Wildcard Topic Management using Bloom Filter in Distributed MQTT Brokers

Ryohei Banno*, Yoshito Watanabe†*
* Graduate School of Social Data Science, Hitotsubashi University, Tokyo, Japan
† Solid Surface Inc., Tokyo, Japan
Email: banno@computer.org, yoshito.watanabe@solidsurface.co.jp

***Abstract*—MQTT is a widely used protocol in IoT systems. Due to the central role of an MQTT broker in handling messages, large-scale systems need to distribute the load across multiple brokers. When using multiple brokers, sharing subscription topics among them is a common strategy to prevent message flooding. Given the potentially large volume of topics, some existing approaches use a Bloom filter to store and manage them with space-saving. However, these methods have difficulties in managing wildcards, i.e., many queries to the Bloom filter could occur to search one topic. In this paper, we introduce a method to handle wildcard topics efficiently using a Bloom filter. The proposed method adds prefixes of subscription topics to the Bloom filter. Searching for a published topic is processed like using Trie structure while considering wildcard patterns. Experimental results demonstrate that our method reduces the number of queries to the Bloom filter compared to existing methods in many cases.**

***Index Terms*—MQTT, Publish-subscribe, Bloom filter, IoT**

I. Introduction

MQTT [1] is one of the major protocols in IoT systems. It follows a publish-subscribe messaging model [2] that provides loose coupling among components. MQTT clients can communicate via an MQTT broker by specifying a topic. A publisher sends a message with a topic to a broker, which forwards the message to subscribers who have subscribed to the topic.

Since an MQTT broker is a concentration of messages, large-scale systems require load distribution by multiple brokers [3]–[5]. Distributed brokers increase the possible number of client connections and improve communication throughput among IoT devices. In using multiple brokers, sharing subscription topics among brokers is a typical approach to avoid flooding PUBLISH messages. Figure 1 shows a simple example. Client C_2 connects to Broker B_2 and subscribes to a topic. By sharing the topic from Broker B_2 to Broker B_1, Broker B_1 can determine whether or not to forward PUBLISH messages from Client C_1 to Broker B_2.

Since the topics shared among brokers could be large, some existing work [6]–[8] utilizes a Bloom filter [9] or its variants. Using a Bloom filter allows for the space-efficient checking of the existence of topics. It significantly reduces communication traffic and memory usage for sharing and managing topics. However, there is a difficulty in handling wildcards. In the MQTT protocol, a subscription topic may have wildcards. Since a Bloom filter uses hash functions and hence assumes exact matching, wildcard topics cause inefficient topic management, i.e., a large number of queries to the Bloom filter. Such inefficiency may impair the performance of brokers, such as throughput and latency.

This work was supported by JST, PRESTO Grant Number JPMJPR21P8, Japan.

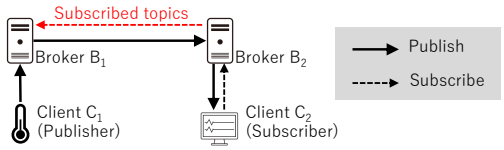

Fig. 1: Example of sharing subscriptions among brokers

In this paper, we propose a novel method to efficiently manage wildcard topics with a Bloom filter. The proposed method manages subscription topics by adding their prefixes to a Bloom filter. Searching for a published topic by the Bloom filter is processed in a manner like Trie [10] with considering wildcard patterns. The proposed method provides a small number of queries to the Bloom filter, especially for diversified subscription topics.

The rest of this paper is organized as follows. Section II describes our assumptions about distributed MQTT brokers. Section III introduces related work on subscription management of MQTT brokers. Section IV explains the proposed method, while Section V presents the experimental evaluation. Finally, we conclude this paper in Section VI.

II. Distributed MQTT brokers

For large-scale systems, load distribution by multiple brokers is essential to improve throughput and latency. In subsequent discussion, we assume there are multiple brokers and every client connects to one of them. The clients can publish and subscribe as if they connect to a single broker. To do so, each broker forwards PUBLISH messages to other brokers as needed. A simple way is forwarding all PUBLISH messages like MQTT-ST [3].

As several studies suggest [4], [5], sharing subscription topics and reducing traffic among brokers is an effective way to improve performance. In this paper, we focus on the data structure to share and manage topics. The following are the assumptions about sharing subscription topics:

TABLE I: Examples of MQTT topics

Subscription topic	Examples of matched published topics
building3/+/temperature	building3/room1/temperature building3/room2/temperature
building3/#	building3/room1 building3/room2/humidity
+/#	building1/room1 building2/room5/temperature

- A broker shares the subscription topics of its clients with other brokers in response to, for example, receiving SUBSCRIBE messages or detecting a new broker.
- When a broker receives a PUBLISH message, it checks if there are corresponding subscriptions in other brokers and determines subsequent actions, such as forwarding the message to the brokers.

Note that these assumptions are general in the existing MQTT load distribution methods. Various topologies and cooperation ways are considerable, e.g., tree and mesh topologies, but discussion on them is outside of the scope of this paper.

Since a set of topics could be large, using a Bloom filter is a typical approach. That is, a broker adds topics to a Bloom filter and shares it with other brokers. Then each broker receiving a PUBLISH message checks the existence of the topic by the Bloom filter. Due to the false positives in a Bloom filter, each broker may need to discard unnecessary messages forwarded from other brokers. We can suppress the false positive rate by appropriately determining the size of the Bloom filter. We omit the details for handling UNSUBSCRIBE messages, but it can be achieved by using some variants of a Bloom filter, like a Counting Bloom filter [11]. Subsequent sections explain the details of MQTT topics and a Bloom filter.

A. MQTT Topics

In MQTT protocol [12], a topic is a slash-separated string of items called topic levels, like "building3/room2/temperature". The maximum length of a topic is $65{,}535$ bytes.

A client can use two kinds of wildcards for subscription topics. One is the single-level wildcard "$+$" that matches any single topic level, and the other is the multi-level wildcard "#" that matches any number of topic levels. The multi-level wildcard appears only at the end of a topic. Examples are shown in Table I.

The number of topics could be enormous in some applications. For example, we can consider using location information in topics by Spatial-ID [13], a universal space identification scheme. Spatial-ID is similar to Slippy map tilenames[1] but extended to 3-dimensional, i.e., space is divided into voxels in a hierarchical manner, and each voxel is assigned an ID. Considering using such voxel IDs as topic levels, there could be a vast number of topics where each topic has tens of levels.

B. Bloom Filter

A Bloom filter [9] is a probabilistic data structure by which we can confirm whether an element exists or not. It is a bit array of length b where initially all bits are set to 0. Adding an element comprises the following steps: calculate hash values by k hash functions, get k positions of the bit array corresponding to the hash values, and set those bits to 1. To search for an element, we calculate k hash values, get k positions, and check the bits. If all corresponding bits are 1, the element exists in high probability. Otherwise, it does not exist. Note that we can easily merge multiple Bloom filters by bitwise logical disjunction.

The search result is possibly a false positive. The false positive rate p_{fp} is calculated as $p_{\mathrm{fp}} = (1 - e^{-(kn_{\mathrm{e}}/b)})^k$ where n_{e} is the number of elements. The optimal value of k that minimizes p_{fp} is $k = (b/n_{\mathrm{e}}) \ln 2$. Assuming the optimal k, the relationship between b and p_{fp} is expressed as $b = -((\ln p_{\mathrm{fp}})/(\ln 2)^2) n_{\mathrm{e}}$.

III. Related work

Since a Bloom filter has superior space and time efficiency, it is often used for topic management. Dominguez et al. [8], [14] utilize a Counting Bloom filter [11], a variant of a Bloom filter, for subscription management in topic-based publish-subscribe messaging, and they conducted experiments with MQTT. Other than MQTT, RabbitMQ, a message-oriented middleware, has a "Stream Filtering" functionality that internally uses Bloom filters for subscription management [15]. Several studies on content-centric networking and named data networking [16], [17] also take a similar approach, using a Bloom filter for message routing. They resemble our proposed method in terms of adding prefixes to a Bloom filter. However, none of these consider efficient handling of the wildcards.

Naaman [6] and Chen et al. [7] introduce a Bloom filter to share subscription topics among multiple MQTT brokers. Their method considers wildcards. In addition to sharing a Bloom filter of subscription topics, each broker shares a set of wildcard topic patterns with other brokers. A wildcard topic pattern is a set of levels where two kinds of wildcards appear. For example, if a broker has subscription topics "a/+/c", "x/+/z", "a/+/+", and "+/+/#", then the set of patterns to be shared is like [{1;-1}, {1,2;-1}, {0,1;2}]. The pattern {0,1;2} means there are one or more subscription topics in which the single-level wildcard appears at levels 0 and 1, and the multi-level wildcard appears at level 2. In the patterns, -1 means there is no corresponding wildcard. Note that we assume the lowest level is 0. When a broker receives a PUBLISH message, it uses the Bloom filter to check whether the topic and its variations obtained by the patterns exist. For instance, if the published topic is "x/y/z" and the shared patterns are the same as the above example, the following variations are searched using the Bloom filter: "x/y/z", "x/+/z", "x/+/+", and "+/+/#". This method assumes the number of wildcard patterns is small. Diversified subscription topics could cause a large number of queries to the Bloom filter.

IV. Proposed method

We propose a method to handle wildcard topics efficiently using a Bloom filter. The proposed method adds the prefixes

[1] https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (accessed June 15, 2024)

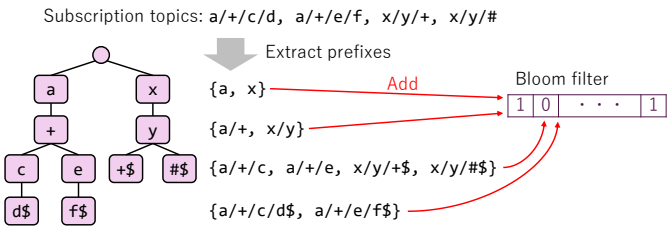

Fig. 2: Adding topic prefixes to Bloom filter

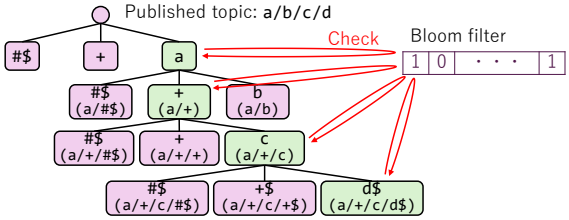

Fig. 3: Searching for topic by Bloom filter

of subscription topics to a Bloom filter. It enables checking the existence of a topic in a manner like searching a topic tree [18], a kind of Trie [10].

As mentioned in Section II, we assume that a broker makes a Bloom filter by its subscriptions and shares it with other brokers. When a broker receives a PUBLISH message, it searches for the published topic by the Bloom filters shared by other brokers to determine subsequent actions, such as forwarding the message to the brokers. The following sections describe how to manage topics with a Bloom filter in the proposed method.

A. Adding topics to Bloom filter

To manage a topic using a Bloom filter, we use its prefixes. Here, a prefix is consecutive topic levels from the beginning to a specific level. For example, considering a topic "foo/bar/baz", there are three prefixes: "foo", "foo/bar", and "foo/bar/baz".

Before adding to the Bloom filter, a terminal symbol is added to the last topic level. Hereafter, $ denotes the terminal symbol. Then, we add each prefix to the Bloom filter. Figure 2 shows an example. A subscription topic "a/+/c/d" involves adding the following prefixes to the Bloom filter: "a", "a/+", "a/+/c", and "a/+/c/d$".

The size of the Bloom filter should be properly determined to suppress false positives. We can calculate the appropriate length of the bit array b from the allowable false positive rate p_{fp} and the assumed maximum number of elements n_{e} by the equation described in Section II-B. We can also determine the optimal number of hash functions, k. To give an example, assuming $p_{\mathrm{fp}} = 0.001$ and $n_{\mathrm{e}} = 100{,}000$, b becomes approximately $1{,}437{,}759$ bits, where k is about 10.

B. Searching for a topic

Searching for a published topic t_p by the Bloom filter is processed like Trie with considering wildcard patterns, as shown in Figure 3. Let d denote the number of topic levels of t_p, and $l_0, l_1, ...l_{d-1}$ denote the topic levels.

We start by searching for the following prefixes whose number of levels is one: "#$", "+", and l_0. These are prefixes of conceivable subscription topics that match t_p. If none of them exists in the Bloom filter, we can conclude that there is no matching subscription. Contrarily, if any of them exist in the Bloom filter, the search continues to the next number of levels. In the next step, the prefixes to search for are obtained by connecting "#$", "+", and l_1 behind the prefix found in the Bloom filter in the previous step. Subsequently, such a process is repeated until the number of levels reaches d. Note that when the number of levels equals d, we add "$" at the end of the prefixes ending with "+" or l_{d-1} before searching. In the process, if any prefix ending with "$" exists in the Bloom filter, the search is terminated at the point, and we can conclude that there is a matching subscription in high probability. Otherwise, the search results in that there is no matching subscription.

Figure 3 shows an example of search process, where we assume using the Bloom filter in Figure 2. Green-colored boxes correspond to the prefixes added in the Bloom filter. In this case, prefixes "a", "a/+", and "a/+/c" exist in the Bloom filter, and finally, "a/+/c/d$" also exists. So, it results in that a matching subscription exists.

In the above search process, we follow the policy below:

- Use the depth-first search.
- Check the prefix ending with "#$" first at each level.

These enable early termination of the search if a matching subscription exists.

Pseudocode is shown in Algorithm 1. The function SEARCH returns **true** if *topic* exists in high probability and **false** if it certainly does not exist. PARSE_TOPIC splits a topic into topic levels and returns a list of them. CONCAT concatenates given strings. SEARCH_HELPER is a function recursively called for the depth-first search with the following arguments: $prefix$ is a prefix existing in the Bloom filter in the previous step. $level$ is the number of levels of prefixes we currently intend to search for. $levelList$ is the list of topic levels of the published topic. From lines 8 to 14, the prefixes to check at $level$ are prepared. From lines 15 to 23, the search is processed recursively by increasing $level$. CHECK_BF is a function that checks if the given element exists in the Bloom filter.

C. Discussion

Since the proposed method can terminate the search if every possible prefix at a level does not exist, fewer queries to the Bloom filter are expected compared to the existing methods that comprehensively search for wildcard patterns. Table II shows the asymptotic analysis. "BF" is a naive method that adds subscription topics to a Bloom filter and searches for all possible wildcard patterns. "BF with patterns" is the existing method [6], [7] described in Section III. Hereafter, we refer to this method as BFP. We use the following notations:

- n_{t}: The number of subscription topics.

Algorithm 1 Search process

```
1: function SEARCH(topic)
2:     levelList ← PARSE_TOPIC(topic)
3:     d ← levelList.LENGTH( )
4:     levelList[d − 1] ← CONCAT(levelList[d − 1], "$")
5:     return SEARCH_HELPER("", 0, levelList)
6: end function

7: function SEARCH_HELPER(prefix, level, levelList)
8:     pList[0] ← CONCAT(prefix, "/#$")
9:     if level = levelList.LENGTH( ) −1 then
10:        pList[1] ← CONCAT(prefix, "/ + $")
11:    else
12:        pList[1] ← CONCAT(prefix, "/ + ")
13:    end if
14:    pList[2] ← CONCAT(prefix, "/", levelList[level])
15:    for i ← 0 to 2 do
16:        if CHECK_BF(pList[i]) = true then
17:            if pList[i].ENDS_WITH("$") = true then
18:                return true
19:            else
20:                return SEARCH_HELPER(
                       pList[i], level + 1, levelList)
21:            end if
22:        end if
23:    end for
24:    return false
25: end function
```

TABLE II: Asymptotic analysis

	BF	BF w/ patterns (BFP)	Proposed
Memory space	$O(n_\mathrm{t})$	$O(n_\mathrm{t} + n_\mathrm{p} n_\mathrm{w})$	$O(n_\mathrm{t} n_\mathrm{lv})$
Search time (average)	Depend on wildcard usage		
Search time (worst)	$O(2^{n_\mathrm{lv}})$	$O(2^{n_\mathrm{lv}})$	$O(2^{n_\mathrm{lv}})$

- n_lv: The average number of levels of topics.
- n_p: The number of wildcard patterns.
- n_w: The average number of wildcards used in a pattern.

Regarding the memory space, the proposed method requires a larger Bloom filter since it adds prefixes instead of topics. The appropriate size of the Bloom filter is proportional to the number of elements, so it becomes $O(n_\mathrm{t} n_\mathrm{lv})$. It is larger than other methods but is not considered to influence significantly because the Bloom filter is quite small compared to storing topics as they are. The average search time strongly depends on the usage of wildcards, so we evaluate it by simulation in Section V. As for the worst case, all methods have exponential orders. However, the worst case is considered to hardly occur in the proposed method, i.e., there is a low probability of searching every branch of the prefix tree embedded in the Bloom filter without termination in the middle of the search. Conversely, the existing methods are relatively easy to face with the worst case; if there is no matching subscription topic, all possible patterns must be searched. We also clarify this tendency in Section V.

V. Evaluation

To clarify the characteristics of the proposed method, we conducted simulation experiments using an implementation in Java. The evaluation criterion is the number of queries to the Bloom filter required to confirm whether a published topic has corresponding subscription topics. The comparison targets are BF and BFP, described in Section IV-C.

TABLE III: Simulation parameters

Parameter	Default value
Number of topics n_t	6,000
Number of topic levels n_lv	6
Wildcard probability p_wc	0.5
Minimum wildcard level l_wc	0

The simulation parameters are shown in Table III. For each parameter, we conducted simulations by changing its value with fixed default values of the other parameters. n_t is the number of subscription topics managed by the Bloom filter. Each topic level constituting a topic is a random string where the length is 10 or a wildcard. n_lv is the number of topic levels each topic has. In the experiments, all topics have the same number of levels for easy analysis. p_wc is the probability of the presence of wildcard topics in n_t topics. Suppose $p_\mathrm{wc} = 0.5$, $n_\mathrm{t}/2$ topics are expected to include wildcards. Note that each wildcard topic is generated as follows. At first, a topic without wildcards is randomly generated. Second, based on it, all possible wildcard topics are generated. Then, one is chosen from them with equal probability. l_wc is the minimum wildcard level. For example, if $l_\mathrm{wc} = 0$, all topic levels could be a wildcard.

We conducted the experiments with the following three patterns for subscription topics:

- Allow to include wildcard-only topics, e.g., "#" and "+/#".
- Exclude wildcard-only topics.
- Exclude wildcard-only topics and matching topics to the published topic.

Wildcard-only topics match a lot of published topics. Especially, "#" matches all topics and could cause large traffic. Hence, assuming a large-scale system, clients may avoid using such wildcard-only topics. In addition, a PUBLISH message does not necessarily have matching subscriptions because of the loose coupling nature of the publish-subscribe messaging model. We use the above three patterns to clarify the difference in these varied situations. The published topic is generated to have n_lv topic levels. Each topic level is a random string with a length of 10.

Figures 4 to 6 show the results. Note that BFP is denoted as "BF w/ patterns" in these figures. We conducted simulations 10 times for each pattern and calculated the average number of required queries to the Bloom filter. Overall, the proposed method achieves a relatively small number of queries. Particularly, the proposed method is significantly superior to the existing methods when the number of levels increases. For $n_\mathrm{lv} = 10$ in Figure 6(b), the proposed method achieves an average of 29.7 queries, less than two percent of BFP.

For most cases in Figures 4(a), 4(b), and 4(c), the average number of queries is 1. This is because there are subscription

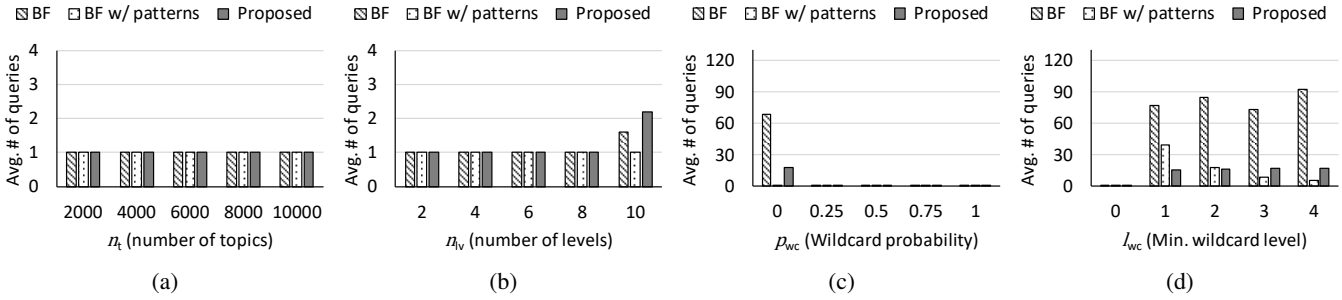

Fig. 4: Average number of queries (with wildcard-only topics)

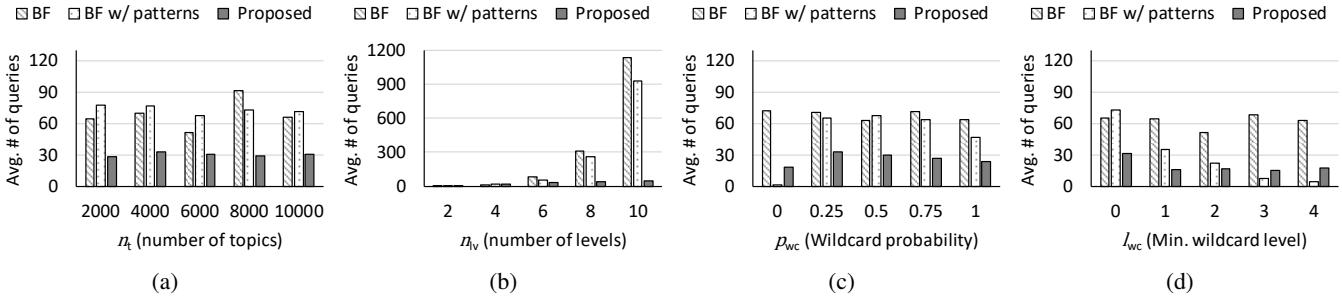

Fig. 5: Average number of queries (without wildcard-only topics)

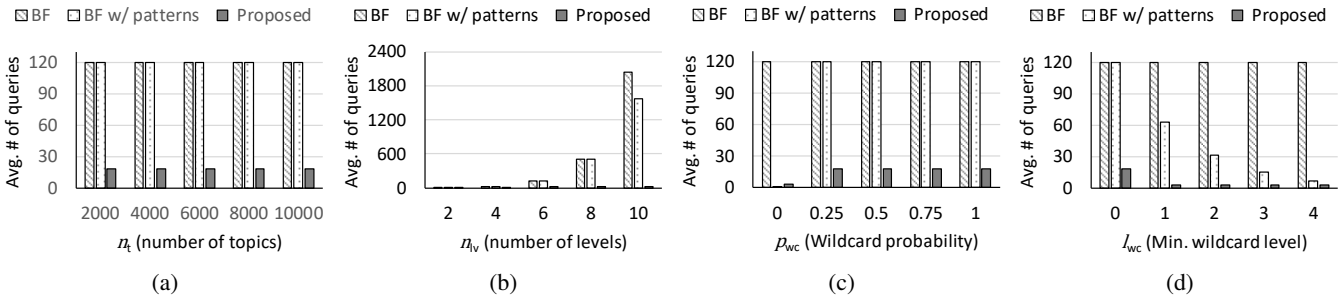

Fig. 6: Average number of queries (without matching subscription)

topics that match most published topics, like "#", in high probability. In every method, such topics are searched first to terminate the search as soon as possible. In Figure 4(c), $p_{\mathrm{wc}} = 0$ induces a relatively large number of queries, because there are no wildcard topics that enable termination in the middle of the search. BFP requires only one query since there is no wildcard patterns. In the cases $l_{\mathrm{wc}} \geq 1$ in Figure 4(d), the topic "#" is excluded, and therefore, the number of queries becomes more than one in every method. For BFP, the larger l_{wc} is, the smaller the average number of queries becomes, due to the decrease of wildcard patterns.

In Figures 5(a), 5(c), and 5(d), BF shows roughly the same number of queries in every case. BF searches for all possible wildcard patterns considered from the published topic. n_{t}, p_{wc}, and l_{wc} do not affect the number of patterns, whereas n_{lv} significantly affects the result of BF as shown in Figure 5(b). BFP is primarily affected by the number of wildcard patterns in subscription topics. Enlarging n_{lv} increases the patterns while increasing l_{wc} involves a decrease in patterns.

Figure 6 shows the cases in which the published topic is not subscribed. In such cases, BF and BFP require a large number of queries because they need to search for considerable patterns comprehensively. Contrarily, the proposed method achieves a smaller number of queries since it can terminate in the middle of the search if prefixes at a level do not exist in the Bloom filter.

Remarkable results are shown in Figures 5(b) and 6(b). In these cases, the proposed method achieves a significantly smaller number of queries than the existing methods. Considering that the search of the Bloom filter occurs for every PUBLISH message, such difference could affect the broker performance. We confirm it in the next section.

A. *Influence on broker performance*

In this section, we explain an experiment to confirm the influence of the number of queries to the Bloom filter on the performance of a broker. The evaluation criteria are the average egress throughput and the average latency. The former

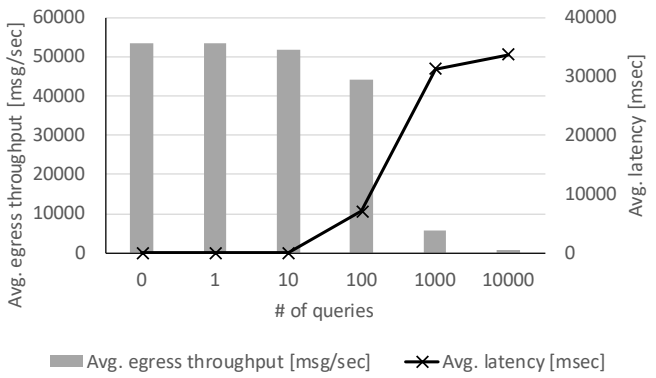

Fig. 7: Affect of searching Bloom filter on broker performance

is the average number of messages sent by the broker to a subscriber per second, and the latter is the average required time from a publisher to a subscriber. We use HiveMQ CE 2024.5[2] for the broker and MQTTLoader v0.8.6[3] for the clients. We modified the broker implementation so that a simplified search process of the Bloom filter is conducted between receiving a PUBLISH message and forwarding it to subscribers. For the Bloom filter implementation, we use guava 33.1.0-jre[4], which internally uses the 128-bit MurmurHash3 hash function. The expected number of elements added to the Bloom filter is set to 10,000, and the desired false positive rate is set to 0.001. Before measuring the throughput and latency, 10,000 elements are added to the Bloom filter. Each element is a random string whose length is 100. When the broker receives a PUBLISH message, processing the specified number of queries to the Bloom filter occurs.

The parameters of MQTTLoader are set as follows:

- MQTT protocol version: v5.0
- The numbers of publishers and subscribers: each 1
- Ramp-up and ramp-down times: each 5 seconds
- Execution time: 70 seconds
- Payload size: 1,024 bytes
- Publish interval: 10 microseconds

We used three host machines for the publisher, subscriber, and broker. Their specs are Intel Core i9-12900 CPU, 64 GB RAM, 1 GbE network, and Ubuntu 22.04 OS.

Figure 7 shows the result. Increasing the number of queries brings about smaller throughput and higher latency. Considering the results shown in Figures 4 to 6, the proposed method is expected to perform better than the existing methods in which the average number of queries is over 100 or 1,000 in some situations.

VI. Conclusion

In this paper, we proposed a method for managing subscription topics, including wildcards, using a Bloom filter. The proposed method adds prefixes of subscription topics to the Bloom filter and enables a Trie-like search. Experimental results indicate that the proposed method achieves fewer queries to the Bloom filter than the existing methods in many cases. Particularly, when the number of topic levels is 10, and there is no matching subscription, the required number of queries is less than two percent of the existing methods. The proposed method could significantly improve the performance of distributed MQTT brokers.

In future work, we plan to conduct additional experiments to clarify the effectiveness of performance improvement by implementing the proposed method in some broker products. We will also consider the strategy of exchanging the bloom filters among brokers.

[2]https://github.com/hivemq/hivemq-community-edition (accessed June 15, 2024)

[3]https://github.com/dist-sys/mqttloader (accessed June 15, 2024)

[4]https://guava.dev/ (accessed June 15, 2024)

References

[1] MQTT, https://mqtt.org/ (accessed June 15, 2024).

[2] 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, 2003.

[3] E. Longo, A. E. Redondi, M. Cesana, A. Arcia-Moret, and P. Manzoni, "Mqtt-st: a spanning tree protocol for distributed mqtt brokers," in *Proc. IEEE International Conference on Communications*, 2020, pp. 1–6.

[4] A. Detti, L. Funari, and N. Blefari-Melazzi, "Sub-linear scalability of mqtt clusters in topic-based publish-subscribe applications," *IEEE Trans. Netw. Serv. Manage.*, vol. 17, no. 3, pp. 1954–1968, 2020.

[5] R. Banno, J. Sun, S. Takeuchi, and K. Shudo, "Interworking layer of distributed mqtt brokers," *IEICE Trans. Inf. Syst.*, vol. E102.D, no. 12, pp. 2281–2294, 2019.

[6] N. Naaman, "Large scale connectivity infrastructure for an internet of things platform," in *Software Architecture Conference*, 2016.

[7] C. Chen, B. Mandler, N. Naaman, and Y. Tock, "Publish-subscribe system with reduced data storage and transmission requirements," U.S. Patent 9 886 513, 2018.

[8] A. M. Dominguez, R. Alcarria, E. Cedeno, and T. Robles, "An extended topic-based pub/sub broker for cooperative mobile services," in *International Conference on Advanced Information Networking and Applications Workshops*, 2013, pp. 1313–1318.

[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, 1970.

[10] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, p. 490–499, 1960.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[12] OASIS Standard, "MQTT Version 5.0," https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (accessed June 15, 2024), 2019.

[13] Information-Technology Promotion Agency, https://www.ipa.go.jp/digital/architecture/guidelines/4dspatio-temporal-guideline.html (in Japanese) (accessed June 15, 2024).

[14] A. M. Dominguez, T. Robles, R. Alcarria, and E. Cedeño, "A rendezvous mobile broker for pub/sub networks," in *International Conference on Green Communications and Networking*, 2013, pp. 16–27.

[15] A. Cogoluègnes, "Stream filtering internals," https://www.rabbitmq.com/blog/2023/10/24/stream-filtering-internals (accessed June 15, 2024), 2023.

[16] J. Lee, M. Shim, and H. Lim, "Name prefix matching using bloom filter pre-searching for content centric network," *J. Netw. Comput. Appl.*, vol. 65, pp. 36–47, 2016.

[17] J. Kim, M.-C. Ko, J. Kim, and M. S. Shin, "Route prefix caching using bloom filters in named data networking," *Appl. Sci.*, vol. 10, no. 7, 2020.

[18] L. Brandl, "Mqtt topic tree & topic matching: Challenges and best practices explained," https://www.hivemq.com/blog/mqtt-topic-tree-matching-challenges-best-practices-explained/ (accessed June 15, 2024), 2023.