Wildcard-Aware Topic Management Using a Bloom Filter for Distributed MQTT Brokers

Ryohei Banno, *Member, IEEE*, and Yoshito Watanabe, *Member, IEEE*

***Abstract*—As a widely used messaging protocol for Internet of Things (IoT) systems, MQTT commonly relies on load distribution across multiple brokers in large-scale deployments, where brokers play a central role in message handling. In multi-broker deployments, brokers often share subscription topics to mitigate message flooding. Because the number of topics can be large, some prior work employs Bloom filters to store and manage them in a space-efficient manner. However, these approaches struggle with wildcard subscriptions; pattern enumeration can trigger a large number of Bloom filter membership queries, especially when no subscription matches a published topic. In this paper, we propose the Wildcard-Aware Prefix Bloom Filter (WAP-BF), which inserts topic prefixes into a Bloom filter and performs a wildcard-aware, depth-first traversal of a candidate-prefix tree derived from the published topic, with early pruning of no-match paths. Simulation results demonstrate that WAP-BF reduces the number of Bloom filter queries in many settings; in particular, under no-matching-subscription conditions WAP-BF achieves reductions of one to two orders of magnitude over prior pattern-based approaches across a wide range of parameter settings. A modified broker pipeline experiment further confirms that reducing Bloom filter lookups directly improves throughput and latency under load. In addition, a minimal three-broker forwarding experiment shows that WAP-BF produced no wasteful inter-broker forwarding in our experimental setting and achieved the lowest reported mean and percentile end-to-end latency values among the methods compared.**

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

I. Introduction

MQTT [1] is a prominent protocol for IoT systems. It adopts a publish-subscribe messaging model [2], enabling loose coupling among system components. MQTT clients communicate through an MQTT broker by using topics. Publishers send messages labeled with a topic to the broker, which then delivers them to subscribers that have subscribed to that topic.

Because an MQTT broker acts as a message aggregation point, large-scale deployments often require distributing the load across multiple brokers [3]–[6]. A distributed broker architecture can increase the number of concurrent client connections and improve communication throughput among IoT devices. Recent IoT scenarios such as smart buildings, industrial monitoring, and city-scale sensing have further intensified this need, and production-grade distributed broker platforms, including TBMQ [7], [8], EMQX [9], and HiveMQ [10], are actively being developed to accommodate such workloads. A recurring question in this line of work is how to disseminate subscription information, and in particular subscription information that contains wildcards, across brokers without incurring prohibitive communication or computation overhead. In such multi-broker settings, brokers typically share subscription topics to avoid flooding the network with unnecessary PUBLISH messages. Figure 1 illustrates a simple example: Client C_2 connects to Broker B_2 and subscribes to a topic. After Broker B_2 propagates this subscription information to Broker B_1, Broker B_1 can decide whether PUBLISH messages from Client C_1 should be forwarded to Broker B_2.

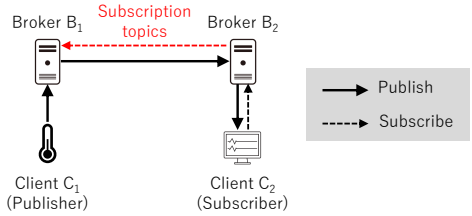

Fig. 1. Example of sharing subscriptions among brokers

Because the set of topics exchanged among brokers can be large, several strands of prior work have applied Bloom filters [11] or their variants, such as counting Bloom filters [12], to the problem of topic management. In the context of topic-based publish–subscribe messaging, Dominguez et al. [13], [14] employed a counting Bloom filter for subscription management and evaluated it with MQTT, and RabbitMQ's Stream Filtering feature [15] uses Bloom filters for a similar purpose. In content-centric and named data networking, Bloom filters are used to summarize name prefixes for routing [16], [17]. These approaches enable space-efficient membership tests but, by design, treat each inserted element as an exact string; they do not handle MQTT wildcards, so their applicability is confined to exact-match topic namespaces.

A second line of work addresses wildcard subscriptions directly. Naaman [18] and Chen et al. [19] combine a Bloom filter with a shared set of wildcard topic patterns: upon receiving a PUBLISH message, a broker enumerates the topic variants prescribed by the shared patterns and queries the Bloom filter for each of them. This enables wildcard-aware summaries to be exchanged among brokers, but the number of Bloom filter queries per PUBLISH grows with the number of distinct wildcard patterns, and is paid even when no

Ryohei Banno is with Graduate School of Social Data Science, Hitotsubashi University, Tokyo, Japan (e-mail: banno@computer.org).

Yoshito Watanabe was with Solid Surface Inc., Tokyo, Japan. He is now with the Computer and Data Science Laboratories, NTT, Inc. (e-mail: yoshito.watanabe@ntt.com).

0000–0000/00$00.00 © 2026 IEEE

subscription matches the published topic.

A gap remains between these two lines. In high-diversity settings, where many distinct wildcard shapes coexist, or in the common no-match case, where the Bloom filter ultimately reports no match, the pattern-enumeration approach issues a number of Bloom filter queries that is disproportionate to the number of actual matches. This overhead can degrade broker performance, for example, in terms of throughput and latency. To our knowledge, no prior work provides a Bloom filter-based topic summary that (i) natively supports MQTT wildcard semantics, (ii) keeps the per-PUBLISH query count low even when no subscription matches, and (iii) maintains the compactness that motivated the use of Bloom filters in the first place. This paper addresses the gap.

In this paper, we propose a novel method, the Wildcard-Aware Prefix Bloom Filter (WAP-BF), to manage wildcard topics using a Bloom filter efficiently. WAP-BF manages subscription topics by adding their prefixes to a Bloom filter. A lookup for a published topic using the Bloom filter is performed as a depth-first traversal of a *candidate-prefix tree*—a tree of prefix strings derived from the published topic that is generated on the fly and tested level by level against the Bloom filter, formalized in Section IV-C—with wildcard patterns considered explicitly at each level. WAP-BF requires fewer Bloom filter queries, especially for diverse subscription topics.

The primary contributions of this paper advance the state of the art of distributed publish-subscribe systems as follows:

- We identify an unresolved gap in Bloom filter-based topic summaries for distributed MQTT brokers: prior summaries are either wildcard-unaware or issue a number of Bloom filter queries per PUBLISH that grows with the diversity of shared wildcard patterns, with this per-PUBLISH query cost paid even when no subscription matches.
- We propose WAP-BF, a Bloom filter-based data structure that inserts topic prefixes rather than full topics and performs a wildcard-aware, depth-first traversal of a candidate-prefix tree when testing a published topic. The construction natively supports all MQTT wildcard constructs, including the parent-inclusion rule of the multi-level wildcard.
- We give a formal treatment of WAP-BF, including an inductive definition of the MQTT matching relation and of the candidate-prefix generator used by the lookup procedure, together with a correctness argument that establishes soundness (modulo Bloom filter false positives) and completeness of the lookup with respect to that matching relation.
- We provide an analytical comparison of the data sizes required to share subscription information across brokers, showing that WAP-BF is only moderately larger than a full-topic Bloom filter (by a factor proportional to topic depth) while remaining substantially smaller than sharing raw topic strings.
- We provide a simulation-based evaluation of the number of Bloom filter queries required per PUBLISH, showing that WAP-BF reduces queries in many settings compared with existing methods, and a modified-broker-pipeline experiment that shows the query-count reduction translates into throughput and latency gains under load.
- We carry out an end-to-end multi-broker forwarding experiment on three physical hosts that compares WAP-BF with Naive flooding and two Bloom filter-based baselines. In the evaluated three-broker synthetic setting, WAP-BF attained the lowest reported end-to-end latency metrics and the smallest measured fraction of wasteful inter-broker forwards among the compared methods.
- Building on a preliminary conference paper [20], this journal version adds (a) the formal correctness argument (Section IV-E), (b) the multi-broker forwarding evaluation (Section VI), (c) an analytical model of communication and memory overhead (Section V-A), (d) a representative use case for spatial IoT (Section II-C), and (e) explicit handling of topic removal in WAP-BF (Section IV-B).

The rest of this paper is organized as follows. Section II describes our assumptions about distributed MQTT brokers, including a representative spatial-IoT use case. Section III introduces related work on subscription management of MQTT brokers, covering both Bloom filter-based approaches and non-Bloom-filter alternatives such as topic trees, compressed prefix trees, and modern distributed broker platforms. Section IV explains the proposed method, WAP-BF, including the formal correctness argument in Section IV-E. Section V presents the analytical and single-broker experimental evaluation, and Section VI presents a multi-broker forwarding evaluation on three physical hosts. Finally, we conclude this paper in Section VII.

II. Distributed MQTT brokers

In large-scale systems, load distribution across multiple brokers is essential for improving throughput and reducing latency. In the following, we assume a multi-broker setting in which each client connects to one of the brokers. From the perspective of clients, publishing and subscribing work as if they were interacting with a single broker. To provide this abstraction, each broker forwards PUBLISH messages to other brokers when necessary. A simple way is to forward all PUBLISH messages, as in MQTT-ST [3].

As suggested by several studies [5], [6], an effective way to improve performance is to share subscription topics and reduce inter-broker traffic. Specifically, each broker disseminates the topics subscribed to by its local clients to other brokers. Using this shared topic information, brokers can avoid unnecessary forwarding of published messages.

In this paper, we focus on the data structure used to share and manage topics. We make the following assumptions regarding the dissemination of topics:

- A broker propagates the subscription topics of its attached clients to other brokers in response to events such as receiving SUBSCRIBE messages or detecting the arrival of a new broker.
- Upon receiving a PUBLISH message, a broker checks whether matching subscription topics exist at other brokers and then determines the appropriate action, e.g., forwarding the message to those brokers.

These assumptions are common to existing MQTT load-distribution approaches. Although various topologies and co-operation schemes, such as tree- and mesh-based designs, have been studied [21], discussing them is beyond the scope of this paper. Our goal is to develop an efficient and versatile data structure, along with its associated operations, for sharing and managing topics.

Since a set of topics can 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 uses the Bloom filter to check whether a matching subscription exists. A multi-hop topology can be efficiently supported because multiple Bloom filters can be merged into a single one via bitwise logical disjunction. For example, consider a topology in which broker B_1 connects to broker B_2, which in turn connects to broker B_3. B_3 sends its Bloom filter to B_2, and B_2 can merge it with its Bloom filter and send to B_1.

A Bloom filter may return a false positive, i.e., indicate that a topic exists when it actually does not. Such false positives are generally acceptable provided that their rate remains sufficiently low. When a broker receiving a PUBLISH message checks the existence of the topic using a Bloom filter and gets a false-positive result, it unnecessarily forwards the message to the corresponding neighbor broker. In such a case, the neighbor broker can simply discard the unnecessary message. The influence of false positives is contained within the brokers and does not extend to clients, so they do not require operations outside the protocol specification. We can suppress the false-positive rate by appropriately setting the Bloom filter size, as described in Section II-B.

Subsequent sections explain the details of MQTT topics and a Bloom filter, and illustrate an example use case where the number of topics could be large.

A. MQTT Topics

MQTT [1], [22] is a communication protocol based on the publish-subscribe messaging model. MQTT clients communicate through topics. 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.

MQTT provides two types of wildcards for subscription topics. The single-level wildcard "+" matches exactly one topic level, whereas the multi-level wildcard "#" matches zero or more topic levels. The multi-level wildcard may appear only at the end of a topic filter. The two wildcard types can also be combined. Examples are provided in Table I.

Note that the multi-level wildcard includes the parent level. For example, as shown in Table I, a subscription topic "building3/#" matches "building3".

To state the behavior of WAP-BF precisely in the following sections, we give a formal description of topics, subscription filters, and the matching relation between them. Let L denote the set of valid topic-level tokens, i.e., non-empty strings over the MQTT topic-level alphabet that do not contain "/", "+", or "#". A published topic is a finite non-empty sequence $t = l_0/l_1/\cdots/l_{d-1}$ with $l_i \in L$, and we write $|t| = d$ for its depth. A subscription filter σ is a finite non-empty sequence over $L \cup \{+, \#\}$ in which the multi-level wildcard "#" may appear only as the last level.

TABLE I
EXAMPLES OF MQTT TOPICS

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

We define the matching relation $\mathrm{match}(\sigma, t) \in \{0, 1\}$ between a subscription filter σ and a published topic t inductively on the length of σ and t. Writing a/σ' for a filter whose first level is a and whose remainder is σ', and similarly l/t' for a topic, we set: (i) $\mathrm{match}(l, l') = 1$ iff $l = l'$, for $l, l' \in L$; (ii) $\mathrm{match}(l/\sigma', l'/t') = \mathrm{match}(\sigma', t')$ if $l = l'$, and 0 otherwise, for $l, l' \in L$; (iii) $\mathrm{match}(+, l) = 1$ for any $l \in L$; (iv) $\mathrm{match}(+/\sigma', l/t') = \mathrm{match}(\sigma', t')$ for any $l \in L$; (v) $\mathrm{match}(\#, t) = 1$ for any t with $|t| \geq 1$; (vi) $\mathrm{match}(l/\#, t) = 1$ iff $t = l$ or $t = l/t'$ for some t', for $l \in L$ (parent inclusion); (vii) $\mathrm{match}(+/\#, t) = 1$ for any t with $|t| \geq 1$. All other cases give $\mathrm{match}(\sigma, t) = 0$. Clauses (vi) and (vii) encode the parent-inclusion rule of the multi-level wildcard illustrated above.

B. Bloom Filter

A Bloom filter [11], [23] is a probabilistic data structure for testing set membership. It consists of a bit array of length b, with all bits initially set to 0. To insert an element, we compute k hash values using k hash functions, map them to k positions in the bit array, and set the corresponding bits to 1. To query an element, we compute the same k hash values, obtain the corresponding positions, and inspect those bits. If all of them are 1, the element is considered to be present with high probability; otherwise, it is definitely absent. Multiple Bloom filters can be merged efficiently via a bitwise OR operation.

A membership query may return a false positive. The false-positive rate p_{f} is given by

$$p_{\mathrm{f}} = \left(1 - e^{-\frac{k n_{\mathrm{e}}}{b}}\right)^k$$

where n_{e} is the number of elements. Let k^* denote the value of k that minimizes p_{f}:

$$k^* = \arg\min_k p_{\mathrm{f}}$$

The optimal number of hash functions is given by

$$k^* = \frac{b}{n_{\mathrm{e}}} \ln 2.$$

Assuming that $k = k^*$, the relationship between b and p_{f} is expressed as

$$b = -\frac{\ln p_{\mathrm{f}}}{(\ln 2)^2} n_{\mathrm{e}}. \tag{1}$$

Note that the growth beyond the initially provisioned set size can be addressed by variants such as Scalable Bloom Filters [24].

C. Use case

We can consider various applications where distributed MQTT brokers are useful. In this section, we introduce a use case: operating service robots inside a commercial building.

We assume a massive IoT scenario where commercial buildings are equipped with numerous sensor devices. Various service robots, with functionality such as cleaning, patrolling, and transportation, move and provide services to workers. The robots and sensors exchange information to efficiently provide appropriate services as needed. By treating spaces as topics, they can communicate with each other via MQTT in a decoupled manner. Each sensor device publishes the sensed data to the topic corresponding to its location, whereas each robot subscribes to the area it is responsible for. In such a scenario, we need distributed MQTT brokers for handling large-scale communication among robots and sensors.

We can consider using Spatial-ID [25], a universal space identification scheme, to treat spaces as topics. Spatial-ID is similar to Slippy map tilenames [26] but extended to three dimensions; that is, space is divided into voxels in a hierarchical manner, and each voxel is assigned an identifier (ID). A Spatial-ID is expressed as "$z/f/x/y$", where z is the zoom level, f is the elevation index, x is the longitude index, and y is the latitude index. Zoom level $z = 0$ corresponds to the largest voxel. The maximum zoom level, i.e., the zoom level of the smallest voxel, is assumed to be 26, but we can consider larger zoom levels as needed. Increasing the zoom level by one involves dividing the voxel by eight. A voxel at zoom level $z = 26$ is approximately $0.6\,\mathrm{m} \times 0.6\,\mathrm{m}$ and $0.5\,\mathrm{m}$ high.

Encoding the containment relationship of Spatial-IDs into the MQTT topic hierarchy enables flexible spatial subscriptions. For example, we can consider the topic structure "$v_0/v_1/\ldots/v_{26}$", where v_i denotes a Spatial-ID of a voxel at zoom level $z = i$ and v_{i+1} denotes a Spatial-ID of a voxel at zoom level $z = i + 1$, which is a child voxel of v_i. Note that each Spatial-ID v_i originally uses slashes as separators, and we need to replace them with other symbols, such as hyphens like "z-f-x-y", to avoid conflicts with MQTT topic separators.

Assuming the above topic structure, a sensor device can publish to a topic indicating its location at the largest zoom level. Additional information can be added at the end of the topic, such as sensor types. For example, a temperature sensor may publish to a topic like "$v_0/v_1/.../v_{26}$/temp". A service robot can subscribe to topics corresponding to the area it is responsible for. Wildcards enable robots to subscribe to areas of various sizes efficiently. For example, "$v_0/v_1/.../v_{24}$/#" indicates an area of approximately $2.39\,\mathrm{m} \times 2.39\,\mathrm{m}$. "$v_0/v_1/.../v_{24}$/+/+/temp" indicates temperature sensors included in the area of v_{24}.

In such an application, the number of topics could be enormous, while each topic has tens of levels. For example, a single commercial building of $50\,\mathrm{m} \times 50\,\mathrm{m}$ and $100\,\mathrm{m}$ high contains over 1.3 million voxels at a zoom level of $z = 26$, which are potential topics. If we consider a smart city consisting of such buildings, the number of potential topics could be substantially larger. The length of each topic could be over several hundred characters, assuming the topic structure "$v_0/v_1/\ldots/v_{26}$", though this depends on the location. Additional topic levels, like the above-mentioned sensor types, could make it longer. A topic is encoded as a UTF-8 string in the MQTT protocol, and each character requires one byte if it is an ASCII character. Distributed MQTT brokers for such applications require a space-saving way to share and manage the vast number of topics.

III. Related work

Owing to their excellent space and time efficiency, Bloom filters are widely used for topic management. Dominguez et al. [13], [14] employed a counting Bloom filter [12], a variant of the Bloom filter, for subscription management in topic-based publish-subscribe messaging, and evaluated it with MQTT. Beyond MQTT, RabbitMQ provides a "Stream Filtering" feature that internally uses Bloom filters for subscription management [15]. Related work in content-centric networking and named data networking [16], [17] has also adopted Bloom filter-based routing. These studies are similar to our approach in that they insert name prefixes into a Bloom filter. However, none of them addresses efficient support for wildcard subscriptions. In particular, the counting Bloom filter-based design of Dominguez et al. treats each inserted topic as an exact string and does not support MQTT wildcard semantics, which confines applicability to exact-match topic namespaces and leaves the wildcard-summary problem targeted by this paper open.

Naaman [18] and Chen et al. [19] propose using a Bloom filter to disseminate subscription topics across multiple MQTT brokers. Their approach explicitly accounts for wildcard subscriptions. In addition to exchanging a Bloom filter representing subscribed topics, each broker also shares a set of wildcard topic patterns with other brokers. A wildcard topic pattern specifies the topic levels at which the two wildcard types appear. For example, suppose a broker holds the subscription topics "a/+/c", "x/+/z", "a/+/+", and "+/+/#". Then the set of patterns to be shared can be expressed as [{1;-1}, {1,2;-1}, {0,1;2}]. The pattern {0,1;2} indicates that there exists at least one subscription topic in which the single-level wildcard appears at levels 0 and 1, and the multi-level wildcard appears at level 2. Within a pattern, -1 denotes the absence of the corresponding wildcard. Here, we assume the lowest topic level is 0. When a broker receives a PUBLISH message, it uses the Bloom filter to test the published topic as well as a set of derived variants generated from the shared patterns. For instance, if the published topic is "x/y/z" and the shared patterns are as above, the broker queries the Bloom filter for "x/y/z", "x/+/z", "x/+/+", and "+/+/#". This method implicitly assumes that the number of wildcard patterns is small; however, highly diverse subscription topics can lead to many patterns and consequently a large number of Bloom filter queries. More precisely, the number of Bloom filter

queries per PUBLISH is proportional to the number of distinct wildcard patterns shared across brokers, and this cost is paid for every PUBLISH, including those for which no subscription matches. The method therefore scales poorly with wildcard pattern diversity and does not offer a way to terminate early on no-match published topics.

Several studies related to publish-subscribe messaging also utilize Bloom filters. Parzyjegla et al. [27] implemented content-based publish-subscribe directly on the network layer using OpenFlow-enabled switches, where publishers embed an in-packet Bloom filter representing the subscriptions matched by each notification and switches consult the bitvector to forward packets accordingly. This line of work aims at low-latency routing by avoiding application-layer broker overlays; in contrast, our work focuses on distributed MQTT brokers and addresses the wildcard-aware topic discovery problem when brokers exchange compact topic summaries.

Badreddine et al. [28] studied an IoT data marketplace architecture in which MQTT-based publish-subscribe activities are recorded and verified via smart contracts on a blockchain, and proposed multiple traceability/monetization designs including a Bloom filter-based option for efficient verification of data exchange. While their Bloom filter is used to reduce on-chain verification/storage overhead for accounting and fraud detection in a marketplace setting, our work uses Bloom filters as a data structure for inter-broker topic management, and specifically targets efficient processing of MQTT wildcard semantics during topic lookup and forwarding decisions.

For the remainder of this paper, BF denotes the conventional full-topic Bloom-filter-based topic-management method, in which each complete subscription topic is inserted into a Bloom filter as a single element. BF-Enum denotes its wildcard-enumeration extension; for each PUBLISH operation, it enumerates all wildcard variants of the published topic and queries the Bloom filter for each variant. BFP, which stands for "BF with patterns," denotes the pattern-sharing approach proposed in [18] and [19].

A. Non-Bloom-filter alternatives

Although Bloom filters are a natural fit for compact inter-broker summaries, they are not the only candidate data structure. For completeness, we discuss the principal non-Bloom-filter alternatives that could, in principle, be used for the same purpose, and explain why they are a poor match for the problem addressed in this paper.

Topic trees and tries. Production MQTT brokers commonly maintain a per-broker subscription index as a *topic tree*, i.e., a trie whose nodes correspond to topic levels and whose leaves carry subscription metadata [29], [30]. Topic trees natively support both the single-level and the multi-level wildcard, but supporting wildcards inflates the worst-case lookup cost from a single root-to-leaf descent to $O(2^{d_{\mathrm{P}}})$ traversal states for a published topic of depth d_{P}, because at every non-terminal level the traversal must follow both the literal branch and the single-level-wildcard "+" branch (and additionally test multi-level-wildcard "#" terminals). Moreover, topic trees are designed as local subscription indexes, not as compact summaries to be exchanged among brokers. The number of internal nodes grows with the number of distinct topic prefixes held by a broker, and serializing a topic tree for inter-broker dissemination yields a representation whose size is, in the worst case, proportional to the number of local topics rather than to a fixed bit budget.

Compressed prefix trees. Space-efficient trie variants such as PATRICIA [31] and other radix-tree representations reduce the node count by collapsing chains of single-child nodes. They are attractive when the topic namespace has long shared prefixes. MQTT wildcard filters can be supported naturally, but the compression benefit diminishes as the variety of wildcard patterns grows, and the serialized size still depends on the number of unique prefixes rather than on a fixed bit budget.

Cuckoo filters. A cuckoo filter [32] is a compact approximate-membership data structure that, like a Bloom filter, supports fixed-size summaries and bounded-error membership tests, but additionally supports element deletion. Used as a standalone summary, however, a cuckoo filter answers only exact-match membership queries on the inserted elements; it provides no built-in mechanism to evaluate MQTT wildcard semantics, so a topic summary that consisted of a cuckoo filter alone (storing full topic strings) would not support wildcards. A cuckoo filter could in principle replace the underlying Bloom filter in WAP-BF, and doing so would simplify the unsubscribe handling discussed in Section IV-B. This substitution is orthogonal to the core mechanism of WAP-BF (prefix insertion and wildcard-aware depth-first lookup) and is therefore outside the scope of this paper; we treat the combination of WAP-BF with a cuckoo filter as a direction for future work.

Taken together, these alternatives illustrate why the problem addressed in this paper is not resolved by simply switching to a different data structure: topic trees and compressed prefix trees are not designed as fixed-size inter-broker summaries, and cuckoo filters address a different aspect (deletion) of the same underlying approximate-membership problem that Bloom filters solve.

B. Modern distributed MQTT broker platforms

In the practitioner literature, several production-grade distributed MQTT broker platforms have been developed, including TBMQ [7], [8], EMQX [9], and HiveMQ [10]. These platforms provide clustered deployments and address broader concerns such as persistence, failover, authorization, and multi-tenancy that go well beyond the scope of this paper. WAP-BF is not intended as a competitor to such platforms. Rather, it is a complementary mechanism: an efficient, wildcard-aware, fixed-size inter-broker topic summary that could be adopted inside any such platform as the data structure used to share subscription information among nodes. We therefore position WAP-BF as an algorithmic contribution that is independent of any specific broker implementation, and we note the existing platforms here to clarify this boundary rather than to evaluate against them.

TABLE II
QUALITATIVE COMPARISON OF TOPIC-MANAGEMENT METHODS FOR DISTRIBUTED MQTT BROKERS.

Method	Wildcard support	Inter-broker summary size (worst case)	Per-PUBLISH lookup cost for given d_{p} (worst case)
WAP-BF (proposed)	Yes	Fixed*	$O(2^{d_{\mathrm{p}}})$
BF [13]	No	Fixed*	$O(1)$
BF-Enum	Yes	Fixed*	$O(2^{d_{\mathrm{p}}})$
BFP [18], [19]	Yes	Fixed* + pattern list	$O(2^{d_{\mathrm{p}}})$
Topic tree / trie [29], [30]	Yes	$O(n_{\mathrm{t}} \cdot d_{\mathrm{s}} \cdot s_{\mathrm{l}})$	$O(2^{d_{\mathrm{p}}})$
Compressed prefix tree [31]	Yes	$O(n_{\mathrm{t}} \cdot d_{\mathrm{s}} \cdot s_{\mathrm{l}})$	$O(2^{d_{\mathrm{p}}})$
Cuckoo filter [32]	No	Fixed*	$O(1)$

* Fixed for a provisioned capacity.
n_{t}, d_{s}, d_{p}, and s_{l} denote the number of local topics, subscription-topic depth, published-topic depth, and topic-level token size, respectively.

C. Critical summary

Across the three families discussed above—Bloom filter-based summaries, non-Bloom-filter data structures, and modern broker platforms—no existing design simultaneously provides (i) native support for MQTT wildcard semantics, (ii) an inter-broker representation whose size is fixed in advance rather than proportional to the number of topics, and (iii) a per-PUBLISH lookup cost that stays low even in the common no-match case. Bloom filter-only designs [13] miss (i); the pattern-sharing approach of Naaman [18], [19] satisfies (i) but compromises (iii) when the number of shared wildcard patterns grows; topic trees and compressed prefix trees [29]–[31] provide (i) but miss (ii); and cuckoo filters [32] are orthogonal to the wildcard-lookup concern. WAP-BF is designed to address this gap by combining prefix insertion into a Bloom filter with a wildcard-aware, depth-first traversal that terminates early on no-match paths, as summarized in Table II.

In Table II, "Inter-broker summary size" refers to the size of the representation exchanged across brokers. Given a PUBLISH topic of depth d_{p}, the "Per-PUBLISH lookup cost" refers to the number of membership tests or traversal states required per PUBLISH message to check whether matching subscription topics exist at other brokers and decide whether to forward the PUBLISH message. The symbols n_{t}, d_{s}, and s_{l} denote the number of local topics, the depth of subscription topics, and the size of a topic-level token, respectively. Among methods that simultaneously provide wildcard support and a fixed-size inter-broker summary—WAP-BF, BF-Enum, and BFP—a more detailed qualitative comparison is given in Section IV-D. The three methods share the same worst-case per-PUBLISH lookup cost of $O(2^{d_{\mathrm{p}}})$, but in many practically relevant cases WAP-BF achieves substantially fewer Bloom filter queries than BF-Enum and BFP, as we demonstrate empirically in Section V.

IV. PROPOSED METHOD

We propose WAP-BF, a Bloom filter-based method that efficiently supports wildcard topics. WAP-BF inserts prefixes of subscription topics into a Bloom filter, enabling membership checks via a depth-first traversal of a candidate-prefix tree [31], [33].

As described in Section II, we assume that each broker builds a Bloom filter from its local subscriptions and shares it with other brokers. Upon receiving a PUBLISH message, a broker consults the Bloom filters disseminated by other brokers to test whether the published topic may match remote subscriptions and to decide subsequent actions, such as forwarding the message to selected brokers. The following sections detail topic management with a Bloom filter in WAP-BF.

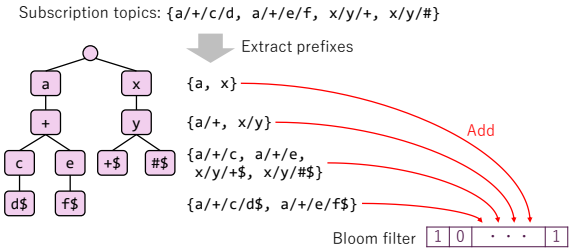

Fig. 2. Adding topic prefixes to a Bloom filter

A. Adding topics to a Bloom filter

To represent topics in a Bloom filter, we encode them using their prefixes. Here, a prefix is defined as the sequence of consecutive topic levels from the beginning of a topic up to a given level. For example, for the topic "foo/bar/baz", the prefixes are "foo", "foo/bar", and "foo/bar/baz".

Before insertion, we append a terminal marker to the final topic level in the internal representation used for Bloom filter insertion. This marker is not a literal MQTT topic character; it is encoded so that it cannot collide with any valid topic level or wildcard token. For example, an implementation may use length-prefixed encoding or type-length-value (TLV)-style encoding, where the terminal marker is represented as an internal tag in the byte string supplied to the hash functions. For readability, we denote this internal terminal marker by \$ in the remainder of this paper. We then insert each prefix into the Bloom filter. Figure 2 illustrates this procedure. For instance, inserting the subscription topic "a/+/c/d" results in adding the prefixes "a", "a/+", "a/+/c", and "a/+/c/d\$" to the Bloom filter.

We formalize the procedure above by a function $\mathrm{prefixes}$ that maps a subscription filter to the set of strings inserted into the Bloom filter. For a filter $\sigma = l_0/l_1/\cdots/l_{d_{\mathrm{s}}-1}$ with

$l_i \in L \cup \{+\}$ (i.e., not ending with "#"), we set

$$\begin{aligned}\mathrm{prefixes}(\sigma) = \{\, & l_0,\ l_0/l_1,\ \ldots,\ l_0/\cdots/l_{d_\mathrm{s}-2},\\ & l_0/\cdots/l_{d_\mathrm{s}-1}\$\,\},\end{aligned}$$

where the terminal symbol \$ is appended only to the deepest prefix. For a filter $\sigma = l_0/\cdots/l_{d_\mathrm{s}-1}/\#$ ending with the multi-level wildcard,

$$\begin{aligned}\mathrm{prefixes}(\sigma) = \{\, & l_0,\ l_0/l_1,\ \ldots,\ l_0/\cdots/l_{d_\mathrm{s}-1},\\ & l_0/\cdots/l_{d_\mathrm{s}-1}/\#\$\,\},\end{aligned}$$

i.e., the terminal marker is placed on the full filter including the trailing "#". For a filter $\sigma = \#$ consisting solely of the multi-level wildcard,

$$\mathrm{prefixes}(\sigma) = \{\, \#\$\,\},$$

i.e., only the terminal-marked wildcard symbol is inserted. Given a local subscription set S, the broker inserts $\bigcup_{\sigma \in S} \mathrm{prefixes}(\sigma)$ into its Bloom filter.

The Bloom filter size should be chosen appropriately to keep false positives under control. Given an allowable false-positive rate p_f and an assumed upper bound on the number of inserted elements n_e, the required bit-array length b can be computed using Equation (1) in Section II-B. The optimal number of hash functions, k^*, can also be derived accordingly. For example, with $p_\mathrm{f} = 0.001$ and $n_\mathrm{e} = 100{,}000$, we obtain $b \approx 1{,}437{,}759$ bits and $k^* \approx 10$.

B. Handling UNSUBSCRIBE

Upon receiving an UNSUBSCRIBE message from a client, some prefixes may need to be removed from the Bloom filter. There are several variants of a Bloom filter to handle deletions, such as the counting Bloom filter [12], the cuckoo filter [32], and the deletable Bloom filter [34]. In WAP-BF, using a counting Bloom filter is straightforward because we need to consider multiple insertions of the same element, i.e., the same prefix. With a counting Bloom filter, we can remove the elements by decrementing the counters at the positions associated with the prefixes of the unsubscribed topic.

While each broker manages subscription topics of its clients using a counting Bloom filter, it can generate a standard Bloom filter from the counting Bloom filter and share the standard one with neighboring brokers. That is, each broker manages topic information from other brokers as standard Bloom filters. When the counting Bloom filter is updated, a broker regenerates a standard Bloom filter and shares it with neighboring brokers. The compactness of a Bloom filter makes such a replacement-based approach practical. Replacement of filters is idempotent, so we need not consider complex consistency handling, which is an advantage of using Bloom filters.

Another approach is to share a counting Bloom filter with neighboring brokers as-is. When a broker receives an UNSUBSCRIBE message and removes a prefix from the filter, it shares the prefix to be removed with neighboring brokers. Although it may be meaningful to suppress communication traffic, strict consistency management is required. If the prefix information is duplicated and a recipient does not recognize it as a duplicate deletion request, an inconsistent state is created, and a false negative may occur during searching.

The selection of the above two approaches, i.e., sharing a standard Bloom filter or a counting Bloom filter, depends on the design policy of the distributed broker system, and it is outside the scope of this paper. In the remainder of this paper, we assume that brokers share standard Bloom filters.

Beyond the choice of filter variant, several operational concerns arise when WAP-BF is deployed across a cluster of brokers under realistic workloads. In an actual distributed broker implementation, maintenance and redistribution of topic summaries belong to the control plane, whereas lookup of the summaries for forwarding decisions belongs to the PUBLISH data plane. SUBSCRIBE and UNSUBSCRIBE events are first reflected in the broker's local counting Bloom filter. A regenerated standard Bloom filter can then be redistributed to neighboring brokers according to an implementation-specific policy; for example, multiple updates may be batched and transmitted periodically or after a configured number of updates have accumulated. This decouples client-side subscription churn from inter-broker communication and bounds the update rate seen by neighboring brokers. The PUBLISH data plane applies Algorithm 1 to the most recently received summary for each neighboring broker. Inconsistencies between the summaries maintained by neighboring brokers and a broker's local subscription set—for example, those caused by transient network partitions or broker failures—are resolved through the same redistribution mechanism. Because the replacement operation is idempotent, no explicit consistency protocol among the summaries is required; consistency is restored once the affected broker redistributes its reconstructed standard filter. Handling PUBLISH messages that would otherwise have been forwarded during such an inconsistency window is delegated to the MQTT quality of service (QoS) and session-persistence mechanisms, rather than to the summary structure itself. How those MQTT mechanisms are extended to a multi-broker setting depends on the design of distributed broker platforms, such as those described in Section III-B, and is therefore outside the scope of this paper.

C. Wildcard-aware lookup for a published topic

Searching for a published topic t_p using the Bloom filter is performed as a depth-first traversal of a candidate-prefix tree derived from t_p, as illustrated in Figure 3; the tree is formalized below. Here, the *candidate-prefix tree* for a published topic t_p is the rooted tree whose nodes are prefix strings to be tested against the Bloom filter: the root is the empty prefix, and the children of a node π at level i are the candidate prefixes obtained by extending π with one of l_i, "+", or "#" (with the parent-inclusion rule of the multi-level wildcard handled at the deepest level). The tree is generated on the fly from t_p and is traversed in depth-first order, pruning any subtree whose root prefix is absent from the Bloom filter. The candidate generator cand_i given later in this subsection makes this construction precise. Let $d_\mathrm{p} = |t_p|$ denote the number of topic levels in t_p, and let $l_0, l_1, \ldots, l_{d_\mathrm{p}-1} \in L$ denote its individual levels (cf. Section II-A).

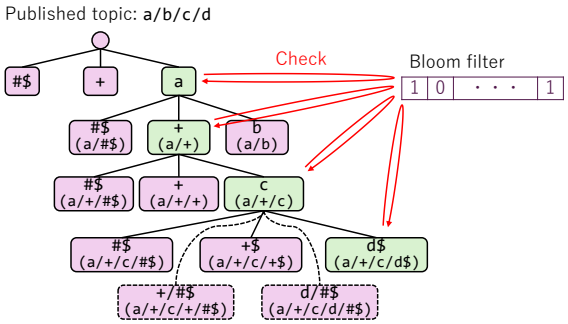

Fig. 3. Wildcard-aware lookup of a published topic using a Bloom filter

We begin by querying the Bloom filter for the level-0 prefixes "#$", "+", and l_0. These prefixes correspond to subscription topics that could potentially match t_p. If none of them is present in the Bloom filter, we conclude that no matching subscription exists. Conversely, if at least one is present, we proceed to the next level.

At level i ($1 \le i \le d_{\mathrm{p}} - 1$), we generate candidate prefixes by appending one of "#$", "+", or l_i to each prefix that was found in the Bloom filter at level $i-1$, and query the Bloom filter for these extended prefixes. This process is repeated until the depth reaches $d_{\mathrm{p}} - 1$.

When the depth equals $d_{\mathrm{p}} - 1$, we append the terminal symbol "$" to candidates ending with "+" or $l_{d_{\mathrm{p}}-1}$ before querying, to represent exact termination at that level. In addition, we must account for the fact that the multi-level wildcard matches the parent level as noted in Section II-A; namely, the subscription topic "$l_0/\ldots/l_{d_{\mathrm{p}}-1}$/#" matches the published topic "$l_0/\ldots/l_{d_{\mathrm{p}}-1}$". Therefore, at depth $d_{\mathrm{p}} - 1$ we also query candidates obtained by appending "/#$" to prefixes ending with "+" or $l_{d_{\mathrm{p}}-1}$.

During the search, if any candidate prefix ending with "$" is found in the Bloom filter, the procedure terminates and we conclude that a matching subscription exists with high probability. If no such terminal prefix is found, the result indicates that there is no matching subscription.

We formalize the above candidate generation by a function cand_i that, given a parent prefix π reached at level $i-1$ and the published topic $t_p = l_0/\cdots/l_{d_{\mathrm{p}}-1}$, returns the set of candidate strings to be tested at level i. For $0 \le i \le d_{\mathrm{p}} - 2$, we set

$$\mathrm{cand}_i(\pi, t_p) = \{\, \pi \diamond \#\$,\ \pi \diamond +,\ \pi \diamond l_i \,\},$$

and for $i = d_{\mathrm{p}} - 1$ we set

$$\mathrm{cand}_{d_{\mathrm{p}}-1}(\pi, t_p) = \{\, \pi \diamond \#\$,\ \pi \diamond +\$,\ \pi \diamond l_{d_{\mathrm{p}}-1}\$,\ \pi \diamond +/\#\$,\ \pi \diamond l_{d_{\mathrm{p}}-1}/\#\$ \,\},$$

where $\pi \diamond x$ denotes the concatenation π/x when π is non-empty and x otherwise, and the level-0 candidates are obtained as $\mathrm{cand}_0(\varepsilon, t_p)$ with ε the empty prefix. The last two elements of $\mathrm{cand}_{d_{\mathrm{p}}-1}$ capture the parent-inclusion rule of the multi-level wildcard (clauses (vi) and (vii) of match). Algorithm 1 then performs a depth-first traversal of the tree whose root is ε and whose children at level i under a parent π are $\mathrm{cand}_i(\pi, t_p)$, pruning any subtree whose root prefix is absent from the Bloom filter.

Figure 3 illustrates an example of the search procedure, assuming the Bloom filter shown in Figure 2. The green boxes indicate prefixes that have been inserted into the Bloom filter. In this example, the prefixes "a", "a/+", and "a/+/c" are present in the Bloom filter, and the terminal prefix "a/+/c/d$" is also found. Therefore, the search concludes that a matching subscription exists.

In the above procedure, we adopt the following policy:

- Perform a depth-first search.
- At each level, query the prefix ending with "#$" first.

These choices allow the search to terminate early if a matching subscription exists.

Algorithm 1 presents the pseudocode. The function SEARCH returns **true** if the input `topic` has a matching subscription with high probability, and **false** if it definitely has none. PARSE_TOPIC splits a topic into its constituent levels and returns them as a list. SEARCH_HELPER is a recursive function used to implement depth-first search. It takes the following arguments: `prefix` is a prefix that was found in the Bloom filter at the previous step; `level` denotes the depth of prefixes currently being examined; and `lvList` is the list of topic levels in the published topic. The auxiliary function CAND returns the candidate list `cand` at the given level; it is a direct implementation of $\mathrm{cand}_{\mathtt{level}}(\mathtt{prefix}, \mathtt{lvList})$ defined above, with the two cases ($\mathtt{level} < |\mathtt{lvList}| - 1$ vs. $\mathtt{level} = |\mathtt{lvList}| - 1$) corresponding one-to-one with the two cases in the definition of cand_i. CHECK_BF tests whether the given element is contained in the Bloom filter. A formal correctness argument for Algorithm 1, including soundness and completeness with respect to match, is given in Section IV-E.

D. Discussion

Table III summarizes the asymptotic analysis. We consider two baselines, BF-Enum and BFP, described in Section III. We use the following notation:

- n_{t}: the number of subscription topics.
- d_{s}: the number of levels in a subscription topic.
- d_{p}: the number of levels in a published topic.
- n_{w}: the number of wildcard patterns.
- p_{f}: allowable false-positive rate in a Bloom filter.

In terms of memory usage, we assume that the Bloom filter size is appropriately configured based on the expected number of elements to be inserted, with a specified allowable false-positive rate p_{f}. We also assume that every subscription topic has d_{s} levels for simplicity. BF-Enum requires

$$O\left(n_{\mathrm{t}} \log \frac{1}{p_{\mathrm{f}}}\right)$$

because the size of a Bloom filter b can be calculated as

$$b = -\frac{\ln p_{\mathrm{f}}}{(\ln 2)^2} n_{\mathrm{t}}$$

from Equation (1) as described in Section II-B.

Regarding BFP, wildcard patterns require additional memory beyond a Bloom filter. A pattern of single-level wildcards

TABLE III
ASYMPTOTIC ANALYSIS

	BF-Enum	BFP	WAP-BF
Memory space	$O\left(n_\mathrm{t} \log \frac{1}{p_\mathrm{f}}\right)$	$O\left(n_\mathrm{t} \log \frac{1}{p_\mathrm{f}} + n_\mathrm{w}\left(d_\mathrm{s} + \log d_\mathrm{s}\right)\right)$	$O\left(n_\mathrm{t} d_\mathrm{s} \log \frac{1}{p_\mathrm{f}}\right)$
Search time (average)	Depends on wildcard usage. (Experimental evaluation is shown in Section V-B.)		
Search time (worst)	$O\left(2^{d_\mathrm{p}}\right)$	$O\left(2^{d_\mathrm{p}}\right)$	$O\left(2^{d_\mathrm{p}}\right)$

Algorithm 1 Search process

```
1: function SEARCH(topic)
2:     lvList ← PARSE_TOPIC(topic)
3:     return SEARCH_HELPER("", 0, lvList)
4: end function

5: function SEARCH_HELPER(prefix, level, lvList)
6:     if level ≥ lvList.LENGTH( ) then
7:         return false
8:     end if
9:     cand ← CAND(prefix, level, lvList)
10:    for i ← 0 to cand.LENGTH( ) −1 do
11:        c ← cand[i]
12:        if CHECK_BF(c) = true then
13:            if c.ENDS_WITH("$") = true then
14:                return true
15:            else
16:                r ← SEARCH_HELPER(c, level+1, lvList)
17:                if r = true then
18:                    return true
19:                end if
20:            end if
21:        end if
22:    end for
23:    return false
24: end function

25: function CAND(prefix, level, lvList)
    ▷ implements cand_level(prefix, lvList)
26:    delim ← ""
27:    if level > 0 then
28:        delim ← "/"
29:    end if
30:    pd ← CONCAT(prefix, delim)
31:    lv ← lvList[level]
32:    cand[0] ← CONCAT(pd, "#$")
33:    if level = lvList.LENGTH( ) −1 then
34:        cand[1] ← CONCAT(pd, "+$")
35:        cand[2] ← CONCAT(pd, lv, "$")
36:        cand[3] ← CONCAT(pd, "+/#$")
37:        cand[4] ← CONCAT(pd, lv, "/#$")
38:    else
39:        cand[1] ← CONCAT(pd, "+")
40:        cand[2] ← CONCAT(pd, lv)
41:    end if
42:    return cand
43: end function
```

can be represented using a d_s-bit string by setting the bits corresponding to the wildcard occurrence positions. For a multi-level wildcard, the number of possible states of the occurrence position is $d_\mathrm{s}+1$: one state for "no occurrence," and d_s states for "one of the topic levels is a multi-level wildcard." To represent these $d_\mathrm{s}+1$ distinct states, we need an r-bit string where

$$2^r \geq d_\mathrm{s} + 1$$

is satisfied. Hence, the minimum value of r is

$$\lceil \log_2(d_\mathrm{s} + 1) \rceil .$$

Therefore, the space complexity for BFP is

$$O\left(n_\mathrm{t} \log \frac{1}{p_\mathrm{f}} + n_\mathrm{w}\left(d_\mathrm{s} + \log d_\mathrm{s}\right)\right).$$

Note that the actual required memory space for wildcard patterns, i.e.,

$$O\left(n_\mathrm{w}\left(d_\mathrm{s} + \log d_\mathrm{s}\right)\right),$$

is expected to be relatively small and not dominant. We demonstrate it with a particular assumption in Section V-A.

WAP-BF requires

$$O\left(n_\mathrm{t} d_\mathrm{s} \log \frac{1}{p_\mathrm{f}}\right),$$

since the expected number of elements to be inserted into a Bloom filter is the number of prefixes which is expressed as $n_\mathrm{t} d_\mathrm{s}$. Although this is larger than BF-Enum and probably BFP, the impact is not expected to be substantial, because a Bloom filter remains much smaller than storing all topics as they are. Note that p_f is the per-query false-positive rate of the underlying Bloom filter, not the per-PUBLISH misjudgment rate. For the same p_f, WAP-BF tends to achieve a lower per-PUBLISH misjudgment rate than BF-Enum and BFP, as discussed at the end of this section.

The average search cost depends heavily on wildcard usage; therefore, we evaluate it via simulation in Section V-B. Because WAP-BF can prune the search when none of the candidate prefixes at a given level is present, it is expected to issue fewer Bloom filter queries than existing approaches that exhaustively enumerate wildcard patterns.

Finally, we discuss the worst-case search time. Here, we consider a published topic with d_p levels. The worst case of BF-Enum occurs when there is no matching subscription topic; BF-Enum enumerates all possible patterns with wildcards derived from the published topic and searches for them in a Bloom filter. We can consider the number of patterns as follows.

- No wildcard: one pattern.
- Multi-level wildcard only: $d_\mathrm{p} + 1$ patterns.
  This count accounts for the fact that a multi-level wildcard includes the parent level, as described in Section II-A.
- Single-level wildcard only: $2^{d_\mathrm{p}} - 1$ patterns.
  The number of patterns including m single-level

wildcards can be represented as ${}_{d_\mathrm{p}}C_m$. So, the total number is

$$ {}_{d_\mathrm{p}}C_1 + \ldots + {}_{d_\mathrm{p}}C_{d_\mathrm{p}}. $$

In general, from the binomial theorem, the sum of binomial coefficients can be calculated as

$$ \sum_{m=0}^{n} \binom{n}{m} = 2^n. $$

Therefore, the number of patterns is $2^{d_\mathrm{p}} - 1$.

- Mixed wildcards: $2^{d_\mathrm{p}+1} - d_\mathrm{p} - 2$ patterns.
  The first topic level is indexed as 0, and a multi-level wildcard can appear at level i where $1 \leq i \leq d_\mathrm{p}$ in this mixed wildcard case. The upper bound $i = d_\mathrm{p}$ accounts for the fact that a multi-level wildcard includes the parent level. When a multi-level wildcard appears at level i, the number of patterns is

$$ \sum_{m=1}^{i} \binom{i}{m}. $$

Therefore, the number of patterns for the mixed wildcard case is

$$ \sum_{i=1}^{d_\mathrm{p}} \sum_{m=1}^{i} \binom{i}{m} = 2^{d_\mathrm{p}+1} - d_\mathrm{p} - 2. $$

From these, the total number of wildcard patterns is

$$ 3 \times 2^{d_\mathrm{p}} - 1. \tag{2} $$

Therefore, BF-Enum requires

$$ O\left(2^{d_\mathrm{p}}\right) $$

search time in the worst case.

The worst case for BFP occurs when there is no matching subscription topic and all possible wildcard patterns exist. The worst-case search time for BFP can also be represented as

$$ O\left(2^{d_\mathrm{p}}\right). $$

In other words, BFP issues

$$ O\left(n_\mathrm{w}\right) $$

membership queries per PUBLISH, and n_w can be

$$ O\left(2^{d_\mathrm{p}}\right) $$

in the worst case.

In regard to WAP-BF, the worst case is exploring every branch of the candidate-prefix tree implied by the Bloom filter without terminating. At depth i ($0 \leq i \leq d_\mathrm{p} - 2$) there are 2^{i+1} nodes that have child nodes at the next depth. Each of the 2^{i+1} nodes has three child nodes for $0 \leq i \leq d_\mathrm{p} - 3$, and five child nodes for $i = d_\mathrm{p} - 2$. Below is an example for a published topic "a/b/c" where $d_\mathrm{p} = 3$:

- At depth 0, there are 2 nodes that have child nodes: "+" and "a". Each of them has three child nodes, like "+/#$", "+/+", and "+/b".
- At depth 1, there are 4 nodes that have child nodes: "+/+", "+/b", "a/+", and "a/b". Each of them has five child nodes like "+/+/#$", "+/+/+$", "+/+/c$", "+/+/+/#$", and "+/+/c/#$".

Considering the candidate-prefix tree, at depth i where $0 \leq i \leq d_\mathrm{p} - 2$, the number of candidate prefixes tested (i.e., Bloom filter queries) is 3×2^i. At depth $i = d_\mathrm{p} - 1$, the number of tested candidates is 5×2^i. Therefore, the total number of Bloom filter queries is

$$ 3 \sum_{i=0}^{d_\mathrm{p}-2} 2^i + 5 \times 2^{d_\mathrm{p}-1} = 4 \times 2^{d_\mathrm{p}} - 3. \tag{3} $$

Hence, WAP-BF requires

$$ O\left(2^{d_\mathrm{p}}\right) $$

search time in the worst case.

As indicated above, all methods exhibit exponential complexity in the worst case. However, the likelihood of encountering the worst case differs among the methods. For WAP-BF, exhaustive traversal of the candidate-prefix tree without early termination is unlikely to occur in practice. In contrast, existing methods are more likely to reach the worst case: when no subscription matches, they must exhaustively test all candidate patterns. We further demonstrate this tendency in Section V-B.

Beyond worst-case query counts, WAP-BF also has an effective advantage over BF-Enum and BFP in the rate at which Bloom filter false positives translate into incorrect match decisions. In BF-Enum and BFP, a single false positive on any tested pattern causes the published topic to be reported as matching some local subscription when none exists, so, under the standard Bloom filter independence assumption, the effective misjudgment rate is $1 - (1 - p_\mathrm{f})^q$, where q denotes the number of Bloom filter queries issued per PUBLISH. In WAP-BF, by contrast, the algorithm reports a positive match only when its depth-first traversal finds a terminal candidate prefix π^* (ending with \$) whose membership test returns true, while every ancestor prefix on the path from the root to π^* has also previously tested as a member; otherwise the traversal would have pruned that branch. Two sub-cases arise. (a) π^* may be a multi-level-wildcard terminal $\pi \diamond \#\$ \in \mathrm{cand}_i(\pi, t_p)$ for some shallow level $0 \leq i \leq d_\mathrm{p} - 2$. A false match of this shape can be triggered by a single false positive on π^* at level i, provided that the ancestor path is alive; keeping the ancestor path alive in turn requires that each ancestor prefix has tested positive, which is most likely when those ancestors correspond to prefixes genuinely inserted by some local subscription. (b) π^* may be a deepest-level terminal in $\mathrm{cand}_{d_\mathrm{p}-1}(\pi, t_p)$; here a false match requires a false positive at depth $d_\mathrm{p} - 1$ together with either (i) false positives at all ancestor levels along the path, with probability $O(p_\mathrm{f}^{d_\mathrm{p}})$ under standard Bloom filter independence assumptions, or (ii) a chain of ancestor prefixes that genuinely correspond to prefixes inserted by some subscription. Both sub-cases require, in addition to the false positive at π^*, that the ancestor path be alive—a structural precondition that does not hold for an arbitrary published topic and whose prevalence is bounded by the broker's subscription coverage. By contrast, in BF-Enum and BFP a false positive on any tested pattern causes a misjudgment directly. Hence, for

the same per-query false-positive rate p_{f}, WAP-BF's effective misjudgment rate tends to be lower than that of BF-Enum and BFP. The same observation also explains why a relatively loose p_{f} suffices for WAP-BF to deliver high-fidelity routing decisions in practice. The multi-broker forwarding experiment shown in Figure 10 in Section VI empirically corroborates this tendency.

E. Correctness argument

Using the formalization in Sections II-A, IV-A, and IV-C, we now state two properties of the lookup procedure (Algorithm 1) with respect to the matching relation match. The Bloom filter used by a broker is built from the set $\bigcup_{\sigma \in S} \mathrm{prefixes}(\sigma)$, where S is the broker's local subscription set. Recall that a Bloom filter admits false positives but no false negatives on inserted elements.

Lemma 1. For every $\sigma \in S$ and every $\pi \in \mathrm{prefixes}(\sigma)$, the membership test CHECK_BF(π) returns true. This follows immediately from the definition of a Bloom filter: any inserted element is reported as present.

Proposition 1 (Soundness). If Algorithm 1 returns true for a published topic t_p, then either (i) there exists $\sigma \in S$ with $\mathrm{match}(\sigma, t_p) = 1$, or (ii) at least one Bloom filter membership test performed by the algorithm returned a false positive.

Proof sketch. Algorithm 1 returns true only when it finds a terminal candidate π^* (ending with \$) in $\mathrm{cand}_i(\pi, t_p)$ at some level $0 \le i \le d_{\mathrm{p}} - 1$ for a parent prefix π reached by the depth-first traversal, and CHECK_BF(π^*) is true. By construction of cand_i, π^* falls into one of two cases. (A) For some $0 \le i \le d_{\mathrm{p}} - 2$, $\pi^* = \pi \diamond \#\$$ with $\pi = m_0/\cdots/m_{i-1}$ (or $\pi = \varepsilon$ when $i = 0$), where each $m_j \in \{l_j, +\}$ because the ancestor candidates along the traversal are drawn from the literal-extension or single-level-wildcard branch of cand_j. This shape coincides with the deepest element of $\mathrm{prefixes}(\sigma)$ for the filter $\sigma = m_0/\cdots/m_{i-1}/\#$ (or $\sigma = \#$ when $i = 0$), and such a σ satisfies $\mathrm{match}(\sigma, t_p) = 1$ via the multi-level-wildcard clauses of match (the trailing $\#$ absorbs levels $i, \ldots, d_{\mathrm{p}} - 1$ of t_p). (B) For $i = d_{\mathrm{p}} - 1$, $\pi^* \in \mathrm{cand}_{d_{\mathrm{p}}-1}(\pi, t_p)$ has one of the five shapes $\pi \diamond \#\$$, $\pi \diamond +\$$, $\pi \diamond l_{d_{\mathrm{p}}-1}\$$, $\pi \diamond +/\#\$$, or $\pi \diamond l_{d_{\mathrm{p}}-1}/\#\$$. By construction these correspond, respectively, to filters ending with $\#$, ending with $+$ at depth $d_{\mathrm{p}} - 1$, fully-specified filters, and the parent-inclusion clauses (vi) and (vii) of match. In either case, if π^* was truly inserted, the corresponding σ exists in S and $\mathrm{match}(\sigma, t_p) = 1$. Otherwise, $\pi^* \notin \bigcup_{\sigma \in S} \mathrm{prefixes}(\sigma)$ and CHECK_BF(π^*) returned a false positive. □

Proposition 2 (Completeness). If there exists $\sigma \in S$ with $\mathrm{match}(\sigma, t_p) = 1$, then Algorithm 1 returns true.

Proof sketch. Fix $\sigma \in S$ with $\mathrm{match}(\sigma, t_p) = 1$. We exhibit a terminal prefix $\pi^* \in \mathrm{prefixes}(\sigma)$ that is reached by Algorithm 1's depth-first traversal of the candidate-prefix tree of t_p. By Lemma 1, every prefix in $\mathrm{prefixes}(\sigma)$ tests positive under CHECK_BF, so once π^* is shown to appear in cand_i at the appropriate level i along a path whose every ancestor also lies in $\mathrm{prefixes}(\sigma)$, the traversal cannot prune that branch and returns true upon finding π^*, which ends with \$. We split on whether σ terminates with a multi-level wildcard "#" strictly before depth d_{p} or not, which mirrors the case split in the Soundness proof. (A) Suppose $\sigma = m_0/\cdots/m_{k-1}/\#$ with $0 \le k \le d_{\mathrm{p}} - 1$ (where $k = 0$ means $\sigma = \#$, and the trailing "#" absorbs levels $k, \ldots, d_{\mathrm{p}} - 1$ of t_p). By match, $m_i = l_i$ or $m_i = +$ for $0 \le i \le k - 1$. Set $\pi_{i+1} = m_0/\cdots/m_i$ for $0 \le i \le k - 1$ (with $\pi_0 = \varepsilon$), and $\pi^* = \pi_k \diamond \#\$$. Each π_{i+1} lies in cand_i via the literal or single-level-wildcard branch and also in $\mathrm{prefixes}(\sigma)$, and π^* lies in cand_k via the "#\$" branch (available in cand_i for every $0 \le i \le d_{\mathrm{p}} - 1$) with $\pi^* \in \mathrm{prefixes}(\sigma)$. (B) Otherwise $\sigma = m_0/\cdots/m_{d_{\mathrm{p}}-1}$ with each $m_i \in L \cup \{+\}$, or $\sigma = m_0/\cdots/m_{d_{\mathrm{p}}-1}/\#$ matching t_p via the parent-inclusion clauses (vi) and (vii) of match. By match, $m_i = l_i$ or $m_i = +$ for $0 \le i \le d_{\mathrm{p}} - 1$. Set $\pi_{i+1} = m_0/\cdots/m_i$ for $0 \le i \le d_{\mathrm{p}} - 2$ as in case (A), and $\pi^* = m_0/\cdots/m_{d_{\mathrm{p}}-1}\$$ or $\pi^* = m_0/\cdots/m_{d_{\mathrm{p}}-1}/\#\$$, respectively. At the deepest level, π^* is the $\pi \diamond +\$$, $\pi \diamond l_{d_{\mathrm{p}}-1}\$$, $\pi \diamond +/\#\$$, or $\pi \diamond l_{d_{\mathrm{p}}-1}/\#\$$ branch of $\mathrm{cand}_{d_{\mathrm{p}}-1}$, which exists to capture the fully-specified case as well as clauses (vi) and (vii); $\pi^* \in \mathrm{prefixes}(\sigma)$ as well. In either case, every ancestor on the path to π^* lies in $\mathrm{prefixes}(\sigma) \subseteq \bigcup_{\sigma' \in S} \mathrm{prefixes}(\sigma')$, so by Lemma 1 each ancestor's CHECK_BF returns true and the depth-first traversal proceeds without pruning until it finds the terminal π^*. Algorithm 1 therefore returns true. □

Proposition 2 establishes that WAP-BF never drops a message whose published topic matches some local subscription; Proposition 1 shows that spurious positive decisions are confined to the false-positive behavior of the underlying Bloom filter, which is governed by the parameters b, k, and p_{f} of Section II-B.

V. Evaluation

In this section, we evaluate (i) the amount of data required to share topic information among brokers and (ii) the effect of Bloom filter lookup cost on PUBLISH-processing performance. The first metric is important because Bloom filters are used to reduce the communication and memory overhead of sharing and managing topic information. The second is also important because PUBLISH messages generally occur much more frequently than SUBSCRIBE messages.

Broker implementation and configuration can affect measured PUBLISH-processing performance. To isolate the algorithmic differences among the methods, we therefore use a two-step evaluation. First, we compare the number of Bloom filter queries issued by WAP-BF, BF-Enum, and BFP. Second, using the same Bloom filter implementation, we measure how the number of queries affects PUBLISH-forwarding throughput and latency by executing a configured number of Bloom filter lookups between message reception and forwarding. This design separates the effect of query count from other implementation- and configuration-dependent factors.

Section V-A analytically evaluates communication and memory overhead, Section V-B presents the query-count results, and Section V-C evaluates the effect of Bloom filter query count on PUBLISH-forwarding performance.

A. Communication and memory overhead

This section analytically evaluates the amount of data required to share topic information among brokers. We compare

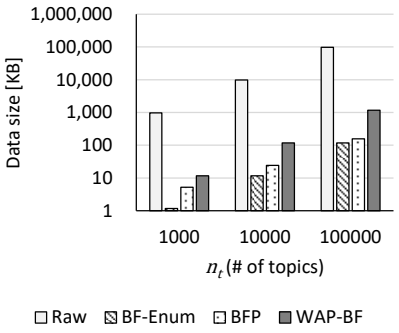

Fig. 4. Data size vs. number of topics

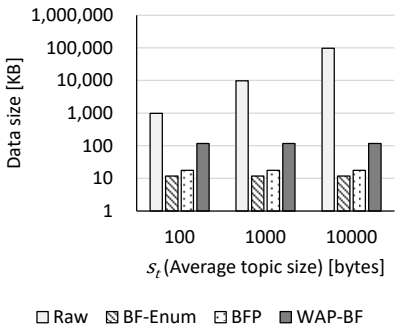

Fig. 5. Data size vs. average topic size

WAP-BF with "Raw", which handles topics as they are. BF-Enum and BFP described in Section IV-D are also comparison targets.

We consider two parameters: s_{t} and n_{t}. s_{t} is the size of each topic in bytes, whose default value is $1{,}000$. n_{t} is the number of topics managed by a broker, with a default value of $10{,}000$. These default values are practical in, for example, the use case described in Section II-C. We also assume constants $d_{\mathrm{s}} = 10$ and $p_{\mathrm{f}} = 0.01$. d_{s} is the number of topic levels in a subscription topic, assuming all subscription topics have the same number for simplicity. p_{f} is the allowable false-positive rate in using Bloom filters.

We can calculate the Bloom filter size from Equation (1) shown in Section II-B. The data size of WAP-BF can be calculated as

$$\left(-\frac{\ln p_{\mathrm{f}}}{(\ln 2)^2} n_{\mathrm{t}} d_{\mathrm{s}}\right) \frac{1}{8}$$

in bytes. That of Raw is

$$s_{\mathrm{t}} n_{\mathrm{t}}.$$

The data size of BF-Enum is

$$\left(-\frac{\ln p_{\mathrm{f}}}{(\ln 2)^2} n_{\mathrm{t}}\right) \frac{1}{8}.$$

Regarding BFP, it shares wildcard topic patterns with other brokers in addition to a Bloom filter. We assume that the pattern occurrence frequency distribution follows a Zipf distribution [35]. It is empirically known that Zipf's law corresponds to Heaps' law [36], [37]. Under Heaps' law, the vocabulary size D as a function of the total number of tokens N is given by:

$$D(N) = KN^{\beta},$$

where K and β are constants. If we include the pattern of "no wildcard," the topics can be regarded as a set of wildcard patterns. By associating the number of topics n_{t} with the total number of tokens N, and the number of wildcard patterns n_{w} with the vocabulary size D, n_{w} can be obtained as:

$$n_{\mathrm{w}} = \min\left(K n_{\mathrm{t}}^{\beta} - 1,\ 3 \cdot 2^{d_{\mathrm{s}}} - 2\right).$$

The cap $3 \cdot 2^{d_{\mathrm{s}}} - 2$ corresponds to the total number of distinct wildcard patterns (treating each level as literal or "+", with an optional terminating "#") over d_{s} topic levels, beyond which the Heaps-law extrapolation would exceed the universe of representable patterns; for $d_{\mathrm{s}} = 10$, this cap equals $3{,}070$. In this analytical evaluation, we assume $K = 65$ and $\beta = 0.5$ as general values [38]. The data size of BFP can be calculated as

$$\left(-\frac{\ln p_{\mathrm{f}}}{(\ln 2)^2} n_{\mathrm{t}}\right) \frac{1}{8} + s_{\mathrm{w}} n_{\mathrm{w}},$$

where s_{w} is the average size of wildcard patterns. We assume $s_{\mathrm{w}} = 2$ bytes. As described in Section IV-D, a pattern of single-level wildcards can be represented using a d_{s}-bit string, and that of multi-level wildcard requires a bit string of length

$$\lceil \log_2(d_{\mathrm{s}} + 1) \rceil.$$

Therefore, assuming that $d_{\mathrm{s}} = 10$, we need a minimum of 14 bits. Taking this into account, we assume that s_{w} is 2 bytes. While the above BFP calculation relies on simplifying assumptions, it is intended only to provide intuition and illustrate overall trends.

TABLE IV
ANALYTICAL BLOOM FILTER SIZE (BYTES) FOR WAP-BF.

$p_{\mathrm{f}} \backslash n_{\mathrm{t}}$	10^3	10^4	10^5
10^{-2}	11,981	119,813	1,198,132
10^{-3}	17,972	179,720	1,797,198
10^{-4}	23,963	239,626	2,396,265

Figures 4 and 5 show the difference in data size among the methods, and Table IV additionally tabulates the analytical Bloom filter size of WAP-BF at $d_{\mathrm{s}} = 10$, across target false-positive rates p_{f} and subscription-topic counts n_{t}, with values rounded to the nearest byte, to provide a finer-grained view of the memory cost. As mentioned in Section IV-D, WAP-BF requires a larger Bloom filter than BF-Enum and BFP since it adds prefixes rather than topics. On the other hand, it is significantly smaller than Raw. Quantitatively, the Bloom filter size of WAP-BF scales linearly in $\log(1/p_{\mathrm{f}})$ and in n_{t}, and carries an additional factor of the subscription-topic depth d_{s} relative to BF-Enum and BFP because it inserts prefixes rather than full topics; the size of Raw, which does not use a Bloom filter, depends instead on $s_{\mathrm{t}} n_{\mathrm{t}}$. Even at the tightest false-positive budget we consider ($p_{\mathrm{f}} = 10^{-4}$) and the largest subscription count ($n_{\mathrm{t}} = 10^5$), WAP-BF remains substantially smaller than the Raw baseline at $s_{\mathrm{t}} = 1{,}000$ bytes, which confirms that the prefix-insertion overhead does not compromise the compactness that motivates the use of a Bloom filter. In this analytical evaluation, we simply use the

TABLE V
SIMULATION PARAMETERS

Parameter	Default value
Number of topics n_{t}	6,000
Number of topic levels $d_{\mathrm{s}} = d_{\mathrm{p}}$	10
Wildcard probability p_{w}	0.25
Minimum wildcard level i_{w}	1
Bloom filter false-positive rate p_{f}	0.001

product of n_{t} and d_{s} to calculate the number of elements stored in the Bloom filter. However, the number could be smaller than that in practice, because many topics share a common prefix at levels closer to the root of the topic namespace.

Smaller data sizes are preferable to reduce communication traffic and memory usage. Particularly, if a broker communicates with multiple other brokers, the volume of topic information increases according to the number of neighboring brokers. WAP-BF can significantly reduce data size compared to Raw.

The analytical comparison above rests on a small set of explicit assumptions, which we list here so that the scope of the conclusions is transparent. (a) For simplicity, every subscription topic has depth d_{s}, which can be regarded as approximating the depth distribution of real MQTT namespaces by a representative value such as its mean. (b) Distinct subscription filters give rise to distinct prefixes, which is the worst case for the number of Bloom filter elements inserted by WAP-BF; in practice, shared semantic prefixes in real deployments reduce the element count, making WAP-BF more compact than the model suggests. (c) Bloom filter false positives at distinct levels are independent, which is the standard assumption used when dimensioning Bloom filters. (d) The analytical model is used only as an orientation device. The simulation study of Section V-B and the implementation-level experiments of Sections V-C and VI complement this model by evaluating WAP-BF under generated workloads and actual broker execution, but they do not eliminate the need for future validation using real MQTT traces and broader deployment conditions.

B. Number of queries

To clarify the average-case search behavior of WAP-BF discussed in Section IV-D, we conducted simulation experiments using a Java implementation. As an evaluation metric, we count the number of Bloom filter queries required to determine whether a published topic has any corresponding subscription topics. We compare WAP-BF against BF-Enum and BFP, as described in Section IV-D.

The simulation parameters are summarized in Table V.

Among these parameters, we varied n_{t}, $d_{\mathrm{s}} = d_{\mathrm{p}}$, p_{w}, and i_{w} one at a time while keeping the other parameters, including p_{f}, fixed at the default settings of Table V. n_{t} denotes the number of topics managed by a broker. The n_{t} subscription topics are generated independently and may include duplicates, which reflects the realistic situation in which multiple clients of a broker subscribe to the same topic filter; the Bloom filter representation and the per-PUBLISH query counts of WAP-BF, BF-Enum, and BFP are unaffected by such multiplicity, so n_{t} is reported as the total subscription count rather than the number of distinct filters. Each topic level is either a token drawn uniformly at random from a shared vocabulary of 10 tokens or a wildcard; the vocabulary is fixed once at the start of each trial and reused across all subscription and published topics, so that subscription topics and published topics share the same token alphabet and prefix structure. For ease of analysis, we set the published topic and the underlying literal topic from which each subscription topic is generated to have the same number of levels in the experiments, i.e., $d_{\mathrm{s}} = d_{\mathrm{p}}$. However, a subscription topic ending with the multi-level wildcard "#" truncates its suffix at some level $i \in [i_{\mathrm{w}}, d_{\mathrm{s}} - 1]$, so its actual number of levels can be strictly less than d_{p}.

p_{w} represents the probability that a topic among the n_{t} topics contains wildcards. For example, when $p_{\mathrm{w}} = 0.5$, we expect approximately $n_{\mathrm{t}}/2$ topics to include wildcards. Each wildcard topic is generated as follows. First, we randomly generate a topic without wildcards. Second, we enumerate all wildcard variants derived from it, considering the i_{w} constraint below. Finally, we select one variant uniformly at random.

i_{w} denotes the minimum level at which a wildcard may appear. For instance, if $i_{\mathrm{w}} = 0$, wildcards may appear at any level. The single-level wildcard "+" is admissible at any level $i \in [i_{\mathrm{w}}, d_{\mathrm{s}} - 1]$, and the multi-level wildcard "#" is admissible at any terminating level $i \in [i_{\mathrm{w}}, d_{\mathrm{s}} - 1]$; in particular, "#" is not restricted to the deepest level $d_{\mathrm{s}} - 1$, so a subscription such as $l_0/\#$ with depth strictly less than d_{s} is generated whenever the enumeration step selects it.

We ran experiments under the following three configurations for subscription topics:

- Allow wildcard-only topics (e.g., "#" and "+/#").
- Disallow wildcard-only topics.
- Disallow wildcard-only topics and also exclude topics that would match the published topic.

Wildcard-only topics can match a large number of published topics. In particular, "#" matches every topic and may induce substantial traffic. Therefore, in large-scale deployments, clients may refrain from using such wildcard-only subscriptions. Moreover, due to the loose-coupling nature of the publish-subscribe model, a PUBLISH message does not necessarily have corresponding subscriptions. We consider the above three configurations to highlight how the methods behave under these different conditions.

The published topic is generated with d_{p} topic levels, where each level is a token drawn uniformly at random from the same 10-token vocabulary used for subscription topics.

Figures 6–8 present the simulation results. For each configuration, we ran the simulation five times with independent random seeds; bars report the mean number of Bloom filter queries per PUBLISH and error bars show one sample standard deviation across the five trials, computed with Bessel's correction.

Overall, WAP-BF consistently requires fewer queries than BF-Enum and BFP at the default depth $d_{\mathrm{s}} = d_{\mathrm{p}} = 10$ across all three configurations. For example, at the default operating point of Table V in Figure 8(b), WAP-BF needs only 57.4 ± 6.4 queries on average, whereas BF-Enum needs $1{,}658.0 \pm 952.2$ and BFP needs 593.6 ± 200.0. The gap widens with d_{s}: in the

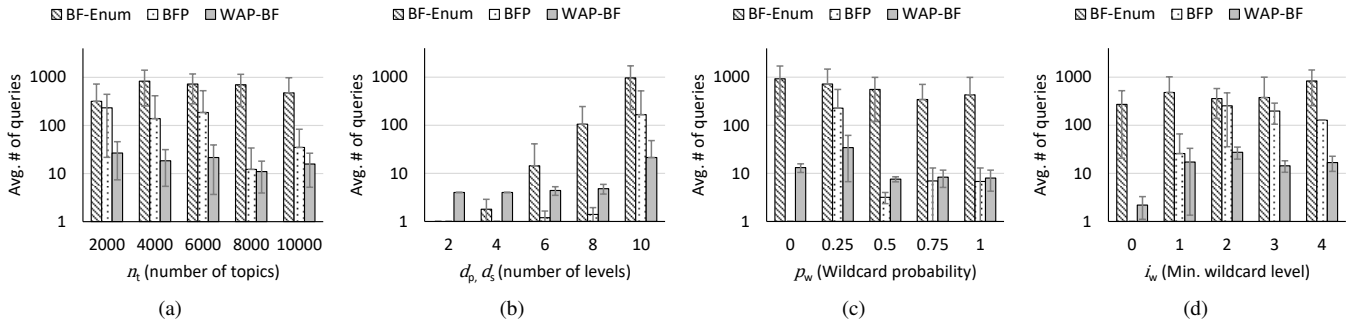

Fig. 6. Average number of queries (with wildcard-only topics)

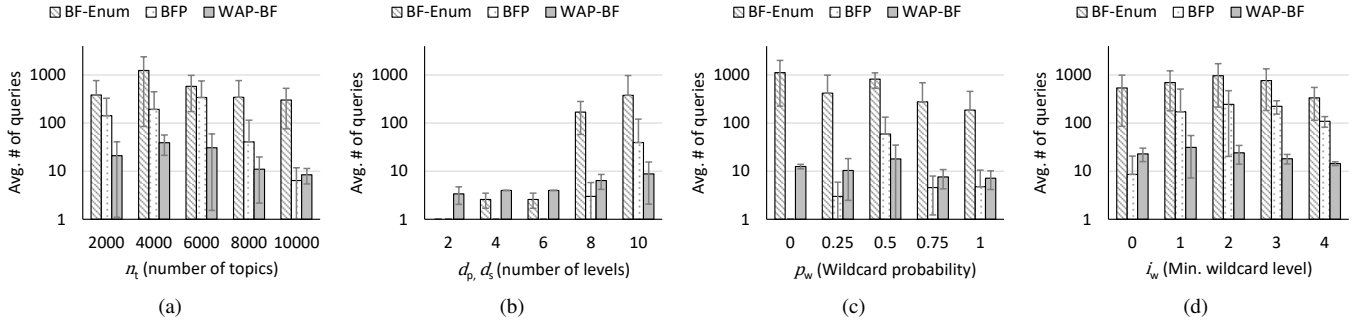

Fig. 7. Average number of queries (without wildcard-only topics)

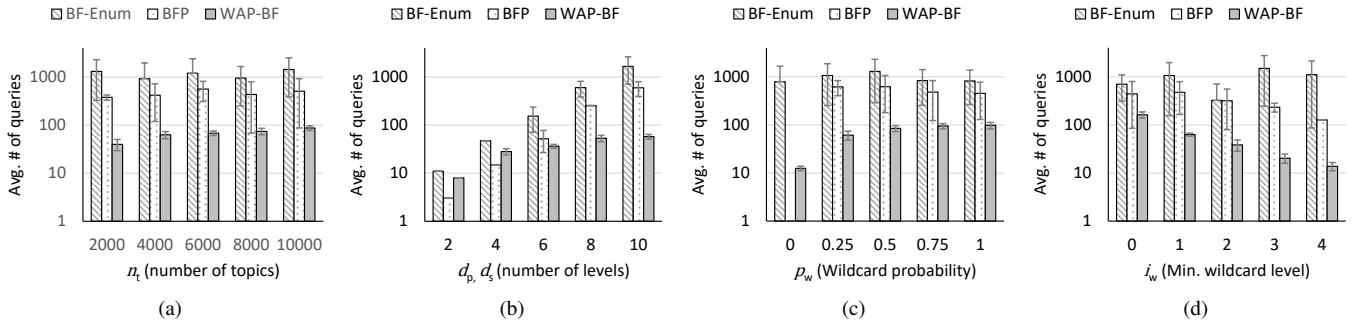

Fig. 8. Average number of queries (without matching subscriptions)

same panel, the WAP-BF curve grows roughly linearly with depth, while BF-Enum grows exponentially in d_{p}.

In Figures 6 and 7, BFP attains 1.0 queries on average at $p_{\mathrm{w}} = 0$ in the p_{w}-axis panels (Figures 6(c) and 7(c)) and at $i_{\mathrm{w}} = 0$ in the i_{w}-axis panel of the with-wildcard-only configuration (Figure 6(d)). The latter happens because, when $i_{\mathrm{w}} = 0$, the wildcard-only filter "#" is admissible and is reliably included in the subscription set, so BFP tests it first and terminates immediately. The former happens because, when $p_{\mathrm{w}} = 0$, no wildcard patterns exist and BFP therefore has only the literal candidate to test.

For BFP, increasing i_{w} at the default $d_{\mathrm{s}} = 10$ generally reduces the number of usable wildcard patterns and therefore the average number of queries, although the reduction is non-monotone in finite samples (Figures 6(d) and 7(d)) because the random selection of five trials interacts with the random presence of short, near-universal wildcard filters.

Across Figures 7(a), 7(c), and 7(d), BF-Enum stays in the hundreds-to-thousands range with no systematic dependence on n_{t}, p_{w}, or i_{w}. This is because BF-Enum tests every wildcard variant derived from the published topic, whose count depends only on d_{p} (cf. Equation (2) in Section IV-D), and is therefore directly affected only by the d_{p}-axis panel of Figure 7(b). BFP, in contrast, is mainly influenced by the number of distinct wildcard patterns present in subscription topics: increasing d_{s} tends to increase the number of patterns, while increasing i_{w} tends to decrease them.

Figure 8 reports the cases where the published topic has no matching subscriptions. In this scenario, both BF-Enum and BFP must exhaustively test a large set of candidate patterns and the gap to WAP-BF is largest: BF-Enum exhaustively queries every wildcard pattern derived from the published topic, BFP queries several hundred shared patterns, and WAP-BF remains below 100 queries on average across the four

sweep panels. WAP-BF achieves this because its level-wise Bloom filter tests prune candidate prefixes as soon as no positive evidence is returned.

Given that Bloom filter lookups are performed for every PUBLISH message, the per-PUBLISH query-count reduction by WAP-BF reported above can translate into measurable broker-level performance gains. We investigate this effect in the next section.

The synthetic workloads used above are appropriate for a controlled analysis but are uniform in ways that real MQTT deployments typically are not. Real subscription topic namespaces exhibit several structural properties; we discuss them in two groups. The first group is properties that are at least partially favorable to WAP-BF: (i) semantic hierarchical prefixes shared across many topics (e.g., site/zone/device identifiers) reduce the number of distinct prefixes inserted into WAP-BF and therefore lower the effective element count of its Bloom filter; and (ii) diverse topic depths cause the candidate-prefix tree to be unbalanced in a way that favors early termination of the depth-first lookup. The second group is properties that are also present in real MQTT deployments but whose effect on the per-PUBLISH query count of WAP-BF and its baselines depends on the actual usage of the topic namespace: (iii) skewed popularity of published topics; and (iv) non-uniform placement of wildcards within subscription filters. A quantitative evaluation on real MQTT traces is left as future work.

The worst-case search cost of WAP-BF is $O(2^{d_{\mathrm{p}}})$, reached when every level-wise Bloom filter membership test needed to keep a candidate branch alive returns positive and the depth-first traversal consequently visits the entire candidate-prefix tree. The structural origin of this bound is the MQTT wildcard alphabet: at each non-terminal topic level, the traversal may continue along both the literal branch and the single-level wildcard "+" branch, while terminal candidates involving "#" are also tested. In ordinary no-match cases, however, Algorithm 1 prunes a branch as soon as a level-wise Bloom filter membership test returns negative. Therefore, a non-existing branch can survive to deeper levels only when the corresponding prefix tests are false positives, or when existing subscriptions share enough prefixes with that branch. The likelihood of such survival is governed by the Bloom filter false-positive rate p_{f} and by the prefix structure of the subscription set. Thus, the exponential bound should be interpreted as a conservative worst-case bound, not as an expected-case guarantee. The query-count measurements above provide empirical evidence that, across the parameter regimes we tested, the observed per-PUBLISH query count for WAP-BF remains substantially below those of BF-Enum and BFP, suggesting that this worst case is rarely reached under realistic Bloom filter dimensioning. From a deployment perspective, the risk of worst-case behavior can be reduced by placing a hard cap on topic depth and by choosing a smaller p_{f}, at the cost of a larger Bloom filter.

The query-count gains reported above arise from the interaction of three algorithmic choices inside the WAP-BF lookup procedure: (i) early pruning by the level-wise Bloom filter membership test, which discards a candidate prefix as soon as the test returns negative (Algorithm 1); (ii) the candidate order at each level, with the multi-level-wildcard-first order placing the multi-level-wildcard candidate "#" before the single-level-wildcard candidate "+" and the literal candidate, so that whichever near-universal wildcard subscription is most likely to match the published topic is tested first (Section IV-C); and (iii) the traversal strategy over the candidate-prefix tree, with depth-first traversal letting a positive test at any level immediately descend toward a terminal-level early-exit. We isolate the contribution of each choice through an ablation study that varies the three choices independently, yielding eight variants of the form `Pruning-Order-Traversal`, where `Pruning` $\in$ {`EP`, `NoEP`}, `Order` $\in$ {`MWF`, `MWL`}, and `Traversal` $\in$ {`DFS`, `BFS`}. Here, `EP` (early pruning) discards a candidate prefix whose level-wise membership test returns negative, and `NoEP` continues without such pruning; `MWF` (multi-level-wildcard first) tests the candidate "#" before "+" and the literal candidate at every level, whereas `MWL` (multi-level-wildcard last) reverses this order; and `DFS` and `BFS` denote depth-first and breadth-first traversal of the candidate-prefix tree, respectively. The variant `EP-MWF-DFS` is the original WAP-BF described in Section IV-C.

TABLE VI
ABLATION STUDY OF WAP-BF LOOKUP CHOICES.

Variant (Pruning / Order / Traversal)	With wildcard-only topics	Without wildcard-only topics	Without matching subscriptions
`EP-MWF-DFS` (original)	10.40 ± 5.90	27.60 ± 29.75	62.80 ± 8.32
`EP-MWF-BFS`	25.60 ± 24.77	30.60 ± 28.25	62.80 ± 8.32
`EP-MWL-DFS`	21.40 ± 15.13	30.80 ± 26.74	62.80 ± 8.32
`EP-MWL-BFS`	27.20 ± 24.00	31.80 ± 27.22	62.80 ± 8.32
`NoEP-MWF-DFS`	$2{,}055.40 \pm 5.90$	$2{,}867.80 \pm 1{,}118.45$	$4{,}093.00 \pm 0.00$
`NoEP-MWF-BFS`	$604.40 \pm 1{,}236.32$	$1{,}654.00 \pm 2{,}226.64$	$4{,}093.00 \pm 0.00$
`NoEP-MWL-DFS`	$2{,}306.60 \pm 430.40$	$2{,}881.40 \pm 1{,}106.08$	$4{,}093.00 \pm 0.00$
`NoEP-MWL-BFS`	$606.00 \pm 1{,}235.42$	$1{,}655.20 \pm 2{,}225.55$	$4{,}093.00 \pm 0.00$

EP: early pruning; NoEP: no early pruning; MWF: multi-level-wildcard first; MWL: multi-level-wildcard last; DFS: depth-first search; BFS: breadth-first search.

Table VI reports the mean number of Bloom filter queries per PUBLISH for the eight ablation variants at the default operating point of Table V ($n_{\mathrm{t}} = 6{,}000$, $d_{\mathrm{s}} = d_{\mathrm{p}} = 10$, $p_{\mathrm{w}} = 0.25$, $i_{\mathrm{w}} = 1$, $p_{\mathrm{f}} = 0.001$), averaged over five independent runs with different random seeds and summarized as mean $\pm$ sample standard deviation (with Bessel's correction) under the same three subscription configurations as Figures 6–8.

To isolate the contribution of each algorithmic choice, we compare `EP-MWF-DFS` with each of the three single-axis ablations `EP-MWF-BFS`, `EP-MWL-DFS`, and `NoEP-MWF-DFS`. Early pruning (`EP-MWF-DFS` vs. `NoEP-MWF-DFS`) is by far the dominant factor: disabling it inflates the per-PUBLISH query count by roughly two orders of magnitude in every configuration; in the no-match configuration the no-pruning variant exhausts the entire candidate-prefix tree, whose size $4 \cdot 2^{d_{\mathrm{p}}} - 3$ is given by Equation (3) in Section IV-D, on every PUBLISH, which establishes that early pruning is the mechanism that lets WAP-BF avoid the worst case. Multi-level-wildcard-first versus multi-level-wildcard-last ordering (`EP-MWF-DFS` vs. `EP-MWL-DFS`) is a secondary factor that matters most when wildcard-only filters are present and short: it cuts the with-wildcard-only mean from 21.40 to 10.40 (about $2\times$), yields a much smaller reduction in the mean query count, from 30.80 to 27.60, without wildcard-only topics,

and contributes nothing in the no-match configuration, where the two orderings reach identical 62.80 means because of the exhaustive exploration of the subtree reachable through prefixes whose Bloom filter tests return positive. Candidate ordering only permutes the order in which this positive-prefix-reachable subtree is explored, and therefore does not change the total number of Bloom filter queries. Depth-first versus breadth-first traversal (`EP-MWF-DFS` vs. `EP-MWF-BFS`) is also a secondary factor: relative to DFS, BFS increases the mean query count from 10.40 to 25.60 with wildcard-only topics and slightly increases it from 27.60 to 30.60 without wildcard-only topics; under the no-match configuration, the two traversal strategies tie at 62.80 for the same reason.

Considering the remaining variants jointly, two further observations are noteworthy. First, when early pruning is disabled, the traversal strategy becomes the dominant axis among the remaining choices: the BFS variants `NoEP-MWF-BFS` and `NoEP-MWL-BFS` have mean query counts lower by factors of approximately 3.4–3.8 than their DFS counterparts in the with- and without-wildcard-only configurations (604.40 vs. 2,055.40, and 1,654.00 vs. 2,867.80). The reason is that DFS always traverses each branch down to its terminal level even when the branch does not match the published topic, whereas BFS can terminate as soon as it tests a multi-level wildcard "#" at a shallow level, so BFS keeps the query count smaller. Second, the no-match configuration collapses every NoEP variant to the same constant $4{,}093 \pm 0$, which equals the size $4 \cdot 2^{d_{\mathrm{p}}} - 3$ of the candidate-prefix tree given by Equation (3) in Section IV-D at $d_{\mathrm{p}} = 10$; this confirms that, in the worst case discussed earlier in this section, the query count of any non-pruning lookup is determined entirely by this structural bound, and is independent of the candidate order or traversal strategy.

C. Influence on broker performance

In this section, we describe an experiment designed to assess how the number of Bloom filter queries affects broker performance. We evaluate two metrics: average egress throughput and average latency. The former is the mean number of messages per second that the broker delivers to subscribers, and the latter is the mean time from a publisher to a subscriber. We use Mochi-MQTT Server v2.7.9 [39] as the broker and MQTTLoader v0.8.6 [40], [41] as the client workload generator. We modified the broker so that Bloom filter lookups are executed after a PUBLISH message is received and before it is forwarded to subscribers.

For the Bloom filter, we use the same Go implementation as in Section VI, backed by the 128-bit MurmurHash3 hash function via spaolacci/murmur3 [42] with Kirsch–Mitzenmacher double-hashing. We configure the Bloom filter with an expected insertion count of 10,000 and a target false-positive rate of 0.001. Before measuring throughput and latency, we prepopulate the filter with 10,000 elements, where each element is a random string of length 100. Upon receiving each PUBLISH message, the broker performs the specified number of Bloom filter queries. For each query-count setting, we report the mean of three independent trials.

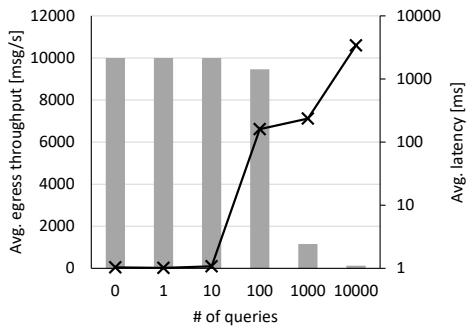

Fig. 9. Effect of Bloom filter lookups on broker performance

The parameters of MQTTLoader are set as follows:

- MQTT protocol version: v5.0
- The number of publishers: 1
- The number of subscribers: 1
- Ramp-up time: 5 seconds
- Ramp-down time: 5 seconds
- Execution time: 70 seconds (i.e., 60 seconds excluding ramp-up/ramp-down times)
- Payload size: 1,024 bytes
- Publish interval: 100 microseconds (i.e., ingress throughput is 10,000 messages per second)

We used three separate host machines for the publisher, subscriber, and broker. Each host is equipped with an Intel Core i9-12900 processor, 64 GB of memory, a 1 Gb/s Ethernet interface, and runs Ubuntu 24.04.

Figure 9 summarizes the results. As the number of Bloom filter queries increases, the egress throughput decreases, while latency increases. Concretely, the broker sustains the ingress rate of 10,000 msg/s for 0 to 10 queries per PUBLISH, where the per-PUBLISH Bloom filter cost is too small to be visible above the baseline (1.01–1.07 ms average end-to-end latency). At 100 queries per PUBLISH, egress throughput falls to 9,459 msg/s and average latency rises to 160 ms. At 1,000 and 10,000 queries per PUBLISH the broker is fully Bloom filter-bound, with egress throughput falling further to 1,157 msg/s and 120 msg/s and average latency rising to 236 ms and 3,400 ms, respectively. In light of the simulation results in Figures 6–8, WAP-BF is expected to outperform existing methods in scenarios where the average number of Bloom filter queries exceeds 100 or even 1,000.

VI. Multi-broker forwarding evaluation

The evaluation in Section V isolates the per-broker cost of Bloom filter lookups. This section complements it with a minimal multi-broker forwarding experiment that exercises four methods (WAP-BF, Naive, BF-Enum, BFP) end-to-end across three physical brokers in a full-mesh, one-hop configuration, quantifying inter-broker forwarding behavior and end-to-end publisher-to-subscriber latency. The "Naive" method here forwards every received PUBLISH message to all neighboring brokers without any subscription-aware filtering.

A. Setup

We deploy three brokers on three separate host machines connected by a 1 Gb/s Ethernet network, forming a full-mesh topology with single-hop forwarding. Each host uses the same hardware specification as the broker host in Section V-C: an Intel Core i9-12900 processor, 64 GB of memory, a 1 Gb/s Ethernet interface, and Ubuntu 24.04. Each broker is a Go program built on top of Mochi-MQTT Server v2.7.9; all four methods (WAP-BF, Naive, BF-Enum, BFP) are implemented in the same binary and selected at startup. Clients are Python 3.12 processes using paho-mqtt 2.1.0. Each broker has one publisher and one subscriber connected to it, so that the deployment has three publishers and three subscribers in total. Each publisher emits PUBLISH messages whose topics are drawn from the workload generator described in Table VII; the generator deterministically replays the same topic sequence per seed. Each publisher sends at a constant rate of 100 messages per second. All MQTT connections use protocol version 5.0, PUBLISH messages are sent at QoS 0, and each PUBLISH carries a small JavaScript Object Notation (JSON) payload (sequence number, publish timestamp, and source-broker ID; approximately 60 bytes). At startup each broker exchanges its initial topic summary (Bloom filter or subscription list, depending on method) with its neighboring brokers via a retained PUBLISH message on a dedicated control topic; this one-time exchange is excluded from the reported metrics. Accordingly, this experiment evaluates a static-summary data-plane setting: subscriptions remain fixed during each run, and no control-plane summary redistribution is triggered after startup. Dynamic subscription churn, update traffic, and temporary summary staleness are not evaluated in this experiment. The host clocks are synchronized using chrony, and the residual Network Time Protocol (NTP) offset is kept below 1 ms.

Table VII summarizes the experimental parameters. Here, p_{f} denotes the target false-positive rate of a single Bloom filter membership query, which is the same value used throughout Sections V-B and V-C. The topic hierarchy follows a 10-level structure inspired by the ISA-95 / Unified Namespace conventions adopted in industrial MQTT deployments [43], [44], in which level 0 corresponds to a Sparkplug B Group ID (a site or edge gateway), levels 1–7 identify area, line, cell, subsystem, asset class, and asset, and levels 8–9 identify metric category and channel. The actual topic depths vary with the occurrence of a multi-level wildcard. Per-level vocabulary sizes grow geometrically toward the leaves to reflect the imbalance observed in such deployments, where the top levels have only a handful of site or area identifiers and the leaf levels have hundreds of sensor identifiers. Level 0 carries a broker-affinity weighting: each broker's publishers and subscribers concentrate on a home Group ID with weight 0.9 and mix into each of the other two Group IDs with weight 0.05, modeling the Sparkplug B convention of one broker per site with limited cross-site traffic. Zipf-weighted level popularity at levels ≥ 1 and the wildcard shape probabilities are chosen so that wildcard subscriptions are non-trivial without being dominated by a matches-everything "#" pattern; the "+" and "#" wildcards are restricted to levels ≥ 2, since a subscription that scans across whole enterprises or whole sites is unrealistic for the workloads this experiment targets.

TABLE VII
PARAMETERS FOR THE MULTI-BROKER FORWARDING EXPERIMENT

Parameter	Value
Number of brokers	3
Topic levels	10
Per-level vocabulary size	$3, 5, 8, 10, 15, 5, 20, 50, 100, 200$
Level-0 broker affinity	home weight 0.9, each non-home 0.05
Level popularity (levels ≥ 1)	Zipf distribution with exponent $s = 1.0$
Subscription shape (levels ≥ 2)	$p_{\mathrm{exact}} = 0.9$, $p_{+} = 0.05$, $p_{\#} = 0.05$
Total subscriptions (target)	6,000 (2,000 per broker, pre-deduplication)
Total publications	9,000 (3,000 per publisher)
Publish rate	100 msg/s per publisher
Bloom filter false-positive rate p_{f}	0.001

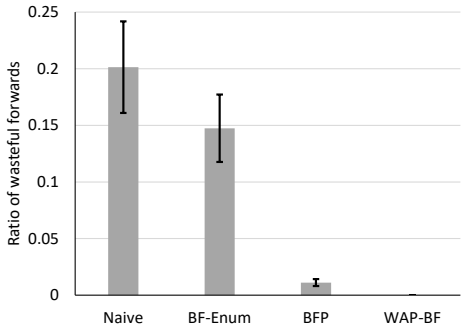

Fig. 10. Ratio of wasteful inter-broker forwards (mean ± sample standard deviation across 5 seeds).

In the subscription shape row of Table VII, each subscription filter is generated level by level: at levels 0 and 1 the token is always an exact literal (drawn from the broker's affinity-biased Group-ID distribution at level 0 and from the shared Zipf sampler at level 1); at levels ≥ 2, each level is independently an exact name with probability p_{exact}, the single-level wildcard "+" with probability p_{+}, or the multi-level wildcard "#" (which also terminates the filter) with probability $p_{\#}$. Within each broker's 2,000-subscription bucket the list is deduplicated before broker startup, so the realized per-broker count is approximately 1,600. Each configuration is repeated across five independent runs with different random seeds, for a total of 20 runs (4 methods × 5 seeds). All figures and tables below report the mean and sample standard deviation across the five per-seed metric values, computed with Bessel's correction.

B. Results

Figure 10 reports the wasteful-forwarding ratio (the number of forwards whose topic does not match any subscription held by the receiving neighboring broker, divided by the total number of forwards) for each method, with error bars showing the seed-to-seed sample standard deviation.

The Naive method, which forwards every publication to every neighboring broker, has the highest wasteful-forwarding ratio at 0.201 ± 0.040: roughly one in five forwarded publications has no matching subscription on the receiving broker. BF-Enum, whose worst-case scan of $3 \times 2^{d_{\mathrm{p}}} - 1 = 3{,}071$

TABLE VIII
END-TO-END LATENCY IN MILLISECONDS (MEAN ± SAMPLE STANDARD DEVIATION ACROSS 5 SEEDS)

Method	Mean	50th percentile	95th percentile	99th percentile
Naive	1.69 ± 0.07	1.58 ± 0.09	2.77 ± 0.22	3.50 ± 0.24
BF-Enum	9.92 ± 0.15	9.12 ± 0.21	17.98 ± 0.58	24.67 ± 1.39
BFP	1.75 ± 0.07	1.68 ± 0.05	2.74 ± 0.20	3.36 ± 0.21
WAP-BF	1.60 ± 0.10	1.55 ± 0.11	2.48 ± 0.11	3.00 ± 0.14

wildcard variants in this 10-level workload (cf. Equation (2)) encounters Bloom filter false positives often enough to forward many publications that no subscriber on the receiving broker has requested, attains 0.147 ± 0.030 – only modestly below Naive. BFP, which queries only the variants induced by the shared pattern set, reduces the ratio by more than an order of magnitude to 0.011 ± 0.003. Under the evaluated three-broker synthetic workload and Bloom filter configuration ($p_f = 0.001$), WAP-BF exhibited a wasteful-forwarding ratio of 0.000 in all five seeded runs. In other words, every message forwarded in these runs matched at least one subscription at the receiving broker. This empirical finding is specific to the workload and parameter setting considered here and should not be interpreted as implying that WAP-BF guarantees zero wasteful forwarding in other deployment scenarios. These results are consistent with the prediction made at the end of Section IV-D that, for the same per-query false-positive rate, WAP-BF's effective misjudgment rate tends to be lower than that of BF-Enum and BFP. At the implementation level, this tendency is manifested, in the evaluated setting, as a substantially lower measured proportion of wasteful inter-broker forwarding.

Table VIII reports end-to-end latency. In this experiment, WAP-BF achieved the lowest mean latency and the lowest 50th-, 95th-, and 99th-percentile latencies among the four methods, including the Bloom filter-based baselines BF-Enum and BFP. The reduction is particularly large with respect to BF-Enum. The mean latency gap between BF-Enum and WAP-BF, 8.32 ms, exceeds 80 times the WAP-BF mean-latency standard deviation (0.10 ms) and is therefore much larger than the observed seed-to-seed variation. Compared with BFP, WAP-BF had numerically lower reported latency values than BFP for the mean and all percentiles, although the margins were small. This ordering is consistent with the per-PUBLISH Bloom filter query-count differences reported in Section V-B: in this 10-level workload BF-Enum performs up to $3 \times 2^{d_{\mathrm{p}}} - 1 = 3{,}071$ Bloom filter queries per forwarding decision (Equation (2)), and that query overhead translates into measurable additional forwarding-path latency in a multi-broker setting, whereas WAP-BF's prefix-based early termination keeps its end-to-end latency at or below that of the Naive method.

The present experiment is limited to a minimal, one-hop, full-mesh deployment with three brokers; broader comparisons against production-grade distributed broker platforms (with persistence, session migration, and partitioned client state) and against non-Bloom-filter alternatives such as tries or prefix trees are left as future work.

VII. CONCLUSION

This paper presented WAP-BF, a Bloom filter-based topic summary designed to support MQTT wildcard subscriptions efficiently in distributed broker deployments. By indexing prefixes of subscription topics and executing a wildcard-aware, depth-first traversal of a candidate-prefix tree, WAP-BF avoids the exhaustive pattern enumeration that can dominate prior Bloom filter-based techniques.

Our evaluation demonstrated that WAP-BF substantially reduces the number of Bloom filter membership queries required per PUBLISH across the synthetic workloads considered in this study. The advantage became particularly pronounced as topic depth increased and when a published topic matched no subscriptions. Under the evaluated no-match conditions, WAP-BF reduced the per-PUBLISH Bloom filter query count by one to two orders of magnitude relative to the pattern-enumeration baselines. The modified broker-pipeline experiment further showed that fewer Bloom filter lookups translate into higher throughput and lower latency under load.

In the three-broker forwarding experiment, conducted with a synthetic workload, a target per-query false-positive rate of $p_f = 0.001$, and five random seeds, WAP-BF yielded a measured wasteful-forwarding ratio of 0.000 and the lowest reported mean, 50th-, 95th-, and 99th-percentile end-to-end latencies among the four methods. These observations are specific to the evaluated workloads, topology, Bloom filter parameters, and experimental design and should not be interpreted as general performance guarantees.

The present evaluation is not intended to constitute a full system-level benchmark. Comparisons with production-grade distributed broker platforms and implemented non-Bloom-filter-based alternatives, as well as evaluations using trace-driven MQTT workloads, remain future work. We also plan to evaluate the effects of dynamic subscription churn and its control-plane overhead, investigate broader broker cooperation architectures that exploit WAP-BF, and introduce caching to accelerate repeated lookups under realistic traffic patterns.

ACKNOWLEDGMENTS

This work was partly supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant No. 24K14935. This work was also partly supported by the Japan Science and Technology Agency (JST), PRESTO, Grant No. JPMJPR21P8.

CONFLICT OF INTEREST

Ryohei Banno has received research funding from Solid Surface Inc. under a collaborative research agreement, which supported this work.

REFERENCES

[1] MQTT, https://mqtt.org/ (accessed May 27, 2026).
[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] E. Longo, A. E. Redondi, M. Cesana, A. Arcia-Moret, and P. Manzoni, "MQTT-ST: a spanning tree protocol for distributed MQTT brokers," in *IEEE International Conference on Communications*, 2020, pp. 1–6.

[4] E. Longo, A. E. C. Redondi, M. Cesana, and P. Manzoni, "BORDER: A benchmarking framework for distributed MQTT brokers," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 728–17 740, 2022.
[5] A. Detti, L. Funari, and N. Blefari-Melazzi, "Sub-linear scalability of MQTT clusters in topic-based publish-subscribe applications," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1954–1968, 2020.
[6] R. Banno, J. Sun, S. Takeuchi, and K. Shudo, "Interworking layer of distributed MQTT brokers," *IEICE Transactions on Information and Systems*, vol. E102.D, no. 12, pp. 2281–2294, 2019.
[7] TBMQ, https://thingsboard.io/products/mqtt-broker/ (accessed May 27, 2026).
[8] A. Shvaika, D. Shvaika, D. Landiak, and V. Artemchuk, "A distributed architecture for MQTT messaging: the case of TBMQ," *Journal of Big Data*, vol. 12, no. 1, p. 224, 2025.
[9] EMQX, https://www.emqx.com/en (accessed May 27, 2026).
[10] HiveMQ, https://www.hivemq.com/ (accessed May 27, 2026).
[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[12] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.
[13] 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.
[14] A. M. Dominguez, T. Robles, R. Alcarria, and E. Cedeno, "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 May 27, 2026).
[16] J. Lee, M. Shim, and H. Lim, "Name prefix matching using Bloom filter pre-searching for content centric network," *Journal of Network and Computer Applications*, 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," *Applied Sciences*, vol. 10, no. 7, article no. 2226, 2020.
[18] N. Naaman, "Large scale connectivity infrastructure for an internet of things platform," https://www.incoseil.org/knowledge-base-item/2486 (accessed May 27, 2026).
[19] 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.
[20] R. Banno and Y. Watanabe, "Wildcard topic management using Bloom filter in distributed MQTT brokers," in *IEEE Consumer Communications & Networking Conference*, 2025, pp. 1–6.
[21] M. A. Spohn, "On MQTT scalability in the internet of things: Issues, solutions, and future directions," *Journal of Electronics and Electrical Engineering*, vol. 1, no. 1, pp. 23–34, 2022.
[22] OASIS Standard, "MQTT version 5.0," https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (accessed May 27, 2026).
[23] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
[24] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.
[25] Information-technology Promotion Agency, Japan, "Definition of spatial ID and spatial voxel," https://www.ipa.go.jp/en/digital/architecture-guidelines/definition-of-spatial-id-and-spatial-voxel.html (accessed May 27, 2026).
[26] OpenStreetMap Foundation, "Slippy map tilenames," https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (accessed May 27, 2026).
[27] H. Parzyjegla, C. Wernecke, G. Mühl, E. Schweissguth, and D. Timmermann, "Implementing content-based publish/subscribe with OpenFlow," in *ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1392–1395.
[28] W. Badreddine, K. Zhang, and C. Talhi, "Monetization using blockchains for IoT data marketplace," in *IEEE International Conference on Blockchain and Cryptocurrency*, 2020, pp. 1–9.
[29] 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 May 27, 2026).
[30] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.
[31] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.
[32] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *ACM International Conference on Emerging Networking Experiment and Technologies*, 2014, pp. 75–88.
[33] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.
[34] C. E. Rothenberg, C. A. Macapuna, F. L. Verdi, and M. F. Magalhães, "The deletable Bloom filter: a new member of the Bloom family," *IEEE Communications Letters*, vol. 14, no. 6, pp. 557–559, 2010.
[35] G. K. Zipf, *Human Behavior And The Principle Of Least Effort*. Addison Wesley Press, Inc., 1949.
[36] H. S. Heaps, *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., 1978.
[37] L. Lü, Z.-K. Zhang, and T. Zhou, "Zipf's law leads to Heaps' law: Analyzing their relation in finite-size systems," *PLOS ONE*, vol. 5, no. 12, pp. 1–11, 2010.
[38] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[39] Mochi-MQTT, https://github.com/mochi-mqtt/server (accessed May 27, 2026).
[40] MQTTLoader, https://github.com/dist-sys/mqttloader (accessed May 27, 2026).
[41] R. Banno, K. Ohsawa, Y. Kitagawa, T. Takada, and T. Yoshizawa, "Measuring performance of MQTT v5.0 brokers with MQTTLoader," in *IEEE Annual Consumer Communications & Networking Conference*, 2021, pp. 1–2.
[42] spaolacci/murmur3, https://github.com/spaolacci/murmur3 (accessed May 27, 2026).
[43] Eclipse Foundation, "Sparkplug specification 3.0," https://sparkplug.eclipse.org/specification/version/3.0/ (accessed May 27, 2026).
[44] HiveMQ, "Designing your UNS — Semantic Information Hierarchy," https://www.hivemq.com/blog/designing-your-uns-semantic-information-hierarchy/ (accessed May 27, 2026).

Ryohei Banno received his Bachelor of Engineering and Master of Information Science and Technology degrees from Hokkaido University in 2010 and 2012, respectively, and his Ph.D. in Science from Tokyo Institute of Technology in 2018. From 2012 to 2018, he was a researcher at NTT Network Innovation Laboratories. From 2018 to 2020, he was a researcher at Tokyo Institute of Technology. From 2020 to 2022, he was an Assistant Professor at Kogakuin University, and from 2022 to 2024, he was an Associate Professor. Since 2024, he has been an Associate Professor at Hitotsubashi University. His research interests include distributed systems and IoT systems. He is a member of ACM, IEEE, IEICE, and IPSJ.

Yoshito Watanabe received the B.E., M.E., and Ph.D. degrees in electrical and computer engineering from Yokohama National University, Yokohama, Japan, in 2009, 2011, and 2016, respectively. From 2016 to 2023, he was with the Social-ICT System Laboratory, National Institute of Information and Communications Technology (NICT), Japan. From 2023 to 2025, he was engaged in research and development with Solid Surface Inc., a venture company in Japan. Since 2025, he has been a Senior Research Engineer with the Computer and Data Science Laboratories, NTT, Inc. His current research interests include distributed systems and multi-AI agent systems. He is a member of ACM, IEEE, and IEICE.