing brokers.

Developing a number of broker products from scratch for each environment or responsible organization/individual results in an increase in costs, including costs for continuing maintenance. Furthermore, IoT projects orienting horizontal linking of data generally has a problem of the chicken-or-egg causality dilemma; increasing the number of partners who has data is desirable to enhance the value of the project, while the value of the project is needed to increase the number of partners. Hence, obtaining a clear estimation of cost-effectiveness in advance is difficult, so that suppressing the cost for enabling the small start is quite important.

To enable brokers placed at the heterogeneous edges to cooperate with each other at low cost, utilizing existing broker products is desirable. There are lots of choices: open source or proprietary, software or embedded appliance, difference in supported OSs, difference in functional features, and so on. However, different products are hard to cooperate. Although some of existing products have functions of cooperation between multiple brokers, e.g., "bridge" of Mosquitto [5] as described in Sect. 7, the standardized specification of MQTT does not provide interoperability between brokers [6]. On the other hand, implementing cooperation functionality into all of the products, which are different in various aspects such as the programming language, results in huge development and maintenance cost.

In this paper, we propose Interworking Layer of Distributed MQTT brokers (ILDM), which enables arbitrary kinds of brokers to cooperate with each other. This means that ILDM enables deploying the edge-based architecture by utilizing existing heterogeneous broker products to be easy. ILDM is designed as a generic mechanism independent of any specific cooperation algorithm. By providing APIs, it facilitates rapid development of a variety of algorithms. We also propose two basic algorithms which have different characteristics to show the feasibility of ILDM and the versatility of its APIs.

To evaluate the usefulness of ILDM, we introduce a benchmark system which can be used for both a single broker and multiple brokers. Our benchmark method ensures that error ratio of resulted performance is not more than five percent.

The contributions of this paper are fourfold:

- First, we give an architecture of ILDM-based cooperation and the APIs.
- Second, we present two basic cooperation algorithms.
- Third, we provide a practical method for benchmark of MQTT brokers.
- Fourth, we show the usefulness of ILDM through experiments.

This paper is an extended version of a paper presented at CloudNet 2017 [7]. The differences include a detailed information of the architecture and the APIs (Sect. 2), the implementation of cooperation algorithms (Sect. 4), and the components of the benchmark system (Sect. 5).

The rest of this paper is organized as follows. Section 2 gives an overview of MQTT protocol, and illustrates a fundamental idea of ILDM with its architecture and APIs. Section 3 describes two basic algorithms of cooperation using ILDM, while the implementations of them are explained in Sect. 4. In Sect. 5, we introduce a benchmark system for MQTT brokers. Section 6 discusses the results of experiments to confirm the usefulness of ILDM. Section 7 explains related works. Finally, we summarize and conclude this paper in Sect. 8.

2. MQTT and ILDM

This section firstly describes MQTT protocol, and subsequently presents the architecture of ILDM.

2.1 MQTT Protocol

MQTT [8] is a protocol of topic-based pub/sub, standardized by OASIS. It is known for lightweight design such as a minimum of two bytes header size. As we stated before, a broker manages topics and mediates between clients. Below is an example flow of using MQTT.

1. A client *X* sends CONNECT message to a broker to establish a connection.
2. *X* sends SUBSCRIBE message to the broker. It informs the topics of interest of *X* to the broker.
3. Another client sends PUBLISH message to the broker, with specifying a topic. If the topic is included in the above topics of interest, this message is forwarded to *X* by the broker.
4. *X* sends DISCONNECT message to the broker, to terminate the connection.

MQTT provides several useful functions for clients, such as "QoS", "Retain" and "Will".

QoS provides capability of configuring the level of delivery confirmation. A client and a broker try to confirm the delivery of a PUBLISH message and resend it if needed, according to the QoS level. Three levels are defined: "At most once delivery", "at least once delivery", and "exactly once delivery".

Retain is for delivering a latest message in the past to a new subscriber. A PUBLISH message has a flag of Retain.

[†]An appropriate product may be determined not only by technical reasons but also by non-technical reasons, e.g., business relationships.

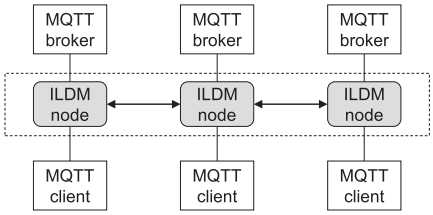


Fig. 3 Interworking Layer of Distributed MQTT brokers.

If the flag is set to *true*, a broker stores the message until a new PUBLISH message whose Retain flag is *true* of the same topic arrives. This stored message will be forwarded to new subscribers of the topic.

Will enables to inform unexpected close of a connection. CONNECT message has a flag of Will. If the Will Flag is set to *true*, a broker stores a Will message and Will topic which are also included in the CONNECT message. The Will message will be published from the broker, when it detects the connection with the client which has sent the CONNECT message is unexpectedly closed.

2.2 Overview of ILDM

In this paper, we propose Interworking Layer of Distributed MQTT brokers (ILDM). ILDM-based cooperation is composed by multiple brokers and ILDM nodes. An ILDM node is arranged between a broker and clients as shown in Fig. 3. As well as relaying MQTT clients and a broker as if it were a proxy, an ILDM node can connect with other ILDM nodes so that multiple and arbitrary kinds of brokers can communicate with each other via ILDM nodes.

Regarding an ILDM node, we assume the following notations: **local client** denotes a client directly connecting with the ILDM node, **local broker** denotes a broker directly connecting with the ILDM node, **remote ILDM node** denotes one of the other ILDM nodes included in the whole cluster, **neighbor ILDM node** denotes one of the remote ILDM nodes directly connecting with the ILDM node, **remote client** denotes a client connecting with a remote ILDM node, **remote broker** denotes a broker connecting with a remote ILDM node.

As there can be a variety of cooperation algorithms, an ILDM node provides APIs which facilitate rapid implementation. Figure 4 illustrates the components of an ILDM node.

We abstracted commonly used functions as five components: session manager, message listener, event listener, status listener, and message generator. These components have programming interfaces so that various algorithms can be easily implemented. Details of practical algorithms we also propose as "PF" and "SF" are discussed later, in Sect. 3.

The architecture of ILDM can be regarded as a microservice in the Service-Oriented Architecture (SOA). Namely, ILDM provides a function block of cooperation with other brokers, and enables flexible development and

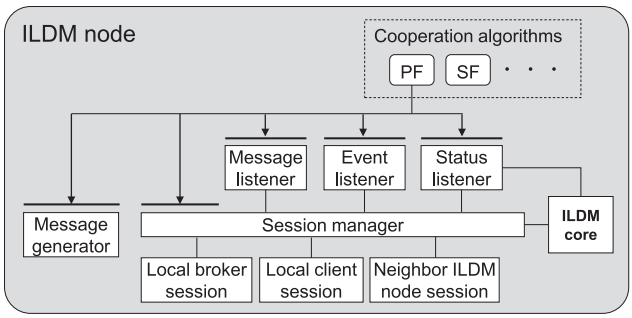


Fig. 4 Components of ILDM node.

operation in the long-term by enhancing the modularity[†]. Note that SOA is somewhat disadvantageous in the aspect of performance in general. In Sect. 6, we clarify the overhead of utilizing ILDM.

From such a viewpoint, ILDM is useful not only for improving the performance such as throughput but also for comparing cooperation algorithms. Unlike developing a cooperable broker from scratch, ILDM-based implementations do not have differences in quality of source codes of functions irrelevant to cooperation. By utilizing this characteristic, we can fairly compare cooperation algorithms.

Note that the design of ILDM assumes heterogeneous environments and organizations/individuals responsible for them as described in Sect. 1. Although utilizing existing protocol-independent scaling techniques such as virtualization can be useful if brokers are placed in uniform environments, e.g., data centers, it is rather difficult to use such techniques in the assuming situation. Especially, some of the broker products themselves have characteristics which make it difficult to apply the existing scaling techniques. For example, Mosquitto [5] is single-threaded, so that increasing the number of cores is not effective. One of the lineups of MessageSight [9] is as an appliance, so that it cannot use software techniques for general purpose servers.

Major APIs provided by the components are listed in Table 1. Details of these APIs are described in the following sections.

2.2.1 Session Manager

Session manager manages communication sessions connecting with a local broker, local clients and neighbor ILDM nodes.

createSession is provided for creating a new session with a neighbor ILDM node or a local broker[††]. The type

[†]The ability to utilize an existing broker product as it is is advantageous; we can leave the maintenance, e.g., updating, and operation to the vendor or the community of the product, and focus on ILDM.

[††]Note that creation of a session between an ILDM node and its local client is triggered by the client, involving the automatic generation of a corresponding session with its local broker inside the ILDM node. **createSession** is mainly used for creating a session with a neighbor ILDM node, or, with a local broker such as to forward MQTT messages from neighbor ILDM nodes.

Table 1 Major APIs of ILDM node.

Component	API

Session manager	`SessionInfo createSession(Address ip, int port)`
	`void closeSession(SessionInfo session)`
	`boolean sendMessage(SessionInfo session, byte[] message)`
	`SessionInfo getPairedSession(SessionInfo session)`

Message listener	`void mqttMessageArrived(SessionInfo session, MsgType type, byte[] message)`
Event listener	`void onTcpDown(SessionInfo session)` `void onRecvFailure(SessionInfo session, Exception exception)`

Status listener	`void onSessionEstablished(SessionInfo session)`
	`void onSessionClosed(SessionInfo session)`
	`void onIldmStop()`
	`void onIldmAdd(Address ip, int port)`
	`void onIldmRemove(Address ip, int port)`

Message generator	`byte[] createMqttMessage(MsgType type, MqttParam parameters)`

SessionInfo indicates meta-information regarding a session, including a session identifier, session status, and input/output streams. Session identifiers can be used for distinguishing the type of each session, e.g., a local client session[†] or a neighbor ILDM node session[††], by storing them into an associative array or other data structure as needed. **closeSession** is used to close an existing session. **sendMessage** sends out a message to a specified session. It returns a boolean value indicating that the transmission is succeeded or not. The **message** argument is a byte array, e.g., an MQTT message. **getPairedSession** is for obtaining a session making a pair with a specified session. When an ILDM node receives a TCP connection request on the listening port for local clients, it automatically creates a session with the local broker. These two sessions make a pair.

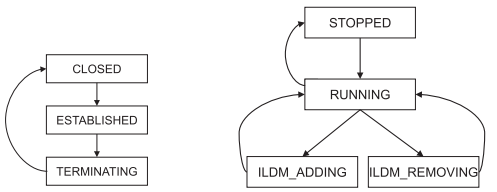
2.2.2 Message Listener

Message listener has an asynchronous callback API: **mqttMessageArrived**. It is called when an ILDM node receives an MQTT message. The type **MsgType** indicates the type of MQTT messages, e.g., CONNECT, CONNACK, and PUBLISH. This callback API is typically a start point of the algorithm-specific processes. If *session* is a local client's session, the ILDM node can relay the MQTT message to the local broker by using **getPairedSession** and **sendMessage**, and then behave according to a cooperation algorithm by using the copy of the message.

Since this API enables to execute arbitrary processes when an ILDM node receives an MQTT message, it is useful for not only cooperation, but also whatever intermediate processing, e.g., validation of data format.

[†]A session identifier of a new local client session can be obtained when **mqttMessageArrived** is called with a CONNECT message. Afterward, the session identifier of a corresponding local broker session can also be obtained by using **getPairedSession**.

[††]A session identifier of a new ILDM node session can also be obtained when **mqttMessageArrived** is called with a CONNECT message. As described later in Sect. 4.1, a CONNECT message from a neighbor ILDM node has a client-ID including a specific character string which can be used for distinguishing from a CONNECT message from a local client.



(a) Statuses of session. (b) Statuses of ILDM node.

Fig. 5 Status transitions.

2.2.3 Event Listener

Event listener has asynchronous callback APIs. **onTcpDown** is called when an ILDM node detects unexpected termination of a TCP connection, while **onRecvFailure** is called when an ILDM node failed to receive a message on a TCP connection.

2.2.4 Status Listener

Status listener enables to insert arbitrary processes in the middle of status transitions.

We define three statuses regarding sessions as shown in Fig. 5 (a). When an ILDM node established a TCP connection, the status of the session changes from **CLOSED** to **ESTABLISHED**. When terminating the TCP connection begins, the status changes to **TERMINATING**. After finishing the termination process, it changes to **CLOSED**.

We also define four statuses regarding the process of an ILDM node, as shown in Fig. 5 (b). When an ILDM node is started, the status changes from **STOPPED** to **RUNNING**. When the ILDM node begins to add or remove a neighbor ILDM node, it changes to **ILDM_ADDING** or **ILDM_REMOVING**. After finishing the process to add or remove, it changes back to **RUNNING**. Note that adding/removing a neighbor ILDM node can occur when an ILDM node starts to run and reads a configuration file or when an operator executes built-in commands at run time, as described later in Sect. 4.4.

Status listener has following synchronous callback APIs: **onSessionEstablished** is called after the status of

the session specified as the argument changes to **ESTABLISHED**. Similarly, **onSessionClosed** is called after the status changes to **CLOSED**. **onIldmStop** is called when the status of an ILDM node changes to **CLOSED**, i.e., just before the ILDM node is terminated. **onIldmAdd** is called when the status of an ILDM node changes to **ILDM_ADDING**. The process of adding a neighbor ILDM node will be coded in this callback. **onIldmRemove** is used in the same manner for removing.

2.2.5 Message Generator

Message generator is a utility component. **createMqttMessage** returns an MQTT message as a byte array. The type **MqttParam** indicates the parameters of MQTT messages, e.g., client identifier and keep-alive interval.

2.3 Authentication and Authorization

Although MQTT 3.1.1 specification [6] does not provide detailed protocols for authentication and authorization besides providing several simple fields such as a user name and a password, these security functionalities are crucial in practical use. Accordingly, we discuss the security aspects of ILDM based on current implementation stated in Sect. 4. In the following, we assume that each ILDM node makes a pair with an MQTT broker and is placed in a local area network (LAN) together with the broker. MQTT clients are placed inside or outside the LAN and connects to the ILDM node.

Some of the security measures are available in the use of ILDM: for one thing, we can use connection encryption between a client and an ILDM node such as by inserting existing TLS termination proxy, e.g., nginx [10]. For another, we can use security-related functions of a broker with regard to the relationship between the broker and its local client, since the intermediated ILDM node creates an MQTT session for each session created by a client and relays MQTT messages as they are. For example, user authentication by using the fields of a user name and a password in a CONNECT message is available.

On the other hand, several restrictions also exist: it is hard to use connection encryption between a client and its local broker in an end-to-end manner. This is because an ILDM node requires to read each MQTT message to cooperate with other ILDM nodes. Although connection encryption between an ILDM node and its local broker is considered less necessary due to the above assumption of placement of them, the inability of utilizing encryption function of an MQTT broker is one of the limitations of current ILDM. In addition to the end-to-end encryption, managing authorization over multiple brokers also has difficulty. Let's consider a simple ACL (Access Control List) based authorization like implemented in Mosquitto, where permissions such as read (subscribe) and write (publish) are granted to specified user names or client IDs. In regard to read-permission, it works; every message received by a client has always gone through its local broker, and thereby only granted clients can read messages as long as the ACL on each broker is appropriately configured. However, in regard to write-permission, it does not work; prohibited PUBLISH messages can be propagated via ILDM. Since an ILDM node does not have information about remote clients, e.g., user names, connected to its neighbor ILDM node, a broker cannot allow/deny a message from its "remote" client based on its user name or client ID. This is another limitation of current ILDM.

There are some possibilities of addressing limitations by future development. For example, we can use connection encryption between a client and an ILDM node without an additional TLS termination proxy by implementing the feature in ILDM itself. Adding the authorization functionality to an ILDM node also can address the difficulty of handling authorization over multiple brokers. As Ramachandran et al. have proposed [11], utilizing blockchain techniques is one of the ways to achieve consistent management over multiple brokers.

3. Cooperation Algorithms

In this section, we propose two basic cooperation algorithms: Publication Flooding (PF) and Subscription Flooding (SF). These algorithms suppose ILDM nodes are connected in a tree structure which does not include closed paths.

3.1 PF-Based Cooperation

PF is a method to share each PUBLISH message among all brokers via ILDM nodes. Each ILDM node relays MQTT messages received from a local client to its local broker. Regarding a PUBLISH message, an ILDM node does not only relay, but also transfers to its neighbor ILDM nodes. ILDM nodes, which receive the transferred PUBLISH message, transfer it to their neighbors recursively, as well as send it to their own local broker. Eventually, all connected brokers receive the PUBLISH message and forward it to their local clients subscribing to the corresponding topic.

Figure 6 shows an example. There are five sets of a broker and an ILDM node: B_1 and I_1 to B_5 and I_5. There are also three clients: C_1 to C_3. We consider the following three steps.

1. Step 1: C_1 subscribes to a topic t.
2. Step 2: C_2 subscribes to the topic t.
3. Step 3: C_3 publishes to the topic t.

Dotted arrows represent the flow of SUBSCRIBE messages, while solid arrows are of PUBLISH messages.

When I_2 and I_3 receive a SUBSCRIBE message from C_1 and C_2, they just relay it to their local brokers. As well as being relayed alike, a PUBLISH message from C_3 is transferred by I_5 to I_3, and spread to all ILDM nodes in a chain reaction.

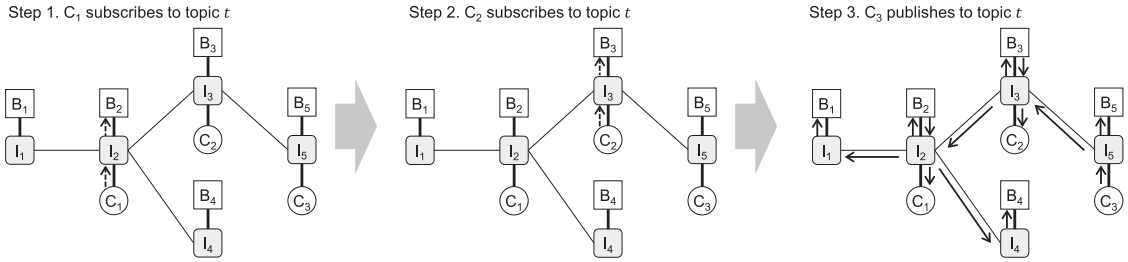


Fig. 6 Example of PF-based cooperation.

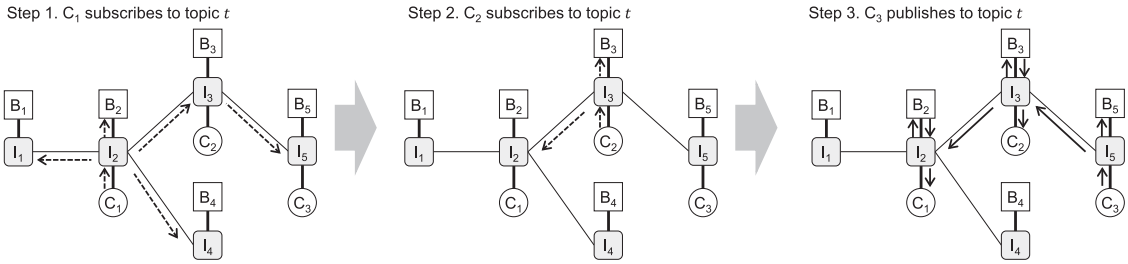


Fig. 7 Example of SF-based cooperation.

3.2 SF-Based Cooperation

Unlike PF, the basic idea of SF is to share subscription information among ILDM nodes. When an ILDM node receives a SUBSCRIBE message, it informs the subscription information such as topic names and QoS levels to its neighbor ILDM nodes, as well as relays the message to its local broker. We call this operation between two ILDM nodes "inter-subscribe", because it looks like the subscribe operation of the MQTT protocol. For example, when an ILDM node X informs the information to an ILDM node Y, it means that "ILDM node X inter-subscribes to topics on ILDM node Y".

When an ILDM node is about to inter-subscribe on a neighbor ILDM node, it checks overlapping with existing subscriptions. If the new subscription information is completely contained in the existing, it will not inter-subscribe redundantly. That is, inter-subscribe operations between ILDM nodes are to share only the difference from existing subscriptions.

Regarding a PUBLISH message, as well as relaying to the local broker, an ILDM node transfers it to neighbor ILDM nodes which have inter-subscribed to the topic of the message. ILDM nodes which receive the transferred PUBLISH message send it to their own local broker. They further transfer the message to their neighbor ILDM nodes in the same manner. Eventually, all brokers which have local clients subscribing to the topic receive the PUBLISH message and forward it to corresponding subscribers.

Figure 7 shows an example. The topology and the scenario are the same as Figure 6. When I_2 receives a SUBSCRIBE message from C_1, it does not only relay the message to its local broker, but also inter-subscribes on I_1, I_3 and I_4. I_3 further inter-subscribes on I_5. In the next step, I_3 receives a SUBSCRIBE message from C_2 and subsequently inter-subscribes on I_2. I_3 does not inter-subscribe on I_5, because I_3 has already inter-subscribed in the first step. Similarly, I_2 does not inter-subscribe on I_1 and I_4. A PUBLISH message from C_3 is transferred by I_5 to I_3, because I_3 has inter-subscribed to the topic t on I_5. I_3 also transfers the message to I_2, and finally C_1 and C_2 receive the message.

3.2.1 Procedure to Inter-Unsubscribe

In regard to inter-unsubscribing, we assume the following procedure:

1. When an ILDM node receives a request of unsubscribing to a topic t from a local client or inter-unsubscribing to t from a neighbor ILDM node, it processes the request.
2. After that, it inter-unsubscribes regarding all of the inter-subscriptions except for those meeting one of the following two conditions:
   - If there is a local client subscribing to t on the ILDM node, it must keep inter-subscription to t on its all neighbor ILDM nodes.
   - If there is a neighbor ILDM node X inter-subscribing to t on the ILDM node, it must keep inter-subscription to t on its all neighbor ILDM nodes except for X.

By these, ILDM nodes can keep an appropriate spanning tree of subscriber nodes without exchanging global information about the topology.

3.3 Comparison of PF and SF

In PF method, each broker receives all PUBLISH messages

even if it has no corresponding subscribers. This means that the total number of ingress messages on each broker is basically the same as the case of a single broker. Therefore, the effect of load distribution mainly depends on a dispersion condition of subscribers. The more scattered the subscribers are, the more effective this method is.

In SF method, a PUBLISH message is delivered to brokers which have subscribers of the same topic as the PUBLISH message. Brokers, which do not have such subscribers and are not on the paths of delivering the message, do not receive it. Hence, this method is effective when publishers and subscribers of a same topic are convergently placed on a small sub-tree.

Note that these methods are, so to speak, a kind of application-level multicast. In this context, utilizing network-level multicast protocols such as PIM (Protocol-Independent Multicast) is one of the means of communication among brokers. However, such protocols have difficulty in handling inter-domain routing. Assuming the exchange of IoT data among various fields over a wide area as described in Sect. 1, application-level multicast like PF/SF methods are considered suitable.

4. Implementation of Cooperation Algorithms

We implemented an ILDM node and PF/SF methods in Java, based on the MQTT version 3.1.1 specification. We diverted the message format of MQTT to the communication between ILDM nodes, because of its lightness. That is, adjacent ILDM nodes establish TCP connections and exchange MQTT messages, e.g., SUBSCRIBE message for inter-subscribing.

In this section, we describe how the cooperation algorithms are implemented.

4.1 Communication between ILDM Nodes

As described above, adjacent ILDM nodes exchange MQTT messages: using a PUBLISH message for transferring a PUBLISH message, while using a SUBSCRIBE message for inter-subscribing. They also use PINGREQ/PINGRESP messages to confirm connections. To distinguish connections with one neighbor ILDM node from with others, adjacent ILDM nodes exchange their identifiers by using the client-ID field of a CONNECT message. This client-ID has a specific character string so that it can also be used to distinguish the session type from the local client sessions. Regarding other parameters in a CONNECT message, protocol name and protocol level are used to validate whether the new connection is of supporting version or not. The keep-alive interval field is used to check the timeout of PINGREQ/PINGRESP messages. Other parameters are not used in the current implementation; it focuses on exchanging minimum essential information by using the MQTT message format, to clarify the feasibility of ILDM. Although utilizing some parameters like user name and password can improve the security of connections between ILDM nodes, such betterment from the practical viewpoint is a part of future works.

Characteristically, we implemented a multi-session mechanism to improve entire throughput. An ILDM node can have multiple TCP connections for each neighbor ILDM node. Before the ILDM node transfers a PUBLISH message, it selects one connection to be used in a round-robin fashion. The number of connections per one neighbor ILDM node can be set by a configuration file.

An ILDM node creates a session with its local broker for each connection with neighbor ILDM nodes. The session is used for forwarding the transferred PUBLISH messages to the broker.

4.2 Relaying between Clients and Broker

An ILDM node relays MQTT messages from its local clients to its local broker, including messages of retransmission caused by QoS control. PINGREQ/PINGRESP messages are also relayed so that clients and the broker can confirm connections. Note that timeout of PINGREQ/PINGRESP messages causes disconnection by a client or a broker. An ILDM node detects the disconnection by using **onSessionClosed** API, and then finds the paired session by using **getPairedSession** API and disconnects the session if it exists.

In regard to a CONNECT message, it has some parameters. As described above, client-ID is used to distinguish sessions inside an ILDM node. An ILDM node also uses parameters of protocol name and protocol level to validate whether the new connection is of supporting version or not. Parameters related to "will" and clean-session are also used by an ILDM node as needed as described later in Sects. 4.5.3 and 4.5.4. Other parameters including authorization and authentication-related ones, e.g., user name, are simply relayed to the local broker.

An ILDM node processes relaying in parallel to improve performance, except for SUBSCRIBE/UNSUBSCRIBE messages and their acknowledgement messages. Regarding those messages, relaying is processed serially because switching the order can cause unexpected status. For example, if a client subscribes to a topic and then unsubscribes to that topic, reversing the order causes the topic still being subscribed after unsubscribing.

4.3 Transferring between Adjacent ILDM Nodes

When using PF method, an ILDM node transfers a PUBLISH message received from a local client to adjacent ILDM nodes, asynchronously with relaying to the local broker. Like MQTT protocol, QoS control for transferring a PUBLISH message between adjacent ILDM nodes is available. The QoS level is set in a configuration file statically.

In case of SF method, an ILDM node receiving a SUBSCRIBE message inter-subscribes when it has not inter-subscribed to corresponding topics yet, after it has successfully finished relaying a SUBACK message to the local client. When the ILDM node receives a PUBLISH message,

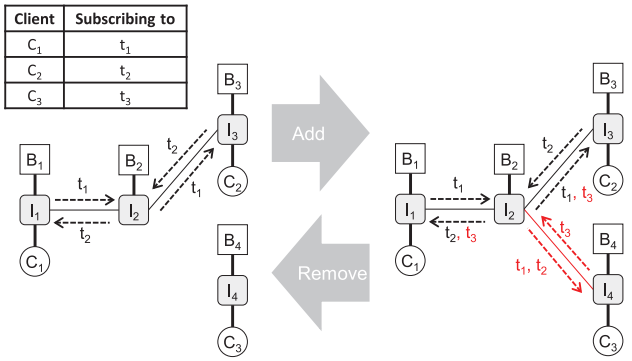


Fig. 8 Add/remove neighbor ILDM nodes in SF method.

it transfers the message in the same manner as in PF method if the topic of the message is inter-subscribed by adjacent ILDM nodes. If it receives a PUBLISH message with the retain flag on, it transfers the message to adjacent ILDM nodes even though the topic is not inter-subscribed. This makes sure each broker can send out an appropriate retained message when a new subscriber of the corresponding topic comes.

4.4 Adding/Removing Neighbor ILDM Nodes

Our implementation has a configuration file and some built-in commands which can be executed at run time. By these, we can specify neighbor ILDM nodes to add or remove not only statically but also dynamically. The process of adding or removing can be implemented by using *onIldmAdd* or *onIldmRemove* callbacks described in Sect. 2.2.4.

In case of PF method, the only thing an ILDM node has to do is creating/closing sessions when adding/removing neighbor ILDM nodes.

When using SF method, an ILDM node additionally needs to synchronize the status of inter-subscribing. For adding, the ILDM node calculates the set of topics to which it has already inter-subscribed. Subsequently, it inter-subscribes to all the topics of the set on the new neighbor ILDM node. If the ILDM node does not have existing neighbor ILDM nodes, the set of topics to which its local clients have subscribed is used instead. The new neighbor ILDM node also inter-subscribes on the ILDM node in the same manner. Afterward, both ILDM nodes newly paired proceed the normal process of being inter-subscribed. For removing, ILDM nodes, which are about to disconnect, inter-unsubscribe against each other.

Figure 8 shows an example. I_2 and I_4 are initially not connected. When I_2 and I_4 add each other as a new neighbor, I_2 inter-subscribes to the topic t_1 and t_2 on I_4, because I_1 and I_3 have inter-subscribed to these topics. I_4 inter-subscribes to the topic t_3 to which C_3 has subscribed. I_2 subsequently inter-subscribes on I_1 and I_3 by adding the topic t_3.

Note that current ILDM does not assume concurrent addition/removal of multiple ILDM nodes. Especially in SF method, concurrent addition/removal may cause an inconsistent situation such as nonexistence of an inter-subscription which should be.

Regarding the failure of an ILDM node, its neighbor ILDM nodes can detect it by using the **onTcpDown** API. In that case, each of them carries out the removal procedure. If the failed ILDM node comes back, and even if it causes re-connection of divided sub-trees, the previous status can be restored by simply adding it as a new neighbor. This is because ILDM nodes have no status which should be synchronized in PF method and can keep an appropriate spanning tree of subscriber nodes without exchanging global information of the topology in SF method.

4.5 Characteristic Functions of MQTT

As we described in Sect. 2, MQTT has some specific functions such as QoS, Will, Retain, and Clean-session. Both implementations of PF and SF enable MQTT clients to use these functions transparently over multiple brokers.

Unless otherwise specified, the descriptions in the following sections are common to PF and SF.

4.5.1 QoS

Both in PF and SF, an ILDM node relays QoS-related messages such as PUBACK so that QoS level configuration is available between a local broker and local clients. Further, we can apply the idea of QoS control to transferring a PUBLISH message between adjacent ILDM nodes. This enables distributed brokers to adjust a trade-off of reliability and performance.

Note that current ILDM does not provide QoS functionality between a client and its remote broker/brokers in an end-to-end manner. Hence, a PUBLISH message with QoS level 1 or 2 may not be delivered to some remote brokers whereas it is assured that its local broker receives it. Such a possibility of the inconsistent situation is one of the limitations of current ILDM. As a possibility by future development, we can consider implementing consistent QoS management into ILDM. For example, when an ILDM node receives a PUBLISH message with QoS level 1, consistent management can be achieved by sending back a PUBACK message to the source client after confirming all brokers receive the PUBLISH message at least once. Since such a mechanism is considered to affect the performance of ILDM significantly, considering suitable strategies is one of the important future works.

4.5.2 Retain

PF method can provide Retain function without adding special processes, because each broker receives all PUBLISH messages and stores them if they have retain-flag being set to *true*. In case of SF method, when an ILDM node receives a PUBLISH message with retain-flag set to *true*, it needs to transfer the message to adjacent ILDM nodes even though

the topic is not inter-subscribed. This makes sure each broker can send out an appropriate retained message when a new subscriber of the corresponding topic comes.

4.5.3 Will

According to the MQTT specification of version 3.1.1, PUBLISH messages from a broker do not have any information to know whether they are will-messages or not. However, an ILDM node is desired to be able to transfer will-messages to its neighbor ILDM nodes as necessary. To do this, we implemented the ILDM node so as to store a will-message and a will-topic internally when it receives a CONNECT message.

When an ILDM node detects the unexpected closing of a network connection with a local client or the local broker, and if the will flag of the connection is set to be *true*, it sends out the corresponding will-message to its neighbor ILDM nodes. The ILDM node needs not to send the will-message to local clients, because the local broker sends it. In case of SF method, sending will-messages to neighbor ILDM nodes is executed only if the will-topics are inter-subscribed.

4.5.4 Clean-Session

When using PF method, there is no specific process regarding the clean-session.

In case of SF method, an ILDM node keeps the status of inter-subscribing related to a local client whose connection has the clean-session flag set to be $false$, even if the connection is unexpectedly closed.

5. Benchmark System

To verify the effects of ILDM, we designed and constructed a benchmark system which can be applied for both a single broker and multiple brokers.

5.1 Performance Indexes

We use the following four performance indexes.

Ingress throughput
: The number of messages which brokers receive from publishers per unit time.

Egress throughput
: The number of messages which brokers send out to subscribers per unit time.

Latency
: Required time since a publisher sends a message until a subscriber receives it.

Loss rate
: The ratio of the number of missed messages to the number of messages which subscribers should receive.

Figure 9 shows the components of the benchmark system. Multiple publishers and subscribers are ran for measuring throughput and loss rate. We denote these clients by

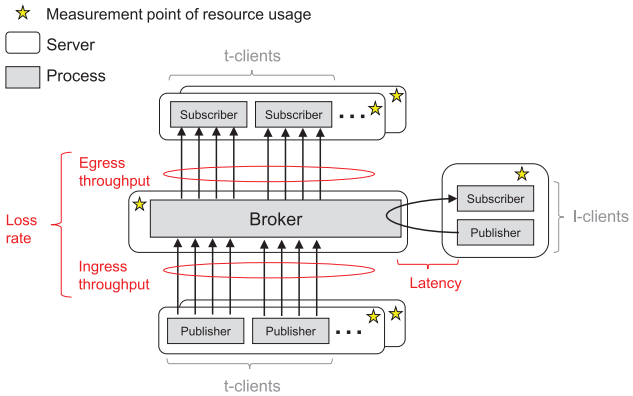


Fig. 9 Components of benchmark system.

t-client. Another pair of a publisher and a subscriber is also placed on a server different from those for t-clients. We denote these clients by **l-client**. They specify same topic so that latency can be acquired by calculating the turnaround time. In parallel, resource usage on each server is recorded, e.g., CPU usage.

5.2 Load Testing Tool

We developed a load testing tool, which is operated as t-clients and l-clients. We used the client library of SurgeMQ [12] known for its high performance so that the tool can send/receive PUBLISH messages with high frequency. The tool has the following functions:

- Subscribe to topics according to a pre-defined scenario.
- Send PUBLISH messages to topics at a certain interval during certain period of time, according to a pre-defined scenario. t-clients and l-clients can be configured with different intervals.
- Record logs of sending and receiving PUBLISH messages with timestamps.
- Gather logs recorded on multiple clients after the duration.
- Calculate performance indexes from the logs.

During the measurement period of time, the tool records a timestamp when: (i) it sends a PUBLISH message, and (ii) it receives a PUBLISH message.

By using (i) of t-clients, the tool calculates the ingress throughput for each second. Similarly, the egress throughput for each second is also calculated by using (ii).

For acquiring latency, the tool uses the difference between (i) and (ii) of l-clients. Since the tool sets identifiers of a client and a message to a payload, it can determine the correspondence of timestamps.

Regarding loss rate, the tool firstly calculates the number of PUBLISH messages to be received by t-clients. This can be derived analytically by considering the scenario, i.e., the number of subscribers belonging to each topic, and the actual number of produced messages calculated by (i). Secondly the tool calculates the number of arrived messages

from (ii). By using these, it can calculate the loss rate as the ratio of the number of missed messages.

5.3 Benchmark Method

The aim of benchmark is to find out the limit of performance of brokers. Assuming every t-client acting a publisher sends PUBLISH messages at a same interval, we define the limit of performance as follows.

Definition 1. *If measured throughput satisfies the following restriction, the performance is under the limit.*

$$\frac{\{\text{egress throughput}\}}{\{\text{ingress throughput}\} \times \{\text{sp-ratio}\}} \geq 0.99$$

$$\textit{where} \quad \text{sp-ratio} = \frac{\sum_i P(t_i) \times S(t_i)}{\sum_i P(t_i)},$$

$$P(t_i) = \{\text{number of publishers of } i\text{th topic}\},$$

$$S(t_i) = \{\text{number of subscribers of } i\text{th topic}\}$$

This is based on the idea that if egress throughput is less than ingress throughput multiplied by sp-ratio, the number of pending messages in the brokers is monotonically increasing. In other words, this definition represents the limit of allowable continuous load.

To find the maximum performance satisfying the restriction defined in Definition 1, we introduce a benchmark method. This method tries to find the point of the very limit by varying the interval of PUBLISH messages. It is conducted along with the following steps.

Step 1: Conduct measurement repeatedly with doubling the interval of sending a PUBLISH message. For example: 1ms, 2ms, 4ms, ...

Step 2: From the results of Step 1, find the minimum interval satisfying the restriction in Definition 1.

Step 3: Divide the segment between the minimum interval and the smaller interval next to the minimum interval into 20. For example, if the minimum interval is 4ms, we divide the segment between 4ms and 2ms like as: 2ms, 2.1ms, 2.2ms, ..., 3.9ms, 4ms.

Step 4: Conduct measurement for each interval calculated in Step 3[†].

Step 5: From the results of Step 4, find the minimum interval satisfying the restriction in Definition 1.

Finally, the result by using the minimum interval clarified in Step 5 indicates the limit of performance.

This result is ensured that the error ratio is not more than five percent. In other words, the throughput resulted by using the smaller interval next to the minimum interval is at most 1.05 times larger than using the minimum interval. We can prove it as follows:

Proof. We assume that x is the minimum interval in Step 2. Therefore $x/2$ is the smaller interval next to the minimum interval. Here the stepping width calculated in Step 3 is $(x - x/2)/20 = x/40$. We denote the width as y. The maximum error ratio is at least $x/(x-y)-1$ and at most $(x/2+y)/(x/2)-1$. Consequently, the highest error ratio is 0.05. □

To evaluate the performance appropriately, it is also important whether there is a bottleneck caused by things other than the performance of brokers. We considered the following viewpoints:

- TCP flow control caused by overloaded receiving on subscribers.
- Ethernet flow control caused by lacking bandwidth of subscribers' side.
- TCP retransmission caused by packet loss on the network.

In our benchmark system, we monitored above matters by checking the window size in TCP ACK frames, occurrence of PAUSE frames, and retransmission logs.

6. Evaluation

We conducted some experiments by using the benchmark system described in Sect. 5. The aim of the experiments is to confirm the usefulness of ILDM; namely, clarify whether it can improve the throughput and how it affects the latency and the loss rate, compared to using a single broker like the cloud-based architecture.

6.1 Experimental Settings

In each experiment, we ran the load testing tool for 80 seconds. The performance indexes stated previously were calculated by excluding the first and last 10 seconds, i.e., substantial measurement time was 60 seconds. QoS level was set to 0, and the size of payload of each PUBLISH message was 32 bytes.

The configuration of topics and clients depicted in Fig. 10 are as follows:

- There are five topics for measuring throughput and loss

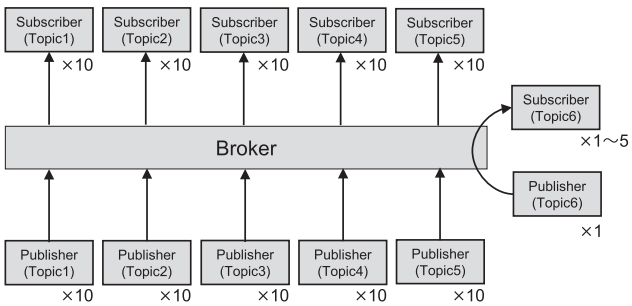
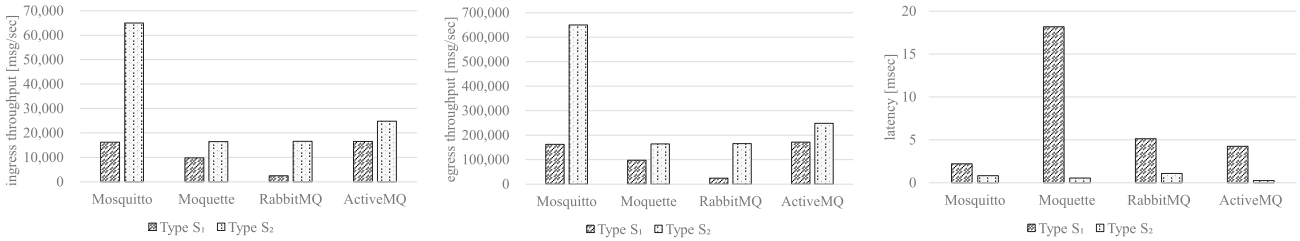


Fig. 10 Configuration of topics and clients.

[†]Although it is considered efficient to use binary search instead of dividing into 20 intervals, the measurement method stated here is rather suitable for automatizing and parallelizing the measurement.



(a) Ingress throughput. (b) Egress throughput. (c) Latency.

Fig. 11 Evaluation of single brokers.

Table 2 Spec of servers.

	Type S_1	Type S_2
Processor	Atom C2750 (8 core, 2.4 GHz)	Xeon E5-2690V3 (12 core, 2.6 GHz) × 2
Memory	16 GB	256 GB
OS	Ubuntu 14.04	Ubuntu 14.04
NW	1 GbE	10 GbE

Table 3 Patterns of measurments.

Pattern	Description
A	Using one broker with one ILDM node.
B	Using 5 brokers with ILDM. t-clients are placed with no locality.
C	Using 5 brokers with ILDM. t-clients are placed with high locality.
D	Using 5 brokers with ILDM. t-clients are placed with low locality.

rate: from "topic1" to "topic5". There also be "topic6" for measuring latency.
- As t-clients, the five topics have 10 publishers and 10 subscribers respectively. Thus, sp-ratio is 10.
- As l-clients, "topic6" has a publisher and up to five subscribers. The publisher sends a PUBLISH message for each one second.

We calculated the average of ingress/egress throughput and latency in the measurement time of 60 seconds, and found the limit of performance by using our benchmark method.

6.2 Hardware Environments

In the evaluation, we use servers described in Table 2 with a non-blocking L2 switch. There are 10 type S_1 servers and one type S_2 server, and we used an appropriate number of servers for each evaluation pattern.

Power saving functions such as EIST (Enhanced Intel SpeedStep Technology) are disabled to avoid unexpected performance control and to clarify the relation between the spec of the servers and the results of measurement.

The time of servers on the benchmark system are synchronized by using an NTP server placed in the same network segment.

6.3 Evaluation of Single Brokers

As preliminary experiments, we evaluated the performance of open-source MQTT brokers alone.

We used the following four products: Mosquitto 1.4.5, Moquette 0.8 [13], RabbitMQ 3.6.0 [14], and ActiveMQ 5.13.3 [15]. Note that Moquette and ActiveMQ are implemented in Java, and we used Oracle Java SE 8. JVM parameters in using these brokers were: using parallel GC, 8 GB of initial and max heap size in an S_1 server, and 64 GB of initial and max heap size in an S_2 server. We measured the performance by changing the types of servers, S_1 and S_2, on which we ran the brokers.

Figure 11 (a) and 11 (b) shows the results of throughput. As the benchmark method indicates, egress throughputs are almost equal to ingress throughputs multiplied by sp-ratio. When using the S_1 server, the performance of ActiveMQ and Mosquitto are the tops. ActiveMQ is slightly larger, but almost even. Regarding the S_2 server, Mosquitto is the largest and its egress throughput reaches over $600,000$. It can be seen that there is a different tendency of performance improvement among brokers by changing the server type, S_1 and S_2. One of the possible reasons is the difference in implementation design; Mosquitto is single-threaded, whereas others are multi-threaded. Since Mosquitto is optimized to run in a single thread and has no overhead of managing threads, it may be more sensitive to the performance of a single core than others.

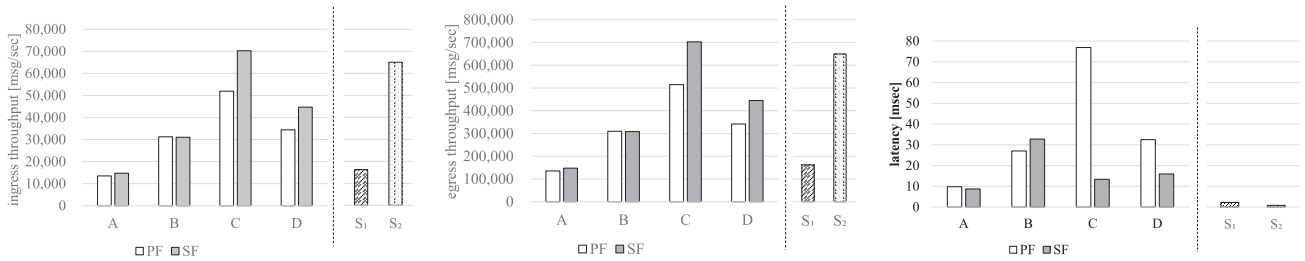
Figure 11 (c) shows the result of latency. As for latency, the shorter the better. In case of the S_1 server, Mosquitto has the shortest latency. On the other hand, using the S_2 server, every broker has approximately less than 1 millisecond latency. ActiveMQ is the best, but the difference is quite small.

In these measurements, the loss rate was zero for all patterns.

6.4 Evaluation of ILDM-Based Cooperation

We evaluated the performance of ILDM-based cooperation. Although the principal feature of ILDM is the capability of connecting heterogeneous brokers, we used one kind of broker to clarify the performance characteristics of ILDM itself. We chose Mosquitto because it indicated relatively better performance among the four brokers in Sect. 6.3.

Table 3 states the patterns of measurements. In these patterns, each pair of a broker and an ILDM node is placed



(a) Ingress throughput. (b) Egress throughput. (c) Latency.

Fig. 12 Evaluation of ILDM-based cooperation.

in an S_1 server.

Pattern B, C, and D use five pairs of a broker and an ILDM node connected in a row. Each ILDM node has the same number of local t-clients, i.e., 20 t-clients. These patterns have a difference of locality of those 100 t-clients. Placement of t-clients on each ILDM node is as follows:

pattern B: Two publishers and two subscribers for every five topics.
pattern C: 10 publishers and 10 subscribers of one topic.
pattern D: Eight publishers and eight subscribers of one topic, one publisher and one subscriber of a different topic, one publisher and one subscriber of another different topic.

In pattern B, C, and D, each of five ILDM nodes has a l-client as a subscriber of topic6. Only one ILDM node placed at the end of the list of the five ILDM nodes has one more l-client as a publisher of topic6. Hence, five data of latency are obtained every second in the measurement time of 60 seconds.

Figure 12 (a) and 12 (b) shows the results of throughput. "PF" and "SF" in the legend denote the cooperation algorithms. The results of single Mosquitto broker are depicted again for comparison, on the right side of the figures. Same as results in Sect. 6.3, egress throughputs are almost equal to ingress throughputs multiplied by sp-ratio.

By comparing pattern A and S_1 of Mosquitto, we can see that the overhead of using an ILDM node is not so large; it was suppressed to approximately 10 percent. Results of pattern B, C, and D indicate that ILDM-based cooperation provides better throughput compared with using a single broker. Especially in pattern C with SF method, the throughput overtook the case of using a single broker on the S_2 server. Since the spec of type S_2 is quite higher than type S_1, this is an impressive result.

It can be said that locality of placing clients affects the performance, by comparing patterns B, C and D. High locality made throughput larger, especially with SF method. This is due to the characteristic of SF method described in Sect. 3.3. Considering edge-heavy data, having high locality, such tendency could be effective.

Figure 12 (c) shows the result of latency. Here also the results of single Mosquitto broker are depicted again for comparison. Basically the patterns using multiple brokers are inferior, because a PUBLISH message is forwarded with multi-hop until it arrives at corresponding subscribers.

Pattern A shows approximately 10 milliseconds. Although this is larger than S_1, the result is considered not to impair the effect of reducing latency in the edge-based architecture, since RTT between IoT devices and data centers could be over 100 milliseconds if it across different countries.

In pattern B, both cases of PF and SF seem to have the same load of throughput. Therefore, the latency of SF method is a little longer due to its complicated processing compared to PF method probably. On the other hand, pattern C and D show that latency of PF method is longer than that of SF method. The reason for this is considered that more redundant messages are propagated in PF method compared to SF method. Pattern C is the case with the highest throughput, so that brokers and ILDM nodes running with PF method tend to be busy and take much time for handling PUBLISH messages.

In these measurements, the loss rate was zero for all patterns.

7. Related Work

Some of existing MQTT products have functions enabling multiple brokers to cooperate with each other. Mosquitto [5] and VerneMQ [16] have a function called "bridge". It enables brokers to forward PUBLISH messages according to predefined rules, e.g., sets of a target broker and topics. This can be used such as for constructing a wireless mesh network [17]. VerneMQ also has a function of "cluster", which makes multiple brokers to share MQTT-related statuses such as subscriptions dynamically and to behave as if they were logically a single broker. Several products have such a clustering feature, e.g., HiveMQ [18], RabbitMQ [14], and EMQ [19]. Each of them has its own approach for sharing the statuses. For example, VerneMQ uses Plumtree [20] for replication of subscriptions and retained messages [21]†, whereas RabbitMQ and EMQ use Mnesia [22] for their data storage. These approaches have pros and cons respectively, e.g., difference in the tolerance

†Detailes can be found in: https://github.com/vernemq/vernemq/issues/83#issuecomment-246173311 (accessed 2019-05-23).

for network partitions. MessageSight [9] can be configured in an HA configuration, i.e., two brokers cooperate so that when the primary-node stops the standby-node can continue to process MQTT messages. Since these cooperation functions are implementation-dependent, only certain and same kinds of brokers can cooperate with each other.

Dynomite [23] makes existing data stores, e.g., Redis and Memcached, into a distributed data store. The aim is to provide high availability and resiliency on storage engines which do not inherently have those functionalities. BondFlow [24] proposes a system enables encapsulated web services to interconnect. These are similar to ILDM from the viewpoint of modularizing functionality of interwork, while the target is different from ILDM.

There are existing methods of topic-based pub/sub with mesh-based topologies [25], [26], unlike that PF and SF assume a tree structure which does not include closed paths. PIQT [27], which is based on PIAX [28], is one of the implementations using such mesh-based methods. We can obtain better characteristics like scalability and reliability by implementing these methods in ILDM.

As far as we know, there are no existing proposals of connecting heterogeneous MQTT brokers and thereby no quantitative evaluation of its performance characteristics.

8. Conclusion

In this paper, we proposed a novel mechanism called ILDM which enables heterogeneous MQTT brokers to cooperate with each other. The APIs provided by ILDM enable to develop a variety of cooperation algorithms easily. Two basic algorithms, PF and SF, and a practical benchmark system for MQTT brokers were also presented.

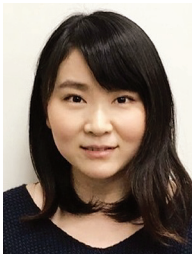
We evaluated the usefulness of ILDM with the benchmark system. By connecting five brokers via ILDM, the throughput increases approximately two to four times than using a Mosquitto broker alone.

Our future work includes evaluating the edge-based architecture in comparison with the cloud-based architecture with actual environments, confirming characteristics of performance with combinations of different kinds of broker products, and developing more scalable cooperation algorithms.

References

[1] 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.
[2] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," ACM Computing Surveys, vol.35, no.2, pp.114–131, 2003.
[3] A. Popov, A. Proletarsky, S. Belov, and A. Sorokin, "Fast prototyping of the internet of things solutions with ibm bluemix," HICSS, pp.1064–1072, 2017.
[4] D. Okanohara, S. Hido, N. Kubota, Y. Unno, and H. Maruyama, "Krill: An architecture for edge heavy data," ASBD, 2013.
[5] Mosquitto. mosquitto.org (accessed 2019-01-07).
[6] OASIS Message Queuing Telemetry Transport (MQTT) TC, "MQTT version 3.1.1 plus errata 01." OASIS, 2015.
[7] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, "Dissemination of edge-heavy data on heterogeneous mqtt brokers," IEEE CloudNet, pp.1–7, 2017.
[8] MQTT. mqtt.org (accessed 2019-01-07).
[9] W.J. Chen, R. Gupta, V. Lampkin, D.M. Robertson, and N. Subrahmanyam, Responsive Mobile User Experience Using MQTT and IBM MessageSight, IMB Corp., 2014.
[10] nginx. nginx.org (accessed 2019-05-22).
[11] G.S. Ramachandran, K.-L. Wright, L. Zheng, P. Navaney, M. Naveed, B. Krishnamachari, and J. Dhaliwal, "Trinity: A byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence," IEEE ICBC, pp.227–235, 2019.
[12] SurgeMQ. github.com/influxdata/surgemq (accessed 2019-01-07).
[13] Moquette. github.com/andsel/moquette (accessed 2019-01-07).
[14] RabbitMQ. www.rabbitmq.com (accessed 2019-01-07).
[15] ActiveMQ. activemq.apache.org (accessed 2019-01-07).
[16] VerneMQ. vernemq.com (accessed 2019-01-07).
[17] A.A.D. Cruz, M.L.A. Parabuac, and N.M.C. Tiglao, "Design and implementation of a low-cost and reliable wireless mesh network for first-response communications," IEEE GHTC, pp.40–46, 2016.
[18] HiveMQ. www.hivemq.com (accessed 2019-01-07).
[19] EMQ. emqtt.io (accessed 2019-05-23).
[20] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," IEEE SRDS, pp.301–310, 2007.
[21] GitHub - VerneMQ. github.com/vernemq/vernemq (accessed 2019-05-23).
[22] H. Mattsson, H. Nilsson, and C. Wikström, "Mnesia — A distributed robust DBMS for telecommunications applications," Practical Aspects of Declarative Languages, vol.1551, pp.152–163, 1998.
[23] Dynomite. github.com/Netflix/dynomite (accessed 2019-01-07).
[24] J. Balasooriya, M. Padhye, S.K. Prasad, and S.B. Navathe, "Bondflow: A system for distributed coordination of workflows over web services," IEEE IPDPS, pp.121a–121a, 2005.
[25] R. Banno, S. Takeuchi, M. Takemoto, T. Kawano, T. Kambayashi, and M. Matsuo, "Designing overlay networks for handling exhaust data in a distributed topic-based pub/sub architecture," Journal of Information Processing, vol.23, no.2, pp.105–116, 2015.
[26] Y. Teranishi, R. Banno, and T. Akiyama, "Scalable and locality-aware distributed topic-based pub/sub messaging for iot," IEEE GLOBECOM, pp.1–7, 2015.
[27] PIQT distributed pub/sub broker. piqt.org (accessed 2019-01-07).
[28] Y. Teranishi, "PIAX: Toward a framework for sensor overlay network," IEEE CCNC, pp.1–5, 2009.

Ryohei Banno received his Bachelor's and Master's degrees from Hokkaido University in 2010 and 2012, then earned his Ph.D. from Tokyo Institute of Technology in 2018. He was a researcher in NTT Network Innovation Laboratories. Since 2018, he has been a researcher in Tokyo Institute of Technology. He received IPSJ Best Paper Award in 2015. His research interests include distributed systems, especially structured overlay networks. He is a member of IEEE, IEEE Computer Society, IPSJ, and IEICE.



Jingyu Sun received her M.E. and Ph.D. degrees from Tokyo University, Japan, in 2013 and 2016. From 2016, she joined NTT and started her research about ubiquitous and Internet of Things platform technologies at Network Innovation Laboratories.

Susumu Takeuchi received his M.E. and Ph.D. degrees from Osaka University, Japan, in 2003 and 2006, respectively. From 2006 to 2009, he was an assistant professor of Graduate School of Information Science and Technology, Osaka University. In 2009, he joined National Institute of Information and Communications Technology (NICT) as an expert researcher. Since he joined NTT in 2011, he researched ubiquitous and Internet of Things platform technologies. He received IPSJ Best Paper Award in 2011 and JIP Specially Selected Paper Award in 2014. He is currently a senior research engineer in Network Innovation Laboratories. He is a member of IEEE and IPSJ.

Kazuyuki Shudo received B.E. in 1996, M.E. in 1998, and a Ph.D. degree in 2001 all in computer science from Waseda University. He worked as a Research Associate at the same university from 1998 to 2001. He later served as a Research Scientist at National Institute of Advanced Industrial Science and Technology. In 2006, he joined Utagoe Inc. as a Director, Chief Technology Officer. Since December 2008, he currently serves as an Associate Professor at Tokyo Institute of Technology. His research interests include distributed computing, programming language systems and information security. He has received the best paper award at SACSIS 2006, Information Processing Society Japan (IPSJ) best paper award in 2006, the Super Creator certification by Japanese Ministry of Economy Trade and Industry (METI) and Information Technology Promotion Agency (IPA) in 2007, IPSJ Yamashita SIG Research Award in 2008, Funai Prize for Science in 2010, The Young Scientists' Prize, The Commendation for Science and Technology by the Minister of Education, Culture, Sports, and Technology in 2012, and IPSJ Nagao Special Researcher Award in 2013. He is a member of IEEE, IEEE Computer Society, IEEE Communications Society and ACM.