Accepted for publication in IEEE Internet of Things Journal,
DOI: 10.1109/JIOT.2021.3113777, For personal use only

Q-Flag: QoS-Aware Flow-Rule Aggregation in Software-Defined IoT Networks

Niloy Saha, *Student Member, IEEE,* Sudip Misra, *Senior Member, IEEE*
and Samaresh Bera, *Student Member, IEEE*

***Abstract*—Software-defined IoT (SDIoT) is a promising approach to address the requirements of IoT, such as network management, quality of service (QoS), and resource utilization. The advantages of SDIoT are facilitated by the separation of the data- and the control-planes using *flow-rules*, that allow fine-grained control over individual flows. However, the number of flow-rules that can be placed at the switches is limited, leading to scalability issues in SDIoT. Existing approaches to flow-rule management either do not consider the impact on quality-of-service (QoS) or are applicable only to a particular topology. In this paper, we propose a QoS-aware flow-rule aggregation scheme for generic network topologies, which aims to achieve satisfactory trade-off among flow-rule compression and its impact on the QoS of IoT traffic flows. Specifically, the proposed scheme adaptively aggregates flow-rules while considering different QoS requirements of IoT applications in the network, and the flow-rule capacity of the switches. The proposed scheme consists of the following components — a) a path selection heuristic to increase the total number of flow-rules that can be accommodated in the network, and b) a multi-arm bandit based flow-rule aggregation scheme capable of reducing the number of flow-rules, while maintaining adequate performance in terms of QoS. Experimental results using IoT traffic show that, on average, the proposed scheme is capable of reducing the average end-to-end delay and QoS-violated flows in the network by 22% and 30%, respectively, compared to the state-of-the-art schemes.**

***Index Terms*—Software-Defined Networking, Internet of Things, Quality of Service, Rule-aggregation**

I. Introduction

Recent advances in the Internet of Things (IoT) technologies have shown that adopting a software-defined network (SDN) based IoT architecture (SDIoT) offers many advantages such as simplification of network management, improved quality-of-service (QoS), efficient mobile-edge computing, and dynamic resource utilization for heterogeneous IoT resources [**?**], [**?**], [**?**], [**?**]. These benefits are facilitated by the separation of the control from the data-plane using match-action forwarding policies (*flow-rules*), allowing fine-grained control over individual IoT traffic flows [**?**]. In this work, we focus on an SDIoT network, as shown in Figure **??**, where an SDN-enabled backbone is used to carry heterogeneous traffic generated by various IoT devices connected through different wired/wireless access networks. Various IoT applications running on top of the SDN controller, such as QoS routing and access-control, express their objectives by placing appropriate flow-rules at the switches. Fine-grained control by these applications requires

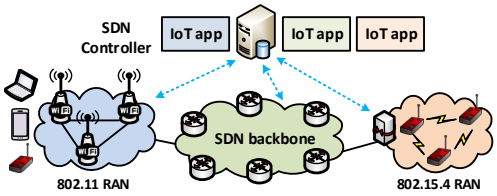


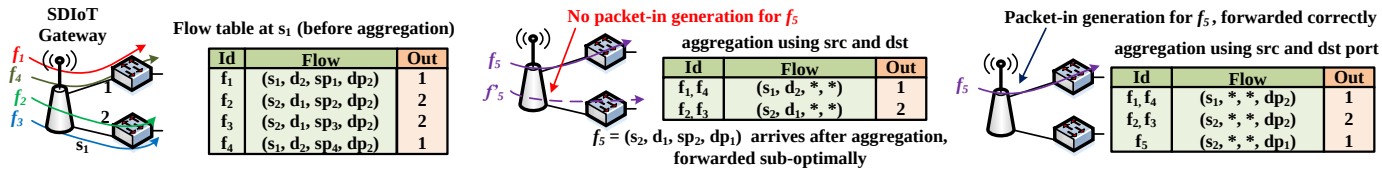
Figure 1: SDN-based IoT architecture

specifying *exact-matches* on multiple header fields, leading to the generation of a vast number of flow-rules. However, due to cost and energy considerations, only a limited number of flow-rules can be accommodated in the *flow-table* of switches [**?**]. This problem is exacerbated in SDIoT networks, with a massive number of low-volume traffic flows, many of which require differential treatment. To address this issue, recent works [**?**], [**?**] considered strategies in which flow-rules are *aggregated*[1] after flow arrival in order to reduce the number of flow-rules. These solutions mainly focused on aggregating a given set of flow-rules without violating the existing forwarding policies. However, in an online scenario, flow-rules *arriving after the aggregation* can be affected, as shown in Figure **??**.

From the figure, we observe that flow f_5 that arrives at an SDIoT gateway after aggregation using nw_src and nw_dst, matches the existing $(s_2, d_1, *, *)$ aggregated flow-rule, and is transparently forwarded out port 2 without contacting the controller (packet-in). This bypasses any IoT applications running at the SDN controller, such as QoS routing, and may cause sub-optimal forwarding and QoS violations. On the other hand, from Figure **??** (right), we observe that aggregating the flow-rules using a combination of nw_src and dst_port is capable of correctly forwarding the IoT flows under consideration. Therefore, the choice of match-fields considered for flow-rule aggregation affects the QoS forwarding of IoT flows. From Figure **??**, it may be noted that for the flows under consideration, aggregating them by nw_src and nw_dst yields a 50% decrease in the number of flow-rules, while aggregating them by nw_src and dst_port yields a 25% decrease. Therefore, there is a fundamental trade-off among the flow-rule reduction and the QoS violations, and the essence of the problem is to determine a suitable combination of match-fields capable of reducing the number of flow-rules, while minimizing the impact of the QoS of newly arriving IoT flows.

• N. Saha, S. Misra and S. Bera are with the Computer Science and Engineering Department, Indian Institute of Technology, Kharagpur, 721302, India, Email: niloysaha@iitkgp.ac.in, smisra@cse.iitkgp.ac.in, s.bera.1989@ieee.org

[1]Flow-rules are aggregated by wildcarding $(*)$ some match-fields which then match on any incoming packet.



Id	Flow	Out
f_1	(s_1, d_2, sp_1, dp_2)	1
f_2	(s_2, d_1, sp_2, dp_2)	2
f_3	(s_2, d_1, sp_3, dp_2)	2
f_4	(s_1, d_2, sp_4, dp_2)	1

Id	Flow	Out
f_1, f_4	(s_1, d_2, *, *)	1
f_2, f_3	(s_2, d_1, *, *)	2

Id	Flow	Out
f_1, f_4	(s_1, *, *, dp_2)	1
f_2, f_3	(s_2, *, *, dp_2)	2
f_5	(s_2, *, *, dp_1)	1

Figure 2: Example showing the impact of flow-rule aggregation on QoS forwarding of IoT flows. Flow-rules are considered in the form (nw_src, nw_dst, src_port, dst_port, action). Correct forwarding paths are drawn solid, incorrect ones are dashed.

Therefore, in this work, we propose a QoS-aware flow-rule aggregation scheme to address the problem described above. The proposed scheme uses a twofold mechanism to address the rule management problem — a) a path selection heuristic termed *Bestfit-Z*, which minimizes the number of flow-rules in the network, b) a fast flow-rule aggregation scheme based on multi-arm bandit, which selects appropriate match-fields to minimize the impact of aggregation on the QoS of newly arrived flows.

The flow-rule aggregation scheme presented in this paper extends our earlier work [**?**] in several aspects. In particular, the extended flow-rule aggregation scheme relies on a multi-arm bandit (MAB)-based approach to automatically select the best key for aggregation, and jointly considers the number of flow-rules present, as well as the number of QoS violations.

In summary, compared to [**?**], we present the following new contributions:

- We propose the *Bestfit-Z* path selection heuristic that takes into account switches with high flow-rule utilization (bottleneck switches), to minimize the total number of flow-rules in the network. The Bestfit-Z heuristic offers 7% improved performance over our previous path-selection approach [**?**].
- We propose a fast reactive flow-rule aggregation scheme, based on a multi-arm bandit (MAB) algorithm, which selects a suitable combination of match-fields to achieve adequate trade-off among flow-rule reduction and impact on QoS of online arriving IoT flows.
- We have widely strengthened the performance analysis by including new performance metrics, different topologies from the Internet Topology Zoo [**?**], and the flow-table sharing (FTS) [**?**] benchmark scheme. Experimental results on IoT traffic using the POX SDN controller and the Mininet network emulator show that the proposed scheme is capable of reducing the average end-to-end delay and QoS-violated flows by 22% and 30%, respectively, compared to the existing schemes, while achieving comparable performance in terms of flow-rule aggregation.

The rest of the paper is organized as follows. In Section **??**, we analyze the relevant state-of-the-art. Section **??** presents an overview of the proposed scheme as well as its applicability to SDIoT networks. Section **??** presents the proposed flow-rule aggregation scheme in detail. In Section **??**, we evaluate the performance of the proposed scheme. Finally, we conclude the paper in Section **??** and present directions for future work.

II. Related Work

We classify the relevant state-of-the-art into a) compression-based schemes, and b) routing-based schemes.

Compression-based schemes focus on aggregating a given set of flow-rules, while maintaining consistency with the existing forwarding policies [**?**], [**?**], [**?**], [**?**]. Rifai *et al.* [**?**] proposed a heuristic approach for compressing flow-rules in data-center networks (DCNs), which yields a 3-approximation for the offline flow-rule reduction problem. Mimidis *et al.* [**?**] proposed a flow-aggregation scheme for SDN-based backhaul networks, where the flows are separated into different QoS classes and the SDN controller is subsequently used to check if recently arrived flows can be aggregated with existing flows while maintaining the QoS requirements. A similar scheme was proposed by Kosugiyama *et al.* [**?**], where flows having the same source and destination are aggregated, as long as delay requirements are satisfied. Zhang *et al.* [**?**] considered joint path-selection and rule aggregation in SDIoT networks using a distributed Markov approximation approach. The authors mainly focused on maximizing network operator revenue, and QoS parameters other than throughput were not discussed. All of these schemes [**?**], [**?**], [**?**], [**?**] mainly focused on achieving a compressed flow-table consistent with the existing forwarding policies. However, in an online scenario, flows arriving after flow-rule aggregation can be affected, as shown in Figure **??**. The authors in [**?**] avoid this by utilizing level-0 switches in a fat-tree topology to contact the controller for every flow, which makes it applicable to only DCNs. Phan *et al.* [**?**] proposed a learning-based approach to pre-emptively determine flow-table overflow, and aggregate flow-rules using layer 2 destination. However, neither flow-table consistency nor impact on QoS were discussed. Singh *et al.* [**?**] proposed a probabilistic data-structure for space-efficient storage of flow-rules at SDN switches. This is complementary to our work, where we focus on reducing the total number of flow-rules.

Routing-based schemes adopt various optimization and heuristic approaches to minimize the number of flow-rules used to route flows, or calculate QoS routing paths under flow-rule limits [**?**], [**?**], [**?**], [**?**], [**?**]. Huang *et al.* [**?**] considered a joint rule-placement and traffic-engineering scheme to reduce redundant flow-rules for multiple unicast sessions under QoS constraints. In contrast, we focus on the more general problem of minimizing flow-rules independent of a particular application. Nguyen *et al.* [**?**] proposed a routing scheme to minimize the number of flow-rules in DCNs, where existing routing paths are explicitly changed, as long as end-point policies are maintained. This does not consider the fine-grained forwarding required for IoT traffic flows. Several

schemes focused on routing within flow-rule limits, rather than minimizing an existing set of rules. Giroire *et al.* [**?**] proposed a scheme for energy-aware routing in SDN while respecting the capacity and flow-rule constraints. Qiao *et al.* [**?**] focused on minimizing the control overhead when the flow-table at a switch is full. The author proposed a routing scheme, which, on flow-table overflow, randomly forwards a new flow to a less utilized next-hop switch instead of sending packet-in to the controller. This forwarding scheme causes uniform utilization of the flow-table across all switches, leading to the minimization of the total number of flow-rules.

Synthesis: Critical analysis of the existing literature reveals that the existing approaches mainly focused on QoS forwarding without rule-aggregation, or consider rule-aggregation with QoS, but apply to a specific topology such as fat-tree. Therefore, we present Q-Flag, a QoS-aware flow-rule aggregation scheme for generic network topologies, which aims to achieve satisfactory trade-off among flow-rule compression and its impact on the QoS of IoT traffic flows.

III. System Model

In this Section, we present the system model of the proposed QoS-aware flow-rule aggregation scheme, named Q-Flag.

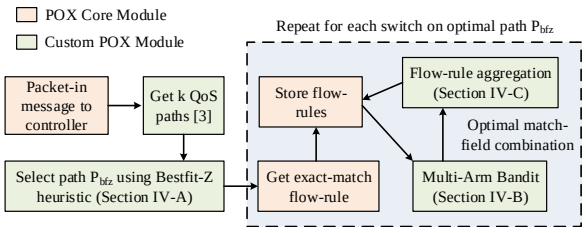


Figure 3: Overview of the proposed scheme, showing the different components built on top of the POX SDN controller

A. System Overview

The different components of the proposed Q-Flag scheme are presented in Figure **??**. The components are built as python-based application modules on top of the POX SDN controller[2]. In the figure, the custom modules for Q-Flag are shown in green, while the built-in core modules of the POX controller are shown in red.

The *packet-in module* is responsible for generating control messages that contain meta-data about flows which is used by the controller to place appropriate flow-rules. For a given flow (packet-in), the *QoS routing module* is used to return a set of k paths that satisfy the QoS requirements of the flow. This module utilizes the QoS routing scheme presented in our earlier work [**?**], which uses Yen's K-shortest paths algorithm to select QoS forwarding paths using exact-match flow-rules. The value of k denotes a trade-off among the number of QoS paths returned, and the time-complexity of the algorithm (given as $O(k|\mathcal{S}|(|\mathcal{L}| + |\mathcal{S}| \log |\mathcal{S}|))$, where $|\mathcal{L}|$ denotes the number of links). In this work, we use the value of $k = 3$. Since the main focus of this work is flow-rule aggregation, we limit our discussion on QoS routing for SDN. Interested readers may refer to [**?**] for details. The *path selection module* utilizes the proposed Bestfit-Z heuristic to select a suitable path from the k QoS paths, in order to minimize the overall number of flow-rules and increase in the number of successfully routed flows. The Bestfit-Z heuristic is described further in Section **??**. The built-in *flow-rule module* of the POX controller is used to translate the path information into appropriate exact-match flow-rules to be placed at the switches. This module also maintains a database of flow-rules and provides the necessary interface between the multi-arm bandit module and flow-rule aggregation module. The *multi-arm bandit (MAB) module* is used to select a suitable match-field combination for flow-rule aggregation in order to achieve satisfactory trade-off between the number of flow-rules and QoS violated flows. This module spawns a separate thread at the controller for each switch, which is then used to run a separate instance of the MAB algorithm. The match-field selection process is discussed further in Section **??**. The *flow-rule aggregation module* is used to aggregate the flow-rules at the switches according the particular match-field combination returned by the MAB module. It is also responsible for interacting with the flow-rule module of the SDN controller to update, modify and delete the flow-rules as necessary. The details of the flow-rule aggregation process are presented in Section **??**.

B. Flow-rule Aggregation for SDIoT networks

Many of the advantages of SDIoT networks, such as per-flow differential treatment depending on the type of IoT traffic [**?**], and flow-scheduling for mixed-criticality applications in cyber-physical systems [**?**], utilize the fine-grained match-action capabilities of SDN. This generates a huge number of flow-rules, and the limited flow-rule capacity of SDN switches leads to scalability issues in SDIoT networks. Thus, flow-rule aggregation plays an important part in improving the scalability of SDIoT networks, by increasing total number of flow-rules that can be accommodated at the switches. Many existing flow-rule aggregation schemes considered network-wide routing and rule-placement strategies [**?**], [**?**], [**?**]. Therefore, they assumed complete control over forwarding by a single application, which is not the case for SDIoT networks, where multiple IoT applications may be running at the SDN controller, with their own forwarding policies. The proposed match-field selection and flow-rule aggregation strategy works on top of existing forwarding policies (by different IoT applications), and is thus applicable to SDIoT networks.

C. Flow-table Representation

Let $\mathcal{S}$ denote the set of SDN-enabled switches in the SDIoT network. The switches communicate with the logically centralized SDN controller through the OpenFlow protocol [**?**], and the traffic forwarding is controlled by placing forwarding rules in the flow-table at the switches. Let $\mathcal{R}$ denote the flow-table at a switch. A flow-rule $r \in \mathcal{R}$ is given as $r = \langle \mathcal{M}_r, \mathcal{O}_r, \mathcal{C}_r \rangle$, where $\mathcal{M}_r$, $\mathcal{O}_r$ and $\mathcal{C}_r$ represent the set of match-fields, output action, and flow-counters, respectively. The match-field set, $\mathcal{M}_r$, is given as $\mathcal{M}_r = \{m_r^k \mid k = 1, 2, .., n\}$ where n is the number of match-fields.

[2]https://github.com/noxrepo/

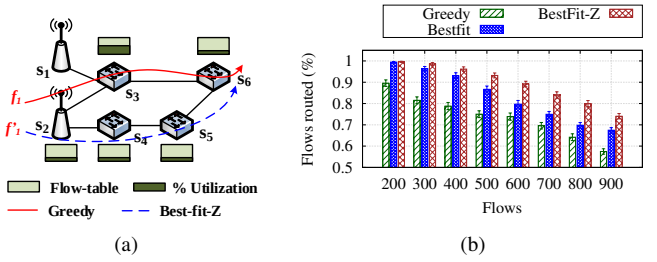
IV. QoS-aware Flow-rule Aggregation

A. Path Selection for Minimizing Flow-table Utilization

In this Section, we present a path selection heuristic to minimize the flow-table utilization in the network. As IoT flows arrive sequentially in the network, multiple paths may be present that satisfy the QoS requirements of each flow (*candidate* paths). A greedy approach of picking the QoS path with the minimum number of flow-rule insertions may lead to faster fill-up of a switch having a high degree (*bottleneck switch*) in the network graph. To address this, we propose the Bestfit-Z heuristic, as presented in Definition **??**.

Definition 1. Bestfit-Z heuristic: Given a set of candidate paths, $\{P_l\}$, choose the path $P_{bfz} = \arg\min_l \delta(P_l)$, where $\delta(P_l)$ represents the cost of a particular path P_l in terms of flow-rule utilization, and is given as $\delta(P_l) = \sum_{i \in P_l} \theta_i |\mathcal{R}_i|/|\mathcal{R}_{max}|$ where $|\mathcal{R}_{max}|$ is the maximum number of flow-rules that can be placed in the flow-table, and $|\mathcal{R}_i|/|\mathcal{R}_{max}|$ denotes the flow-table utilization at switch $i \in P_l$. The term $\theta_i = i^{-\gamma}/\sum_{j \in \mathcal{S}} j^{-\gamma}$ is a decreasing vector that assigns importance (or rank) to each switch based on its flow-table utilization, according to a Zipf distribution, where γ represents the skewness of the distribution flow-table utilization across all switches.

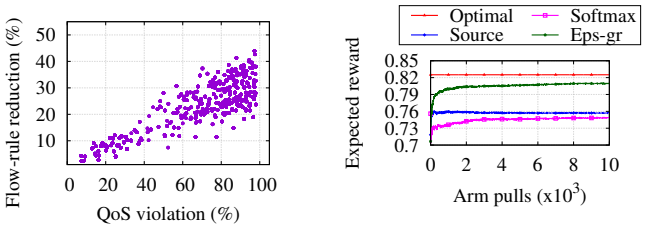
The Bestfit-Z heuristic takes into account all the bottleneck switches ranked according to the severity of the bottleneck (flow-table utilization), which leads to amelioration of the bottleneck switches, and increases the number of flows successfully routed in the network.



(a) (b)

Figure 4: Figure showing a) toy example illustrating the operation of the Bestfit-Z heuristic, and b) relative performance of the Bestfit-Z heuristic with increasing number of flows

Figure **??** shows an illustrative example of the Bestfit-Z heuristic. The switch s_3 lies on multiple paths, thus, its flow-table gets filled up faster. If the bottleneck switch, s_3, is completely filled, new flows arriving at that switch will be dropped. Consequently, all paths with s_3 as an intermediate switch will become invalid, leading to sub-optimal performance. The greedy approach chooses path f_1 with three flow-rule insertions at s_2, s_3 and s_6. On the other hand, the Bestfit-Z heuristic takes into account the bottleneck switch, s_3, and chooses path f_1' with four flow-rule insertions at s_2, s_4, s_5 and s_6. In our earlier work [**?**], we presented the simpler Bestfit heuristic, which accounted for only one bottleneck switch on a given path. Figure **??** shows the relative performance of the Bestfit-Z heuristic with increasing number of flows. From the figure, it is evident that the proposed BestFit-Z minimizes the



(a) (b)

Figure 5: Figure showing a) trade-off between flow-rule reduction and QoS violations for different combinations of match-fields, and b) comparison of different bandit algorithms and exhaustive search

flow-table utilization of the network, and is hence able to successfully route a larger number of flows compared to Greedy and Bestfit. In particular, from the figure, we observe that due to the consideration of multiple bottleneck switches, the relative performance of Bestfit-Z improves with increase in the number of flows. Overall, the Bestfit-Z heuristic outperforms the Bestfit and Greedy heuristics by 7% and 20%, respectively. For a given set of paths $\mathcal{P}$, calculating $\delta(P_l)$ involves iterating over at most $\max_l |P_l|$ switches. Assuming that there are k candidate paths for a flow, the time-complexity for the Bestfit-Z heuristic is given as $O(k \max_l |P_l|)$. The Bestfit-Z heuristic utilizes the distribution of flow-table utilization (γ) at the switches, which may be found by periodically querying the number of flow-rules present. In OpenVSwitch[3], this can be done using the *dump-flows* command.

B. Match-field Selection for Flow-rule Aggregation

Using flow-rule aggregation, the reduction in the number of flow-rules and the number of QoS violated flows depends on the particular combination of match-fields chosen. Given a set of n match-fields, for each field, we can choose to either include it or not (wildcard), leading to 2^n possible combinations. However, for every combination, there exists a fundamental trade-off between flow-rule reduction and QoS violations, as shown in Figure **??**. The figure shows a scatter-plot of flow-rule reduction and QoS violated flows obtained by applying flow-rule aggregation using the 2^n different combinations of match fields where $n = 10$. Thus, our goal is to choose a particular combination of match fields which achieves a favourable trade-off amongst these two criteria. Since n is a constant, thus, for a fixed set of flow-rules, we can perform an exhaustive search in polynomial time to find the optimal arm which achieves the best (in terms of some utility) trade-off between flow-rule aggregation and QoS violations. However, in an online model, where IoT flows arrive sequentially, the distribution of utility associated with a particular match-field combination is not known in advance. Therefore, match-field combinations have to be chosen with incomplete information such that over time, the expected utility achieved is close to that of the optimal arm in the offline case. This problem can be conveniently modeled as a MAB problem.

[3]https://www.openvswitch.org/

Multi-arm bandits are instances of sequential decision-making problems that deal with the *exploration-exploitation* trade-off, i.e., the balance between exploiting alternatives (arms) that performed well in the past and arms that may yield higher utility in the future. In the flow-rule aggregation problem, in each iteration, each of the 2^n combinations of match-fields yields some utility (in terms of flow-rule reduction and QoS-violated flows) depending on the traffic. However, in each iteration, only one combination of match-fields may be chosen for flow-rule aggregation, and its utility observed. The goal is to choose, in each iteration, the combination of match-fields, which maximizes the cumulative sum of utility over time.

1) Design of Utility Function: Let $\mathcal{A}$ denote the set of 2^n match-field combinations (arms)[4], where n is the number of match-fields. Let R_a and F_a denote the reduction in the flow-table length and the number of correctly forwarded flows, when aggregation is done with arm $a \in \mathcal{A}$. Here, the term *correctly forwarded flows* denotes the flows forwarded by the aggregated flow-table, which are consistent with the forwarding policies before aggregation. Since the objective is to minimize the number of flow-rules (i.e., increase the reduction in flow-table length) as well as to reduce the number of QoS-violated flows (i.e., increase the number of correctly forwarded flows), we design a utility function $\mathcal{U}_a$ as:

$$\mathcal{U}_a = \alpha \hat{R}_a + (1-\alpha)\hat{F}_a; \quad \alpha = \frac{e^{\frac{|\mathcal{R}|}{|\mathcal{R}_{max}|}} - 1}{e-1} \tag{1}$$

where $|\mathcal{R}|/|\mathcal{R}_{max}|$ denotes the flow-table utilization, and $\hat{R}_a$ and $\hat{F}_a$ denotes the variables R_a and F_a normalized in $[0, 1]$. The parameter α in Equation (**??**) controls the relative importance of flow-rule reduction and number of correctly forwarded flows. The weight parameter α is designed to take into account the flow-table utilization at the switches. This ensures that when flow-table utilization is low ($|\mathcal{R}|/|\mathcal{R}_{max}| \to 0$), greater importance is placed on correctly forwarding flows. On the other hand, at higher flow-table utilization, flow-rule aggregation is given priority to improve the overall performance.

Algorithm 1 Algorithm for Utility Calculation

Inputs: Flow-table $\mathcal{R}$, arm $a \in \mathcal{A}$
Output: Calculated utility $\mathcal{U}_a$;

```
Initialize dictionary d to store aggregation keys;
for each rule r in R do
    Extract aggregation key λ ← {m_r^k | k ∈ a};
    if λ ∉ d then
        Add λ to d and initialize array out_count_λ;
        Increment out_count_λ[O_r] by 1;
    else
        Increment out_count_λ by 1;
Get flow-rule reduction R_a ← |R| − |d|;
Get correct forwarding F_a ← Σ_λ max out_count_λ;
Calculate reward U_a from R_a and F_a using Equation (??);
```

Algorithm **??** is used to calculate the utility $\mathcal{U}_a$ for a given arm $a \in \mathcal{A}$. Step **??** extracts an unique *key* λ from exact-match rule r, where λ contains values of r in only those match-fields given by arm a. Thus, rules having same values of λ are essentially aggregated. Figure **??** illustrates how λ is extracted from exact-match rule r. For each key λ, an array out_count_λ keeps count of the number of times a particular port is used for that key. Steps **??** to **??** are used to select the most used port ($\max out\_count_\lambda$) for each key λ, as the default port. If ports other than the default port are used, flows are forwarded sub-optimally (i.e., output port changed due to aggregation) which may lead to QoS-violations. Step **??** is used to calculate the number of correctly forwarded flows.

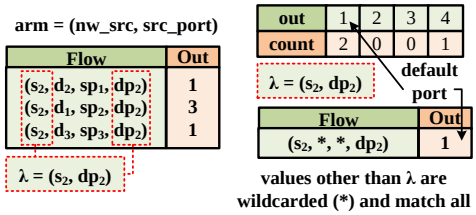


Figure 6: Illustrative example showing extraction of key λ_r and aggregation using default port

2) Multi-Arm Bandit: MAB algorithms specify a strategy for choosing an arm at each iteration, which maximizes the total utility over time. We compare the results of two popular MAB algorithms — Epsilon-Greedy and Softmax, with random aggregation, source-based aggregation [**?**], [**?**] and exhaustive search (optimal). In the Epsilon-Greedy algorithm, with each iteration, the algorithm keeps track of the current average utility of each arm. It selects the arm with the highest average utility with probability $1 - \epsilon$ and does a random exploration of the other arms with probability ϵ. We use an annealing version of the Epsilon-Greedy algorithm in which ϵ decreases over time [**?**]. In the Softmax algorithm, the probability of choosing an arm is dependent on the utility achieved by picking that arm, following a Boltzmann distribution. Figure **??** shows the expected reward achieved by the different algorithms with an increasing number of iterations (arm pulls). In each iteration, utility achieved is given by Algorithm **??** using the flow-table at that instant, and the arm selected for that iteration. From the figure, we observe that with an increasing number of iterations, the Epsilon-Greedy (Eps-gr) method outperforms all the other algorithms and achieves performance close to the exhaustive search (optimal).

Using the MAB algorithms, flow-rule aggregation can be done with the different arms chosen at each iteration, and over time, the total utility is close to that of the *offline* exhaustive search. In the experiment shown in Figure **??**, the optimal arm chosen by the exhaustive search is (dl_dst, dl_type, dl_vlan, nw_dst). For our experiments, we choose the Epsilon-Greedy algorithm due to its superior performance over other MAB algorithms. In each iteration, Epsilon-Greedy chooses a random arm with probability ϵ, and arm $a = \arg\max_{a \in \mathcal{A}} \mathcal{U}_a$ with probability $1 - \epsilon$. The MAB algorithm spawns a thread at the controller for each switch. Iterating the MAB algorithm too often causes high load on the controller; on the other hand, iterating infrequently causes greater deviation from the optimal. In our experiments, we have found iterating every

[4]Henceforth, we use the terms *arms* and *match-field combinations*, interchangeably.

50 ms to give good results. It is noteworthy that other MAB algorithms can easily be used in place of Epsilon-Greedy simply by using Algorithm **??** to calculate the utility or reward at each iteration.

C. Flow-rule Placement and Aggregation

In this Section, we present the reactive flow-rule placement scheme, which utilizes the match-fields (arms) yielded by the MAB algorithm (Section **??**), to aggregate flow-rules.

Algorithm 2 Algorithm for Flow-rule Placement

Inputs: Packet-in message, M, dictionary d, arm $a \in \mathcal{A}$
Output: Flow-rule placed (aggregated if possible)

1: Get exact-match rule r from M;
2: Extract key $\lambda \leftarrow \{m^k \mid k \in a\}$ from r;
3: **if** $\lambda \notin d$ **then** ▷ Exact-match absent
4: Place exact-match rule r in switch;
5: **else** ▷ Exact-match present
6: Find the most used port $\mathcal{O}^* \leftarrow \max out\_count_\lambda$;
7: **if** $\text{count}(\mathcal{O}^*) \geq 1$ **then** ▷ Aggregation possible
8: Create wildcard rule r_w using λ and $\mathcal{O}^*$;
9: Place r_w in switch with higher priority than r;
10: **else**
11: Place exact-match rule r in switch;

Algorithm **??** presents the reactive flow-rule placement scheme, which places flow-rules (aggregated if possible), upon arrival of a packet-in message. Step **??** is used to get an exact-match flow-rule r from the packet-in message M. Step **??** is used to extract a unique key λ from exact-match rule r, as shown in Figure **??**. Steps **??** to **??** are used to aggregate the flow-rules using the most used port (default port), when more than one exact-match flow-rule is present for a particular key λ. Figure **??** shows an illustrative example of the aggregation process, where exact-match flow-rules are aggregated using a wildcard rule r_w, created from r, which wildcards[5] all the match-fields $\notin \lambda$, and sets the output action as the most used port $\mathcal{O}^*$. In line **??**, we choose to place the wildcard rule with higher priority, so that the existing flows switch to using the wildcard and the exact-match rules can be deleted automatically following the idle-timeout of the OpenFlow protocol, thereby reducing the number of flow-rules. In our earlier work [**?**], we proposed a similar flow-rule aggregation scheme. In the present work, we have updated the scheme to aggregate flow-rules using the most used port $\mathcal{O}^*$, which leads to an overall 8% increase in utility. It is noteworthy that aggregation using the most used port $\mathcal{O}^*$ with a higher priority may cause sub-optimal forwarding for some flows; however, the arm $a \in \mathcal{A}$ is chosen in a way that such cases are minimized.

In Algorithm **??**, the most expensive step in terms of time-complexity is finding the most used port $\mathcal{O}^*$. In the worst case, there may be $p-1$ different output actions for a key λ, before an action is repeated, where p is the number of output ports. Thus, iterating p times to find the most used port $\mathcal{O}^*$, leads to a time-complexity of $O(p)$ for Algorithm **??**. Algorithm **??** can easily be extended to newer versions of OpenFlow, with multiple pipelined flow-tables, by extending the number of output actions to include instructions such as *goto-table*. With T pipe-lined stages, there will be $T-1$ *goto-table* instructions, which increases the time-complexity of Algorithm **??** to $O(p+T)$.

[5]A wildcard (x) in a match-field implies that it matches all flows.

V. Performance Evaluation

Table I: Simulation parameters

Parameter	Value
Topology	AttMpls , Goodnet [**?**]
Number of switches	25 (AttMpls), 17 (Goodnet) [**?**]
Number of links	57 (AttMpls), 31 (Goodnet) [**?**]
Avg. packet size	$94 - 699$ bytes [**?**]
Active volume	$142 - 27,716$ bytes [**?**]
Mean rate	$562 - 516,540$ bps [**?**]
Active time	$1 - 34$ s [**?**]
OpenFlow rule-timeout	10 s

We evaluated the performance of the proposed scheme using the POX SDN controller[6] and the Mininet network emulator[7]. The experiments were carried out on a Intel i7 2.7 GHz PC with 8 GB RAM, running Linux kernel 4.15. The different parameters considered for the experiments are presented Table **??**. For evaluation, we considered two topologies— AttMpls topology and Goodnet topology from the Internet Topology Zoo [**?**]. We used the D-ITG traffic generator to model IoT traffic flows based on real traces in [**?**]. The performance of the proposed scheme was evaluated by sending IoT traffic though the SDN network over a period of 5 minutes with average packet size, mean rate, active volume, and active times as given in Table **??**. The flow-rules are generated by the POX controller using the OpenFlow protocol v1.0 on sending traffic through the network. Each experiment was repeated 30 times and the results show the average value along with the 95% confidence interval.

To show the effectiveness of the proposed scheme, we compare the proposed scheme, Q-Flag, with the following existing baselines discussed in Section **??** — delay-based flow-aggregation scheme (DBA) [**?**] and flow-table sharing scheme (FTS) [**?**]. In the DBA scheme, flows having the same QoS path are aggregated using nw_src and nw_dst. In the FTS scheme, when the flow-table is full and a table-miss occurs, the packet is randomly forwarded to a next hop neighbor instead being sent to the controller as a packet-in message.

We consider the following performance metrics to evaluate the performance of the proposed scheme — a) average flow-rule utilization across all switches $(1/\mathcal{S}\sum_i |\mathcal{R}_i|/|\mathcal{R}_{max}|)$, b) the number of packet-in messages received at the SDN controller, c) average end-to-end delay experienced by flows in the network, and d) the number of flows that violate QoS requirements of IoT traffic due to sub-optimal forwarding.

A. Results and Discussion

1) Analysis of flow-rules: Figure **??** shows the variation in flow-rule utilization with an increasing number of flows. From

[6]https://github.com/noxrepo/
[7]http://mininet.org/

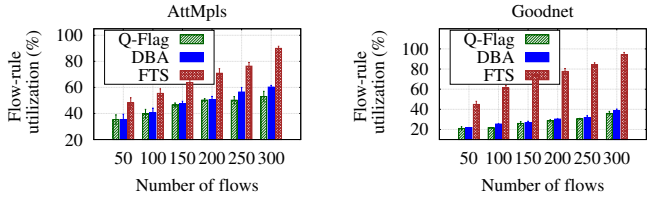


Figure 7: Variation in flow-rule utilization in the network with number of flows (averaged across all switches)

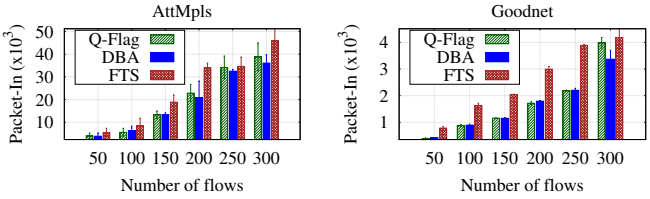


Figure 8: Variation in the number of Packet-In messages in the network with number of flows

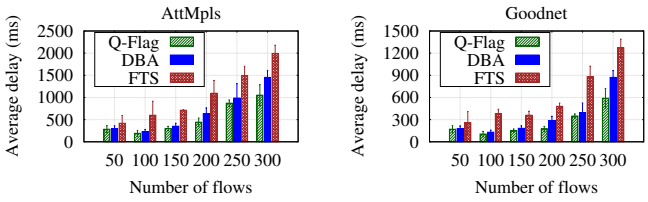


Figure 9: Variation in average delay in the network with number of flows

the figure, we observe that on average, Q-Flag achieves 5% and 30% (AttMpls), and 7% and 60% (Goodnet) reduction in flow-rules compared to the DBA and FTS schemes, respectively. FTS does not consider rule aggregation and instead avoids bottleneck switches by using random forwarding to achieve uniform flow-table utilization. With a large number of low-volume IoT flows, the exact-match strategy followed by FTS leads to higher average flow-table utilization. DBA utilizes aggressive aggregation by source-destination pair, which leads to an appreciable reduction in flow-rules. On the other hand, Q-Flag also takes into account the impact of aggregation on QoS of newly arrived flows and hence is not able to achieve significant improvement over DBA. The slight improvement is due to the combined effect of — a) suitable match-field selection, and b) Bestfit-Z heuristic.

Flow-rule aggregation is more apparent in the Goodnet topology than in AttMpls. Goodnet is relatively sparse compared to AttMpls, providing fewer options while choosing QoS paths, which in turn increases the opportunities for flow-rule aggregation. Overall, in both the topologies, we see that Q-Flag outperforms the existing schemes in terms of flow-rule aggregation. This increases the total number of flows that can be accommodated in the network and thus, increases the scalability of SDIoT. Moreover, from the figure, we observe that the proposed scheme performs well for generic network topologies.

2) Analysis of packet-in messages: Figure **??** shows the variation in the number of packet-in messages received at the POX SDN controller, indicating the overall control-plane load. From the figure, we observe that, on average, Q-Flag achieves a reduction of 19% (AttMpls) and 34% (Goodnet) in the number of packet-in messages compared to FTS. FTS focuses on minimizing packet-in by circumventing the deletion of long-lived active flows, thus preventing repeated packet-in for an ongoing flow. Low-volume intermittent or bursty IoT flows do not benefit as much from this strategy and incur greater packet-in due a higher number of flows rather than deletion of flow-rules associated with an ongoing flow. On the other hand, Q-Flag causes a 5% increase in packet-in messages compared to DBA. Since DBA uses aggressive aggregation by only two match-fields, its corresponding flow-rule matches are *wider* than that of Q-Flag. Therefore, more flows are directly routed through the data-plane without arriving at the controller, thereby reducing packet-in.

3) Analysis of average delay: Figure **??** shows the average delay experienced by flows in the network. From the figure, we observe that overall, Q-Flag achieves 20%, 50% (AttMpls), and 24%, 57% (Goodnet) reduction in delay compared to the DBA and FTS schemes, respectively. FTS leads to increased packet-in for IoT traffic as discussed above (Section **??**). For each packet-in, a corresponding flow-rule is installed only after which packets can be forwarded. Each flow-rule installation incurs a *flow-setup* delay in the range of $3-8$ ms [**?**]. Therefore, a large number of packet-in messages leads to increased flow-setup delay, which in turn increases the average delay in the network. On the other hand, DBA may take sub-optimal forwarding decisions for newly arrived flows due to aggressive aggregation using source-destination pair, leading to increased delay in the network. Overall, Q-Flag outperforms the existing schemes in terms of average delay, and is thus beneficial for many emerging IoT applications such as connected vehicles, industrial IoT, and augmented reality, which are *latency-critical*. Reduction in the delay is also beneficial for non-critical IoT applications, such as turning on a smart-bulb, where response time should be on the order of a few hundred milliseconds for optimal user experience [**?**].

4) Analysis of QoS-violated flows: Figure **??** shows the percentage of flows that violate the QoS. From the figure, we observe that overall, Q-Flag reduces the QoS-violated flows in the network by 31%, 41% (AttMpls), and 29%, 38% (Goodnet) compared to the DBA and FTS schemes, respectively. In DBA, many newly arrived flows match with the existing source-destination wildcards without generating packet-in messages and contacting the controller. This condition causes sub-optimal forwarding decisions, which leads to increased QoS violations. FTS incurs increased QoS violations mainly due to — a) a large number of packet-in messages causes increased delay, and b) the randomized forwarding action in the data-plane, causing packets to travel along sub-optimal paths. From the figure, we note that the relative performance of Q-Flag is better in AttMpls, where the dense topology offers many routing alternatives, and hence greater chances to forward traffic sub-optimally due to aggregation. However, overall, Q-Flag reduces the total QoS violations in the network. Along

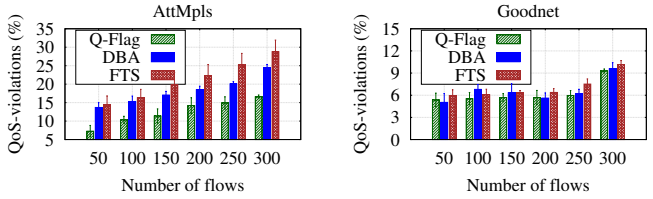


Figure 10: Variation in QoS-violated flows in the network

with delay-constrained applications, this is also useful for many non-critical IoT applications that use application layer re-transmissions[8]. Flows that violate the QoS may need some packets to be re-transmitted, leading to increased energy consumption at battery-operated IoT devices.

From the analysis above, we see that the proposed scheme, Q-Flag, reduces the average delay and QoS violations in the network and is thus useful for both latency-critical and non-critical IoT applications. Further, it is capable of reducing the number of flow-rules at the switches, which helps in accommodating more flows in the network, and thus, increases the scalability of SDIoT.

VI. Conclusion

In this paper, we proposed a QoS-aware flow-rule aggregation scheme for SDIoT networks to address the flow-rule capacity problem of SDN switches. The proposed scheme utilizes a two-fold approach to aggregate flow-rules, while considering the QoS of heterogeneous IoT traffic. First, a path-selection heuristic is used to increase the total number of flow-rules that can be accommodated in the network. Second, a multi-arm bandit (MAB)-based rule-aggregation scheme is used to aggregate flow-rules using a suitable combination of match-fields, while minimizing its impact on the QoS of newly arrived IoT flows. Experimental results using IoT traffic showed that the proposed scheme is capable of reducing the average delay as well as the number of QoS violations in the network, making it suitable for both delay-critical as well as non-critical IoT applications.

The flow-rule aggregation approach considered in this work reduces the ability of the SDN controller to effectively collect network statistics, which may adversely affect SDN-based schemes that deal with network security. Therefore, in the future, we plan to extend this scheme with a suitable network statistics collection framework.

Acknowledgements

A preliminary version of this work has been published in IEEE GLOBECOM, 2018, Abu Dhabi, UAE (December 9 – 13, 2018), DOI: 10.1109/GLOCOM.2018.8647471. This work was partly supported by IMPRINT-II project (Sanction no. SERB/F/12680/2018-2019;IMP/2018/000451 , Dt. 25-03-2019), which the authors gratefully acknowledge.

[8]Popular IoT protocols such as Constrained Application Protocol (CoAP) use application-layer retransmissions for reliability.