Toward eBPF-Accelerated Pub-Sub Systems

Beihao Zhou
University of Waterloo
Waterloo, Canada
beihao.zhou@uwaterloo.ca

Samer Al-Kiswany
University of Waterloo
Waterloo, Canada
alkiswany@uwaterloo.ca

Mina Tahmasbi Arashloo
University of Waterloo
Waterloo, Canada
mina.arashloo@uwaterloo.ca

Abstract

Publish-subscribe (pub-sub) systems are a fundamental building block for real-time distributed applications, where high throughput and low latency are critical. Existing brokers can suffer performance bottlenecks as they operate in user space and rely on the socket API and full kernel stack traversal for every message. We present BPF-Broker, a novel pub-sub broker that leverages eBPF to accelerate message dissemination by decoupling the broker's control and data paths. Subscriber management is handled in user space, while message forwarding is done early in the kernel using the TC ingress and XDP hooks. Our evaluation shows that BPF-Broker achieves up to 3× higher throughput compared to our Socket-based baseline broker under high subscriber counts, and up to 2-10× lower end-to-end latency. These results highlight the potential of eBPF in accelerating pub-sub systems.

CCS Concepts

• **Networks → In-network processing**; **Cloud computing**; • **Software and its engineering → Real-time systems software**; **Message passing**.

Keywords

Publish-subscribe system, Data streaming, eBPF (extended Berkeley Packet Filter), Message queueing systems

ACM Reference Format:
Beihao Zhou, Samer Al-Kiswany, and Mina Tahmasbi Arashloo. 2025. Toward eBPF-Accelerated Pub-Sub Systems. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3748355.3748365

1 Introduction

Publish-subscribe (pub-sub) systems are a widely adopted communication paradigm in distributed infrastructures, enabling decoupled and scalable message dissemination. In a typical pub-sub architecture, *publishers* generate messages associated with specific topics, and *subscribers* subscribe to receive messages published at a certain topic. Pub-sub service is often offered by *Brokers*, dedicated servers that host topics, queue messages for each topic, and fan-out messages to all subscribers of a topic.

Pub-sub systems serve as the backbone for real-time applications across a wide range of domains, including metrics pipelines, streaming data analytics, IoT networks, autonomous vehicles, and microservice orchestration [2, 5, 12, 15, 18]. In these environments, a single publisher may need to broadcast frequent updates to tens or hundreds of subscribers [1, 8, 28]. Moreover, in many cases, timeliness outweighs reliability: missing an occasional sensor update is acceptable as long as the next one arrives in time [4, 10, 11, 20]. This makes message fan-out in the broker a particularly performance-critical operation, demanding high throughput and low latency.

Existing brokers such as RabbitMQ [29] and ActiveMQ [3] are implemented as user-space applications that use the socket interface and the traditional Kernel stack to send and receive messages. That is, when using these brokers, every publish request has to traverse up the entire kernel network stack and cross over to the user space to get processed, and each of the per-subscriber messages generated in response has to cross over the boundary and traverse back down the stack. The stack overhead can grow prohibitively large with an increasing number of subscribers – for large message fan-outs, this can result in tens of milliseconds of latency in processing publish requests and throughput degradation, which worsens further under high load [6, 7, 27].

In this paper, we explore using eBPF to accelerate pub-sub brokers. We observe that the core logic of the performance-critical data path of a pub-sub broker, i.e., the message fan-out for publish requests, is conceptually simple: when receiving a publish request, the broker needs to identify the topic, look up the subscribers, and forward the message. This simplicity makes the broker's publish logic an ideal candidate to implement in eBPF hooks that are lower in the stack, i.e., XDP and TC. This way, the publish requests will be processed and disseminated to the subscribers without going through any of the traditional protocol processing logic in the kernel network stack or a user-space process.

Specifically, we present BPF-Broker, a broker with its message dissemination path fully implemented in eBPF. In designing BPF-Broker, we decouple the control and data paths: topic registration and subscriber management occur in a user-space process, while message dissemination in response to publish requests is performed fully in-kernel in eBPF hooks. The user-space control logic stores the subscriber list for each topic in a map-of-maps accessed by eBPF programs at the traffic control (TC) layer. BPF-Broker intercepts incoming packets at the TC ingress hook. For publish requests in a single UDP packet, it uses an eBPF map-of-maps to retrieve the subscriber list for the corresponding topic. It then uses `bpf_clone_redirect()` to replicate and send out messages to subscribers directly to the NIC transmit queues, bypassing the traditional protocol processing, sockets, queues, and crossing into user space altogether. To further accelerate message dissemination, BPF-Broker identifies, in the XDP hook, when a publish message is for a topic with only one subscriber and processes it right there, avoiding sending the packet to the kernel stack at all.

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/25/09
https://doi.org/10.1145/3748355.3748365

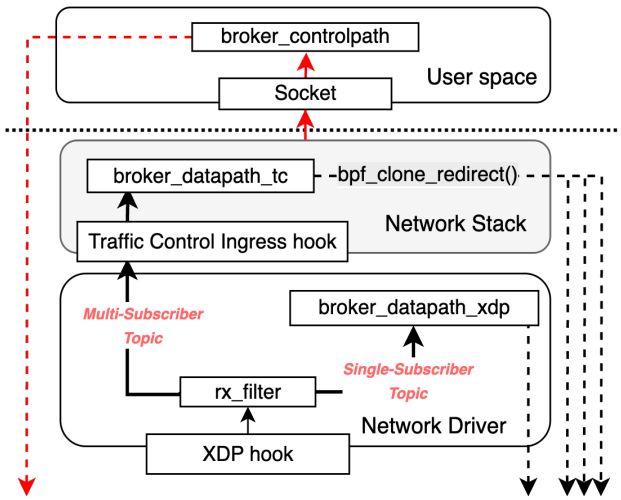


Figure 1: BPF-Broker Architecture

We evaluate BPF-Broker by comparing it against a broker that implements both the control and data paths in a user-space application and uses UDP sockets to send and receive messages. Our evaluation using synthetic workloads shows that BPF-Broker can achieve up to 2-10× lower end-to-end latency and 3× higher throughput. BPF-Broker can scale to hundreds of subscribers while maintaining stable performance under high fanout pressure, demonstrating the potential of eBPF in accelerating pub-sub systems.

2 BPF-Broker's Design

A pub-sub broker is responsible for two key functions: (1) maintaining topic-subscriber mappings, and (2) disseminating messages related to a certain topic to that topic's active subscribers. Maintaining topic-subscriber mapping, which involves topic registration and subscription management, is not performance-critical and benefits from flexibility. As such, it is implemented in BPF-Broker's control path in a user space process (Figure 1). Message dissemination, on the other hand, is performance-critical and is implemented in BPF-Broker's data path in low-level eBPF hooks for low latency and high throughput. This section provides an overview of the pub-sub protocol used for communication between BPF-Broker, publishers, and subscribers (§2.1), presents how BPF-Broker maintains topic-to-subscriber mappings in its control path (§2.2), and describes how it accelerates message fan-out in its data path (§2.3).

2.1 The Pub-Sub Protocol

There are several existing pub-sub protocols for communication between the broker and publishers and subscribers, including AMQP [14], MQTT [23], DDS [19], STOMP [24], and XMPP [22]. As a first step in accelerating brokers with eBPF, we focus on the core messages common across most protocols for topic registration, subscription, and publishing. All messages are sent over UDP and consist of a command keyword followed by one or more arguments, separated by spaces. Table 1 provides an overview of the message formats. We discuss extensions to support TCP and other commands and features in §4.

Topic Registration. Publishers can create a new topic using the `REGISTER` command. If the topic does not exist, the broker creates an entry for it. Regardless of whether the topic is newly created or already exists, the broker acknowledges the request with a `REGACK` message.

Operation	Incoming	Outgoing
Registration	`REGISTER <T>`	`REGACK <T>`
Subscription	`SUBSCRIBE <T>`	`SUBACK <T>`
Publishing	`PUBLISH <T> <M>`	`PUBLISH <T> <M>`

Table 1: Protocol message formats for registration, subscription, and publishing operations on the broker (<T> denotes the topic name and <M> the message content).

Topic Subscription. To receive messages for a topic, subscribers can issue a `SUBSCRIBE` command with the topic name. The broker extracts the subscriber's IP address and port number from the UDP packet metadata and records this as the destination information for that subscriber. The broker then responds with a `SUBACK` to confirm the subscription.

Publishing. A publisher sends a `PUBLISH` command, which includes the topic name and the message. The broker delivers a copy of the message to all the endpoints subscribed to the specified topic.

2.2 Maintaining Topic-Subscriber Mapping

BPF-Broker's control path is responsible for building and maintaining the topic-subscriber mapping. It runs in user space and uses UDP sockets for sending and receiving messages. When a client sends a `REGISTER` or `SUBSCRIBE` command, the message is delivered to the control-path's UDP socket through the traditional network stack. The control-path then parses the command and updates the corresponding eBPF maps responsible for keeping the topic-subscriber mapping.

Topic-Subscriber map. BPF-Broker uses eBPF maps, which are kernel-resident data structures used to store and exchange state between eBPF programs and user space [17]. BPF-Broker uses a map type that supports a single level of nesting, namely `BPF_MAP_TYPE_HASH_OF_MAPS`. The outer map is a hash table where each key is a topic name, and each value is a file descriptor referencing an inner map. The inner map is a hash table used to store the destination information of the subscribers for each topic. Specifically, each inner map stores subscriber endpoints using 64-byte keys: the first 32 bytes store the subscriber's IPv4 address, the next 16 bytes store the port number, and the remaining 16 bytes are reserved for alignment and future extensions.

Processing `REGISTER` and `SUBSCRIBE`. When a `REGISTER` request is received, the broker's control path checks whether the topic already exists in the outer map. If not, it creates an empty hash map to track the information for that topic's subscribers and inserts it into the map of maps with the topic name as key. For a `SUBSCRIBE` request, the control path extracts the topic and the subscriber's IP address and port from the packet, constructs the 64-byte subscriber key from the IP address and port, and inserts the key into the subscriber map.

2.3 Publishing Messages in eBPF

BPF-Broker's data path is responsible for processing `PUBLISH` commands. Our goal is to implement the data path in an eBPF hook in the kernel as early in the stack as possible to optimize performance.

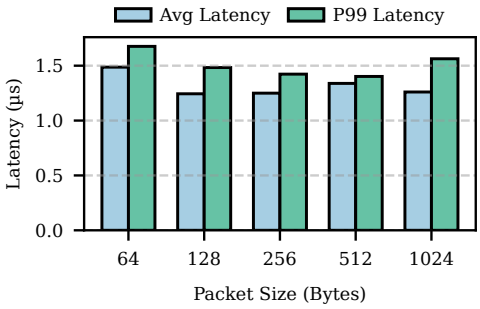


Figure 2: Time taken by a single packet to go from the XDP to the TC ingress hook for varying packet sizes.

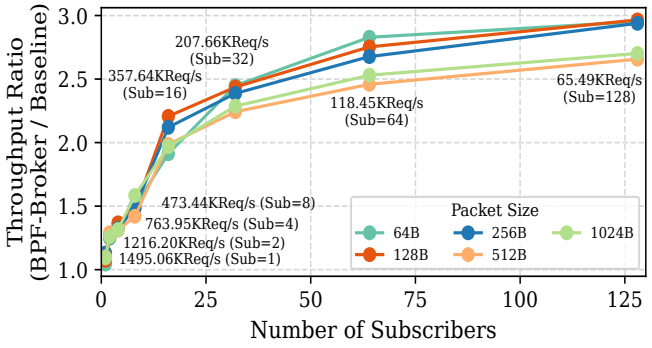


Figure 3: Throughput ratio of BPF-Broker over the baseline across varying packet sizes. The line is annotated with the absolute throughput of BPF-Broker in MReq/s.

eXpress Data Path (XDP). XDP is the earliest possible hook in the packet receive path. It runs in the Network Interface Card (NIC) driver after packets are received from the NIC and before they enter the kernel network stack. By operating on raw Ethernet frames from the DMA-backed ring buffer and avoiding the kernel's networking stack entirely, XDP can provide the lowest latency and highest throughput among the packet processing hooks.

However, XDP lacks support for packet cloning, which is essential for topics with multiple subscribers. As such, BPF-Broker can only use it for processing publish requests when there is only one subscriber for the corresponding topic. Specifically, when a packet is received at the XDP hook, BPF-Broker will extract the topic and look up the corresponding subscriber map in the topic-subscriber map of maps. If there is only one subscriber, it rewrites the UDP header in-place to update the source and destination fields, then forwards the packet using `XDP_TX`. Otherwise, it will use `XDP_PASS` to pass the packet to the Linux kernel, where it will be further processed by the Traffic Control (TC) ingress hook.

Traffic Control (TC) Ingress. The TC ingress hook runs after the network stack has performed some initial packet processing (including allocating `sk_buffs`) but before any layer-3 (IP) processing. This hook supports more complex packet handling, including cloning and redirection to multiple destinations. When a `PUBLISH` message makes it to this hook, the eBPF program looks up the inner map of subscribers for the target topic and iterates over each entry using `bpf_for_each_map_elem()`. For each subscriber, it clones the incoming packet, updates the headers to reflect the correct destination, and uses the helper function `bpf_clone_redirect()` to directly enqueue the modified packet to the TX queue. Once a copy of the message is created and redirected to each subscriber, the `PUBLISH` message has finished processing and can be safely dropped, without the need to traverse the rest of the stack.

Although TC introduces slightly more overhead than XDP, our measurements show that the additional per-packet latency is modest – on the order of 1μs (Figure 2). This hybrid use of XDP and TC allows BPF-Broker to opportunistically optimize processing publish requests even further when there is only one subscriber for a topic.

3 Evaluation

We have implemented BPF-Broker's control and data paths in ~1.3K lines of C code, and compared it with a baseline that implements the same functionality in user space and uses UDP sockets for receiving and sending messages. We show that implementing a broker's data path logic for message fan-out in the lower-level eBPF hooks (i.e., skipping most of the kernel stack) can improve latency, throughput, and CPU utilization under varying load, subscriber counts, and message sizes.

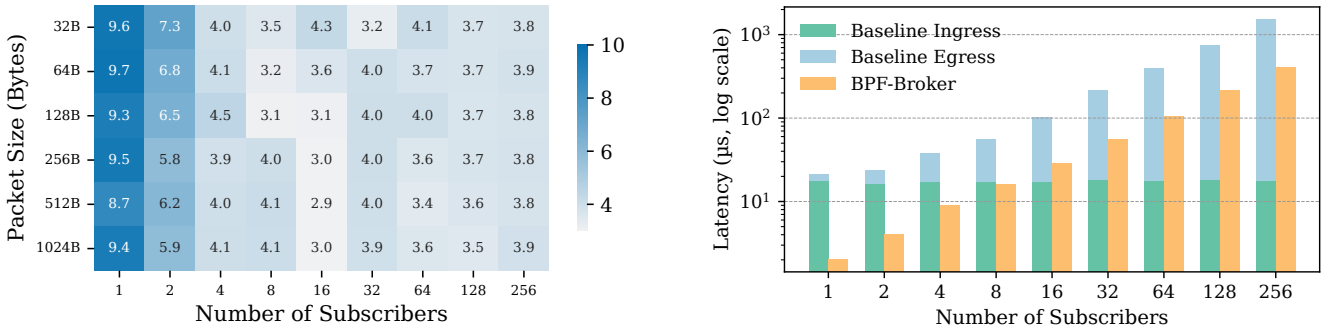
Experimental Setup. All experiments are conducted on two nodes from the CloudLab [21] c6525-25g cluster, one acting as the subscribers and publishers, and the other as the broker. Each node is equipped with a 16-core AMD EPYC 7302P processor running at 3.00 GHz, 128 GB of ECC RAM, two 480 GB SATA SSDs, and dual-port Mellanox ConnectX-5 25 Gbps NICs (PCIe v4.0). To eliminate variability from shared hardware threads, hyperthreading is disabled. Unless otherwise stated, we configure the broker with 16 dedicated threads, each pinned to a unique CPU core. Each core is also pinned to a dedicated RX and TX queue, and each RX/TX queue is configured with a depth of 8192 entries.

Baseline. Our baseline consists of a multi-threaded user-space program that spawns 16 threads, each pinned to a dedicated core and bound to a unique UDP port. Clients send packets to these 16 ports using a uniform random distribution to balance load across threads. RX steering is configured such that packets destined for a given port are directed to a specific RX queue, matching the core affinity of the receiving thread. Each thread uses `recvfrom()` to receive publish requests and forwards the received message to multiple subscriber addresses using individual `sendto()` calls on its UDP socket.

3.1 Throughput

The sender generates fixed-size publish requests to the same topic in an open loop, using a configurable number of threads to control the generated request load. We vary the load and observe that BPF-Broker saturates at ~1.5 MReq/s.

Figure 3 shows the ratio of BPF-Broker's throughput over the baseline for various message sizes and subscriber counts, where throughput is defined as the number of packets the broker receives and successfully fans out to all intended recipients. The absolute throughput of BPF-Broker for several data points is included in the plot. The results show that the BPF-Broker consistently outperforms the baseline under all packet sizes. The throughput ratio increases steadily with the number of subscribers, reaching up to 3×



(a) Per-packet latency ratio of baseline over BPF-Broker.

(b) Breakdown of ingress and egress latency for baseline (Packet size 1024B), compared to BPF-Broker's total latency.

Figure 4: Latency of BPF-Broker vs. baseline.

at high fan-out levels. For example, at 32 subscribers, the throughput gap widens to nearly 2.5× (210K vs. 85K req/s for packet sizes of 64B, 128B, and 256B) and with 128 subscribers, BPF-Broker sustains up to 3× the throughput of the baseline. This trend implies that BPF-Broker is more resilient to replication pressure and can maintain significantly higher throughput under both high load and large fan-out. We also observe that the throughput ratio increases more slowly for larger packet sizes. This is because we need to use `bpf_skb_pull_data` in the TC ingress hook to explicitly pull all the payload into the hook, which will take longer for larger packets.

We observe that BPF-Broker experiences throughput degradation as the number of subscribers increases. This is expected as every request is "done" processing when it generates as many packets as the number of subscribers. However, BPF-Broker degrades more gracefully and maintains a consistent performance advantage over the baseline.

3.2 Latency Micro-Benchmark

To isolate the benefits of performing the publish operation in XDP/TC, we run this set of experiments using a single CPU core and one RX/TX queue pair. Each experiment involves a single publisher sending one packet, avoiding batching, concurrency, or contention effects. For BPF-Broker, we measure latency as the time between the packet's arrival at the TC ingress hook and the completion of all clone operations. For the baseline, we similarly begin timing at TC ingress and record completion when TC egress observes the final clone being transmitted. Each experiment is repeated 200 times. We use the average in our reported results, but observe similar trends for the 99th percentile.

Figure 4a shows the ratio of per-packet latency of baseline over BPF-Broker, across varying packet sizes and subscriber counts. The ratio peaks at 9.7×, with an average around 4×. The highest ratios occur at low subscriber counts (1–2), where baseline incurs a high fixed ingress cost from traversing the kernel networking stack, while BPF-Broker clones packets directly in the kernel with minimal overhead before they hit any major protocol processing.

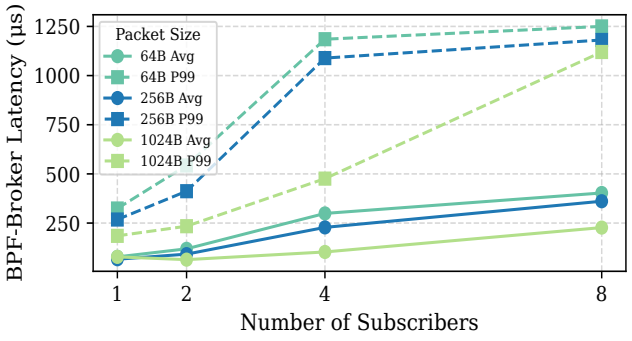
This trend is further clarified in Figure 4b, which breaks down the latency of the baseline's ingress and egress paths for 1024B packets (y-axis is in log-scale). Baseline ingress latency is measured from the TC ingress hook to when the packet is delivered to the UDP socket in user space, before any `sendto()` occurs. Baseline egress latency spans from the first `sendto()` call to when all cloned packets are seen at the TC egress hook. For BPF-Broker, we cannot meaningfully distinguish between ingress and egress as all the processing happens in the TC ingress (or XDP) hook.

At low fan-out (i.e., low subscriber count), baseline ingress dominates the total latency. BPF-Broker avoids this cost entirely as it skips most of the processing in the kernel stack and crossing into user space, hence the higher baseline to BPF-Broker latency ratio shows in Figure 4a. As fan-out increases, baseline egress dominates the total latency. BPF-Broker incurs an increasing cost as well, but not as much as the baseline. This is because the baseline makes a separate (`sendto()`) system call for each copy of the publish message and each of those messages needs to traverse the entire stack, while BPF-Broker performs the cloning in the TC ingress hook early on in the kernel. As such, the total baseline over BPF-Broker latency ratio stabilizes at higher subscriber counts, where the baseline ingress overhead is increasingly amortized.

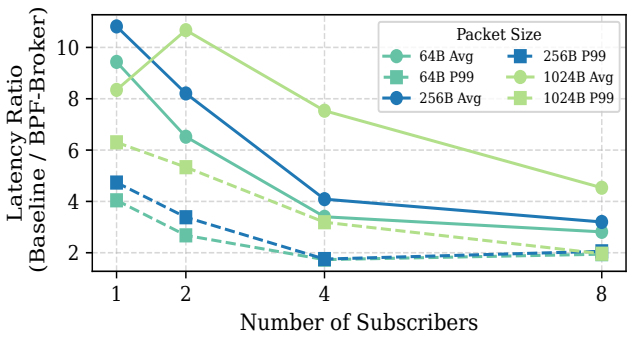
3.3 End-to-End Latency

We compare the end-to-end (E2E) latency of BPF-Broker with the UDP-socket baseline under both light and high load. Under light load, one thread on the client sends publish messages in an open loop without saturating the broker, allowing us to observe latency under low system stress. In contrast, the high load setup drives BPF-Broker to saturation, ensuring it operates near peak throughput with no bottlenecks observed at either RX or TX queues. Due to coarse-grained rate control for publish messages in our evaluation setup, we limit our evaluation to <= 8 subscribers per topic to avoid overwhelming the broker beyond its saturation point.

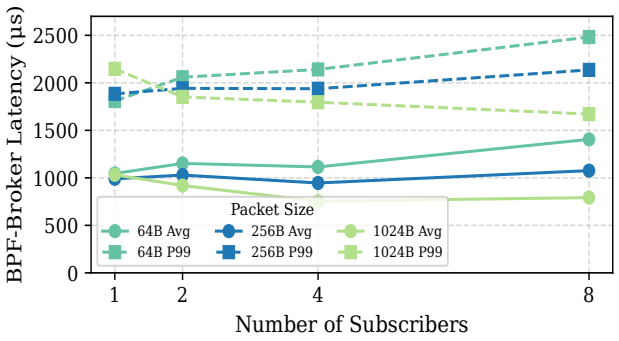
The results are depicted in Figure 5. Under light load (Figures 5a and 5b), BPF-Broker consistently maintains at least a 2× latency advantage over the baseline. BPF-Broker achieves low average latency, ranging from below 100μs with a single subscriber to below 500μs with 8 subscribers. The baseline exhibits significantly higher average and tail latency, resulting in normalized latency ratios (baseline / BPF-Broker) of up to 10× for small subscriber counts. This advantage narrows slightly as the number of subscribers grows, due to the increasing number of per-packet clone operations. Under high load (Figures 5c and 5d), BPF-Broker sustains sub-millisecond



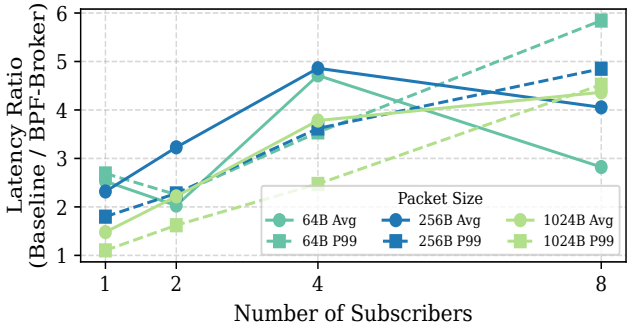
(a) End-to-end latency of BPF-Broker (light load).



(b) Latency ratio of baseline over BPF-Broker (light load).



(c) End-to-end latency of BPF-Broker (high load).



(d) Latency ratio of baseline over BPF-Broker (high load).

Figure 5: End-to-end latency comparison between BPF-Broker and UDP-socket baseline under light and high load. Solid lines are for average and dashed lines are for 99th-percentile latency.

average latency for most packet sizes and $< 1ms$ for $1024B$ packets across all configurations. The normalized latency ratio remains favorable, peaking at up to 6× improvement over the baseline.

Interestingly, we observe opposite latency trends in the latency ratio under light and high load. Under light load, the latency ratio decreases as the number of subscribers increases. This matches our observations in §3.2 that at low contention, BPF-Broker's latency gap from the baseline is higher at low subscriber count, where the baseline ingress dominates the total latency. In contrast, under high load, the latency ratio increases with more subscribers. Here, the baseline suffers significantly from kernel socket buffer pressure that BPF-Broker avoids through in-kernel replication. Nevertheless, BPF-Broker achieves significantly lower latency than the socket-based baseline under both light and high load, and remains robust and performant even under substantial replication pressure.

3.4 CPU Utilization

Figure 6 shows per-core CPU utilization, averaged across 16 cores, under increasing publish request rates for the UDP socket baseline, BPF-Broker's XDP path (single-subscriber topics), and BPF-Broker's TC ingress path (multi-subscriber topics). The error bars represent the standard deviation across repeated measurements.

We observe that the baseline quickly saturates CPU resources, reaching over 90% utilization per core at just 1.2 MReq/s, and flattening at full saturation (near 100%) beyond that point. In contrast, BPF-Broker's TC ingress path shows a more efficient CPU usage pattern, increasing almost linearly with throughput and reaching near-saturation only at around 1.45 MReq/s. Notably, BPF-Broker's XDP path has the lowest CPU usage across all throughput levels, remaining below 55% even at the highest tested rate of 1.48 MReq/s.

This behavior highlights the benefit of BPF-Broker's two-tier data path architecture for handling mixed pub-sub workloads. Topics with a single subscriber are served via the XDP path, enabling early delivery with minimal processing overhead. Multi-subscriber topics are processed through the TC ingress hook, which supports in-kernel packet cloning for efficient replication. BPF-Broker 's two-tier data path architecture minimizes contention at high load.

4 Discussion and Future Work

BPF-Broker demonstrates that in-kernel fanout using eBPF at TC ingress and XDP can substantially reduce broker resource usage, lower end-to-end pub-sub latency, and increase throughput. However, real-world pub-sub systems using production-grade protocols such as MQTT and AMQP often require richer broker capabilities. As we discuss below, we plan to explore hybrid designs that combine multiple eBPF hooks as in-kernel fast paths with user-space control paths to implement these features. These extensions would position BPF-Broker as a low-overhead, standards-compliant broker core for high-performance pub-sub systems.

QoS guarantees. BPF-Broker currently supports QoS 0 (fire-and-forget), which is common in latency-sensitive pub-sub systems [13]. Exploring support for higher QoS levels such as at-least-once (QoS

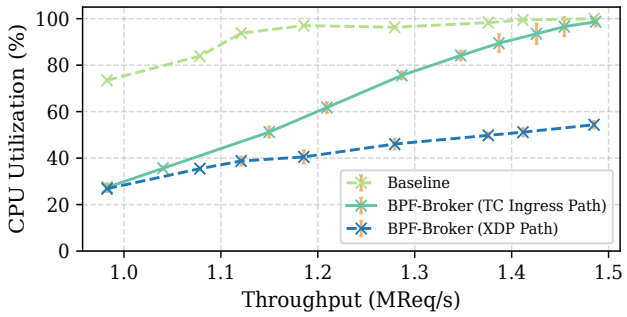


Figure 6: CPU utilization of BPF-Broker (TC ingress and XDP paths) compared to a UDP socket baseline (orange lines are error bars).

1) and exactly-once (QoS 2) delivery, and providing message delivery acks is part of our future work. For example, one avenue we are exploring is to track acknowledgments in eBPF maps and incorporate a lightweight user-space scheduler that periodically inspects these maps to trigger retransmissions as needed.

TCP vs. UDP and message persistence. Our current design uses UDP, which is suitable for high-throughput, real-time, fire-and-forget scenarios such as metrics pipelines and sensor networks, where occasional loss is acceptable and the benefits of bypassing more complicated TCP processing are significant. We plan to explore reliable transport protocols like TCP for scenarios that need ordered, loss-free delivery and/or persistent connections using eBPF hooks such as `sockmap` and `sk_msg`. We also plan to explore using these hooks to support message persistence, e.g., by employing snapshot-based buffering strategies for lightweight retransmission, allowing selective recovery when subscribers rejoin or experience loss without incurring the overhead of userspace buffering and stream management.

Other advanced broker features. We also plan to extend BPF-Broker with support for priority-based message scheduling and rate control using the TC eBPF hooks to allow for more fine-grained control over how the broker's compute and communication resources are allocated across various publishers, subscribers, and topics. Moreover, we plan to use the XDP hook to do complex topic-matching policies, message filtering, and per-topic access control.

Deployment considerations. All experiments and development were conducted on Ubuntu 24.04 with Linux kernel version 6.8. While BPF-Broker eliminates userspace processing overhead, it introduces deployment frictions not present in mainstream brokers like RabbitMQ or Redis. Specifically, it requires a modern Linux kernel ($\geq$ 5.10) that supports key features such as map-of-maps and `bpf_clone_redirect()`, and typically needs root privileges to attach programs to TC or XDP hooks. Although these features are supported as of version 5.10, we used kernel version 6.8 to leverage the latest improvements in stability and performance. These constraints may limit deployability in containerized or multi-tenant environments without privileged access.

Ethics statement. This work does not raise any ethical issues.

5 Related Work

eBPF-based Acceleration. Recent work has explored eBPF to accelerate networked applications. BMC [9] accelerates Memcached by serving UDP GET requests in XDP from in-kernel cache, while using the TC egress hook to monitor responses and maintain cache coherence. Electrode [31] accelerates distributed protocols (i.e. Multi-Paxos) by offloading performance-critical operations like broadcasting and quorum handling to XDP and TC egress. DINT [32] targets distributed transaction processing by offloading key-value access, locking, and logging into the kernel via eBPF. It uses XDP for parsing and execution, and TC egress for response finalization and state synchronization. BOAD [25] optimizes broadcast and aggregation using XDP for early processing and TC egress for coordinated replication. XAgg [30] implements in-kernel gradient aggregation using XDP to improve distributed machine learning performance.

Unlike these systems, BPF-Broker targets generic pub-sub workloads and performs in-kernel message replication. It opportunistically uses XDP to handle single-subscriber topics, avoiding kernel buffer allocation and reducing CPU overhead. XDP handles lightweight fast paths, while TC ingress manages cloning for multi-subscriber delivery.

Accelerating Publish-Subscribe Systems. Prior work has improved pub-sub performance through various broker-level optimizations. For instance, Mosquitto [16], a lightweight pub-sub broker for the MQTT protocol [23], improves general-purpose efficiency by handling I/O in user space using event loops (e.g., `epoll()`). KafkaDirect [26] uses one-sided RDMA to bypass broker CPUs, enabling zero-copy writes and direct remote reads by producers and consumers. Instead of relying on user-space optimizations or specialized hardware, BPF-Broker offloads pub-sub logic into the kernel, enabling low-latency message replication with broad deployability on commodity systems.

6 Conclusion

We presented BPF-Broker, an eBPF-based pub-sub system that performs in-kernel fan-out using the TC ingress hook, eliminating the need for userspace message handling. Our preliminary evaluation shows that BPF-Broker achieves significantly lower latency and higher throughput compared to a socket-based broker, highlighting the potential of programmable kernel hooks for accelerating pub-sub systems.

Acknowledgments

We thank the anonymous reviewers of the ACM SIGCOMM eBPF workshop for their helpful feedback. This work is partially supported by the Canada Research Chair grant on "Minimizing Human Error in Modern Networks".

References

[1] Amazon Web Services. 2025. AWS IoT Core Endpoints and Quotas. (2025). https://docs.aws.amazon.com/general/latest/gr/iot-core.html#thing-limits Accessed: 2025-05-18.

[2] Kyoungho An, Subhav Pradhan, Faruk Caglar, and Aniruddha Gokhale. 2012. A publish/subscribe middleware for dependable and real-time resource monitoring in the cloud. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management (SDMCMM '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2405186.2405189

[3] Apache Software Foundation. 2025. Apache ActiveMQ: Open Source Message Broker. https://activemq.apache.org/. (2025). Accessed: 2025-05-18.

[4] Helbert da Rocha, Tania L Monteiro, Marcelo Eduardo Pellenz, Manuel C Penna, and Joilson Alves Junior. 2020. An MQTT-SN-based QoS dynamic adaptation method for wireless sensor networks. In *Advanced Information Networking and*

Applications: Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA-2019) 33. Springer, 690–701.

[5] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert Podhorszki. 2014. Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 246–255. https://doi.org/10.1109/CCGrid.2014.104

[6] Jasenka Dizdarevic, Marc Michalke, and Admela Jukan. 2023. Engineering and experimentally benchmarking open source MQTT broker implementations. *arXiv preprint arXiv:2305.13893* (2023).

[7] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS '17)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/3093742.3093908

[8] Julien Gascon-Samson, Franz-Philippe Garcia, Bettina Kemme, and Jörg Kienzle. 2015. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 486–496. https://doi.org/10.1109/ICDCS.2015.56

[9] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 487–501.

[10] Kannan Govindan and Amar Prakash Azad. 2015. End-to-end service assurance in IoT MQTT-SN. In *2015 12th Annual IEEE consumer communications and networking conference (CCNC)*. IEEE, 290–296.

[11] Xiaolong Guo, Song Han, X. Sharon Hu, Xun Jiao, Yier Jin, Fanxin Kong, and Michael Lemmon. 2021. Towards scalable, secure, and smart mission-critical IoT systems: review and vision. In *Proceedings of the 2021 International Conference on Embedded Software (EMSOFT '21)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3477244.3477624

[12] Daniel Happ, Niels Karowski, Thomas Menzel, Vlado Handziski, and Adam Wolisz. 2017. Meeting IoT platform requirements with open pub/sub solutions. *Annals of Telecommunications* 72 (2017), 41–52.

[13] HiveMQ. 2025. MQTT Essentials Part 6: MQTT Quality of Service Levels. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/. (2025). Accessed: 2025-07-22.

[14] Joshua Kramer. 2009. Advanced message queuing protocol (AMQP). *Linux Journal* 2009, 187 (2009), 3.

[15] Seda Kul and Ahmet Sayar. 2021. A survey of publish/subscribe middleware systems for microservice communication. In *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. IEEE, 781–785.

[16] Roger A Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software* 2, 13 (2017), 265.

[17] Chang Liu, Byungchul Tak, and Long Wang. 2024. Understanding Performance of eBPF Maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions (eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3672197.3673430

[18] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge computing for autonomous driving: Opportunities and challenges. *Proc. IEEE* 107, 8 (2019), 1697–1716.

[19] Object Management Group (OMG). 2015. Data Distribution Service (DDS) Specification. https://www.omg.org/spec/DDS/1.4/. (2015).

[20] Thomas Rausch, Stefan Nastic, and Schahram Dustdar. 2018. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 191–197. https://doi.org/10.1109/IC2E.2018.00043

[21] Robert Ricci, Eric Eide Wong, Leigh Stoller, Mike Hibler, Jon Duerig, David Webb, Kirk Johnson, Aditya Akella, and Glenn Ricart. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, Philadelphia, PA. https://www.cloudlab.us/

[22] Peter Saint-Andre. 2011. Extensible Messaging and Presence Protocol (XMPP). https://datatracker.ietf.org/doc/html/rfc6121. (2011).

[23] Dipa Soni and Ashwin Makwana. 2017. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, Vol. 20. 173–177.

[24] STOMP Protocol Working Group. 2023. STOMP - Simple (or Streaming) Text Oriented Messaging Protocol. https://stomp.github.io/stomp-specification-1.2.html. (2023).

[25] Jianchang Su, Yifan Zhang, Linpu Huang, and Wei Zhang. 2024. BOAD: Optimizing Distributed Communication with In-Kernel Broadcast and Aggregation. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions (eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3672197.3673438

[26] Konstantin Taranov, Steve Byan, Virendra Marathe, and Torsten Hoefler. 2022. KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2191–2204. https://doi.org/10.1145/3514221.3526056

[27] Tyler Treat. 2016. Benchmarking Message Queue Latency. (February 2016). https://bravenewgeek.com/benchmarking-message-queue-latency/ Accessed: 2025-05-06.

[28] P. Triantafillou and A. Economides. 2004. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* 562–571. https://doi.org/10.1109/ICDCS.2004.1281623

[29] VMware. 2025. RabbitMQ: Open Source Message Broker. https://www.rabbitmq.com/. (2025). Accessed: 2025-05-18.

[30] Qianyu Zhang, Gongming Zhao, Hongli Xu, and Peng Yang. 2024. XAgg: Accelerating Heterogeneous Distributed Training Through XDP-Based Gradient Aggregation. *IEEE/ACM Transactions on Networking* 32, 3 (2024), 2174–2188. https://doi.org/10.1109/TNET.2023.3339524

[31] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou

[32] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. {DINT}: Fast {In-Kernel} Distributed Transactions with {eBPF}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 401–417.