An Enhanced Flow-based QoS Management within Edge Layer for SDN-based IoT Networking

Avewe BASSENE, Bamba GUEYE

Université Cheikh Anta Diop, Dakar, Senegal
avewe.bassene@ucad.edu.sn
bamba.gueye@ucad.edu.sn

Abstract. *IoT* infrastructure makes great demands on network control methods for an efficient management of massive amounts of nodes and data. This network requires fine traffic control management to ensure an adequate *QoS* for data transmission process, especially in a low-cost network that covers smart territories deployed in so-called "technological lag" areas. Software-Defined Networking (*SDN*) enables to handle dynamically network traffic as well as flexible traffic control on real-time. However, *SDN* technology exhibits several issues with regard to additional processing time or loss that are associated to control plan. These factors can lead to performance degradation of the *SDN* control traffic flows within data plane which is not tolerated in medium/low capacity *IoT* environment.
This paper proposes an Enhanced Flow-based *QoS* Management approach, called *EFQM*, that reduces spent time within control plane as well as uses *SDN* controller either to reduce loss or to optimize bandwidth according to flows latency and bandwidth requirement. Our experimental results show that *EFQM* outperforms *AQRA* in terms of response time and packet loss rate. Furthermore, by considering a default routing and delay as metrics, *EFQM* improves the average end-to-end flow performance by 7.92% compared to *AQRA*. In addition, *EFQM* enhances end-to-end flow performance by 21.23% and 23.52% compared to *AQRA* respectively according to delay and packet loss rate. The measured *EFQM* runtime is 23.29% shorter than *AQRA*.

Keywords: Edge Computing, Internet of Things, Quality of Service, Software-Defined Networking, Performance.

1 Introduction

Recent years, Africa has registered many *IoT* environment projects that plane to develop by rapidly reducing the technological divide that affects the continent. This environment is well known according to the huge and various volumes of generated traffic. *IoT* networks are equipped by a large number of sensors, and thus, they should be managed efficiently by network operators [1] [2].

Software Defined Networking (*SDN*) is a constantly progressive technology that offers more flexible programmability support for network control functions and protocols [3]. *SDN* provides logical central control model for implementation and maintenance of programmable networks. *SDN* decouples data and control plane over a well-marked and comprehensible controlling protocol like "*OpenFlow*" [4]. *OpenFlow* acts as de facto signaling protocol between control and data planes that are used to program *SDN* switches. By decoupling control and data planes, *SDN* technology enable to monitor network conditions and network resource allocation on the fly. Therefore, *SDN* is amongst the key enabling technology for new generation networks.

Congestion is often the most used criterion to improve network performance in *IoT* environments. Indeed, from *SDN* control plane, congestion management makes it possible to improve the network Quality of Service (*QoS*) by optimizing traffic important factors such as delay, loss, bandwidth, etc. In addition, *IoT* networks are reputed for their non-compliance according to fixed standards (for instance, protocols and ports used). As consequence, it is not just sufficient to give a good QoS-aware approach by just reading such an instable traffic characteristics. Furthermore, with respect to a real-time QoS-aware study that incorporating *SDN* technology, it is mandatory to take into account traffic characteristics. In fact, selected parameters include both information coming from external entity to which device is connected (*IoT* server) and current traffic data *QoS* requirements in terms of delay, bandwidth and loss recorded from different architecture layers.

Previous work like *AQRA* [5] aims to guarantee adaptive multiple *QoS* requirements of high-priority *IoT* applications by dropping low/medium priority flows that seize the network resource of high-priority flows until the *QoS* requirements can be guaranteed. However, this removing operation is not trivial since it leads to longer transmission delay and processing overhead at the *SDN* switches. Furthermore, the end-to-end traffic *QoS* management as described in [5] can be improved by reducing packets disruption at the edge layer and transposing the optimization factors lower in network architecture. Indeed, this improvement can decrease loss rate, avoid congestion and consequently increase the network scalability to adapt it to different environment devices ability.

Therefore, this paper aims to reduce processing latency due to *SDN* switches transmission disruption, which leads to packets lost and a delay extension. In fact, the obtained network degradation is caused by "*Flow_mod*" rules sent from the *SDN* controller [6] and can lead to mighty waste time (up to 64 *ms* in normal operation, when changing paths occurs). Starvation problem is considered.

In addition, according to 3*GPP Long Term Evolution* (*LTE*), each bearer has a corresponding *QoS* class identifier (*QCI*), and each *QoS* is categorized by service type, priority, packet error rate (*PER*) and packet delay budget (*PDB*) [7]. Some flows have *QCIs* vector that allow a low *PDB* values. Avoiding the transfer of such packets to the control layer could considerably reduce latency or otherwise (allowing them) can be effective for bandwidth and loss sensitive flows.

The rest of the paper is organized as follows. Section 2 reviews related works. Processing delays, bandwidth and loss impact in SDN switches and the multi-layer traffic flow operations are discussed in section 3. Our $EFQM$ SDN-based framework from perception to network layers is described in Section 4. Section 5 evaluates $EFQM$ overall performance. Finally, Section 6 concludes this paper.

2 Related work

Various motivations have led to numerous proposals on IoT networking QoS improvements in SDN-based network architecture. Most of related work particularly focus on algorithmic optimization which can give an effective approach to overcome QoS problem in IoT environment.

Deng et al. [5] propose $AQRA$ for SDN-based IoT network to fulfill a multi-QoS requirement of high-priority IoT application. The key idea is to remove low or medium priority flows in favor of high priority flows until QoS requirements can be guaranteed. However, frequent deletion of flow causes traffic loss in current SDN hardware switches when currently active traffic flow is modified during ongoing traffic transmission. The deletion operation adversely impacts in the end-to-end transmission delay performance and packet loss rate. This action requires processing overhead at the hardware switch ($i.e.$ $TCAM$ reordering [8]) and it is new type of traffic disruption that is not currently handled by SDN switches [6].

X. Guo et al. [9] present $DQSP$ an efficient QoS-aware routing protocol with low latency and high security. $DQSP$ is one of the widely-used deep reinforcement learning method that combines $DDPG$ algorithm and centralized control characteristics of SDN-based IoT network. $DQSP$ outperforms the traditional $OSPF$ routing protocol in term of delay especially when network is under attack.

Authors in [10] propose SDN-based framework to fulfill IoT service QoS requirements. It consists of finding shortest path with minimum-delay and maximum-bandwidth for delay/bandwidth-centric traffic. It decomposed problems into server selection problem and path selection problem which are implemented in controller as QoS-aware route and least-load IoT server modules. The proposed framework achieves high throughput and low delay. Nevertheless, the authors just considered two metrics which are not discriminator.

According to $PFIM$ [7], the authors proposed a pre-emptive flow installation mechanism for IoT devices. It can learn the transmission intervals of periodic network flows and install the suited flow entry into SDN switches before packets arrival. However, they considered only delay metric.

The authors of [11] describes an admission control approach called $REAC$ which can control traffic flows. Indeed, the edge router monitors the delay performance to admit flows to the network that guarantees good quality for high-priority flows. However, we only considered a single QoS requirement based on delay. In addition, the same authors do not consider the starvation problem of low priority flows.

Deep packet Inspection (*DPI*) is use in [12] to improve *QoS* for certain network traffic. DPI-based traffic classification is used with current port and queue capacity from utilization monitor for a network flow routing decision based on *DiffServ* for *QoS* and *multi-path* for load balancing. However, the proposed model increases runtime delay due several initial packets traffic duplication from the ingress port. In addition, it needs additional data plane entities and only treat two metrics (delay and throughput).

Finally, a fog computing with heuristic algorithm of lower complexity is proposed in [13] in order to provide a low cost and QoS-aware *IoT* infrastructure. However, *IoT* end devices must support a specific functions, for instance, act as gateway.

In contrast, to previous studies, *EFQM* promotes several metrics in order to cover a wide performance of *IoT* traffic characteristics as well as limits flow deletion process by fixing different sorting levels.

3 Brief overview on considered SDN-based QoS problem

LTE is an end-to-end *IP* network that provides *IP* connections from the terminal to the core network. *QoS* implies services to be differentiated based on the *QCI* which determines the priority level of each service class and specifies the maximum one way allowed values in terms of *delay*, *jitter*, and *packet loss* [14]. Nevertheless, complexity residing in such data leads to increased processing operation. This cause disruption in network traffic that directly affect bandwidth, delay and loss sensitive services due to networks bottleneck.

According to hierarchical network, the backhaul portion of the network comprises the intermediate links between the core network and the small sub-networks at the edge of the network. Since *LTE* architecture is designed to support high data traffic and a guaranteed *QoS* to end-to-end *IP* based service [15], we believe that network degradation can be considerably limited with fine grained low levels traffic managing, *i.e.* portion between network and perception layer which is often subject to local control. In addition, a local performance management adapted to the quality of the network, to the cost of equipment and to local available resources would give more scalability and cost adaptability to the proposed model. Thus, a fine grained *QoS* management at edge and control layers is proposed to effectively improve end-to-end transmission performance.

In section 4, we describe SDN-based *EFQM* framework and its operation in these specific network areas. Let's first exploit the SDN-based *IoT* network architecture and explain the current state of addressed problem.

3.1 SDN-based IoT network architecture

The SDN-based *IoT* network architecture is composed by five layers. Each layer, according to specific embedded components, ensures communication with adjacent levels components (highest, lowest and centralized control equipment). Fig.

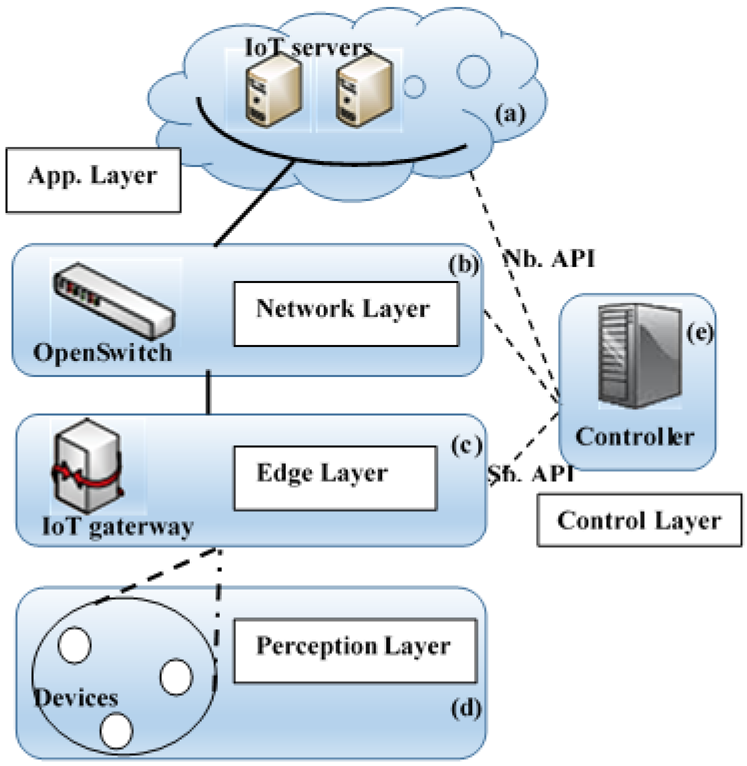


Fig. 1. Network architecture

1 presents our architecture where the network layer consists of a set of programmable devices that perform packets forwarding towards data plane.

The control plane (*Controller*) is the component that caries communication between other equipment via a dynamic routing protocol. The main goal of the controller is to tell to the second major component (*dataplane*) of the network how to process each incoming frame/packet/dataset using "*OpenSwitches*".

The hierarchical network architecture as illustrated in Fig. 1 is formed by:

1. "(a)" *Application layer*: contains *IoT* applications or services;
2. "(b)" *Network layer*: consists of set of *SDN OpenFlow* switches;
3. "(c)" *Edge layer*: consists of set of edge equipment (*APs*);
4. "(d)" *Perception layer*: *IoT* devices belong to this layer;
5. "(e)" *Control layer*: the control plane consists of the *SDN* controller which communicates with "(a)" through Northbound *API* (Nb. *API*) and with "(b)" and "(c)" through Southbound (Sb. API).

The edge equipment is *OpenFlow-enabled* so they can be controlled by the SDN controller using the *OpenFlow* protocol. This layer connects to "(d)" via wireless communication technologies. Devices in "(d)" forward/receive data to/from "(a)" by accessing layers "(c)" and "(b)".

The controller contains module named "topology discovery" which discovers all network elements in the data plane and builds real time network topology. Another module ("network status monitoring") monitors and collects the network condition periodically. The communication between the data plane and the controller uses a standardized *OpenFlow* protocol.

Furthermore, $EFQM$ Framework is able to manage the behavior of both $Open - switches$ and IoT gateway via southbound API. It can also receives messages from IoT servers via northbound API. However, a gateway has the possibility to decide whether it must route traffic to control plane or not.

3.2 Problem statement

When the IoT devices transfer the message from "*Perception layer*" to "*Application layer*", *Packet_In* message undergo a set of processing in each intermediate node before reaching their destination. These processes to ensure the optimal management of traffic for high performance level. Indeed, with advanced communication emergence devices, current networks should support several services such as video streaming, web browsing, online gaming, etc. These services that have different delay constraints, bandwidth and QoS requirements can cause network processing problems.

These problems often create network performance degradation which results in congestion at data plane equipment. Our aim is to overcome these constraints by ensuring that each packet fulfills all its QoS requirements from source to destination nodes. Therefore, a controller with a global and centralized network programmability view can give dynamic control flows and flexible network resource management which avoid IoT network contention and anomalies. In fact, most transport protocols only consider network congestion as a factor of traffic degradation, when adjusting end-to-end traffic behavior towards improving flow reliability.

However, it has been shown that traffic loss can occur in current SDN hardware switches when the forwarding rule being applied to a current active traffic flow is modified during ongoing traffic transmission [6]. It is attributed to the processing latency, which is the amount time we need in order to modify forwarding rule within a hardware switch. The obtained latency can cause transmission disruption that leads to packet loss for a transient period of time, as well as congestion due to the frequent recovery caused by these losses.

Relative to this last case, authors in [5] propose an approach that considers SDN controller *Flow_mod* message to remove low or medium priority flows which use network resource of high priority flows up to that the QoS requirements can be guaranteed. We think that, avoiding intentional flow deletion and reducing the controller computational overhead can improve existing approach. Indeed, Section 3.3 highlights a couple of issues according to $AQRA$.

The use of SDN technology could lengthen the processing delay for latency-sensitive packets or could be an improving factor for metrics such as loss or bandwidth. Otherwise, these metrics also is related to the QCI vector parameters assigned to each flow. Therefore, a suitable QoS-aware proposal must be approached from two main point of view: technological adhesion and real time traffic requirement. It is worth noticing that good performance could be achieved both in terms of end-to-end delay and runtime when a QoS-aware decision includes flow QCI vector parameters specification and overhead related to SDN technology adhesion.

3.3 *AQRA* drawbacks

The basic idea of $AQRA$ [5] in QoS-aware admission control is to remove low or medium priority flows which use network resources in favor of high priority flows until QoS requirements are guaranteed. This operation is not trivial since it causes both processing overhead with respect to SDN switches and delayed transmission. Indeed, when $Flow\_Mod$ messages are sent from controller to switches, a delete command for current flow rule F_c arrives to a switch. Afterwards, selected switch removes F_c and applies the next matching flow rule (F_{next}) to the current traffic. Subsequently, F_{next} aims to replace F_c to serve the current traffic after F_c deletion.

In fact, the operation consists of: *(i)* remove the current flow rule; *(ii)* replace current flow rule with respect to the next flow rule that fulfills the same criteria as the deleted one in order to preserve the current traffic which should not be used otherwise. Therefore, during the time between the corresponding flow rule searching and its application to current traffic, any other packet arriving at the current switch will be lost since the previously matching flow rule has already been deactivated.

In addition, the path change events are applied to all switches along dedicated path. In fact, the total flow transmission time grows with increases in the number of path change events which varies between 1 and 8. In regard to normal operation, path change causes disruption time for approximately 64 ms [6]. This leads to both a substantial transmission delay and congestion that overload the network traffic. This phenomenon can be even worse during a repetitive flow deletion as observed in [5]. The disruption time is also related to total transmission delay and runtime.

To overcome this problem, we directly send, according to the default algorithm, the high priority, loss-sensitive (to avoid traffic jams in the edge gateway) and delay-sensitive packets according to QCI values. Only medium and low priority packets will be transferred to $EFQM$ to ensure traffic QoS requirements. By so doing, we reduce network contention as much as possible for loss-sensitive and delay-sensitive flows that were directly sent.

4 *EFQM* SDN-Based framework

4.1 *EFQM* background

In contrast to previous studies like [5], [12], [16], $EFQM$ involves two major steps: a *simulated annealing* (SA)-based QoS routing and *Admission Control*

(AC). The general idea is to compute a QoS-aware best routing paths for each flow and then to control its admission by choosing path that fulfills traffic QoS requirement in a dynamic way at the controller. To reduce the controller workflow and improving delay, traffic classification is performed at the edge layer.

In fact, the edge layer is the first level of sieving in relation to our model. A classifier is used at IoT gateway. Therefore, a "(*classScpt*)" script, based on QCI vector parameters of each flow, figures out whether data packets should be rerouted under controller advices or not (*i.e.* default routing).

Table 1. Different classified classes in EFQM.

Classes	QCI Values
Prioritized	1, 2, 4, 5, 6
Non-Prioritized	3, 7, 8, 9

It is worth noticing that the shortest path routing (default forwarding) is a simple and fast packet forwarding protocol that always routes every traffic via shortest path, but lacks the sense of load balancing [17]. However, our traffic classification class takes into account this issue. Table 1 defines the classification model used by *classScpt* for each incoming flow. Two classes are defined: "*prioritized*" and "*non-prioritized*" classes.

The *classScpt* algorithm ensures that bandwidth sensitive flows does not compete bandwidth utilization and buffer resources with the small flows (*prioritized*) which can lead to loss. It ensures faster completion times and lower latency for time sensitive traffic while minimally impacting throughput. It is worth noticing that *classScpt* algorithm will be explained in section 4.3.

Note that the scheduling scheme presented here is a bearer class QoS control scheme. A bearer is a logical channel which establishes a connection between IoT server and $enodeB$. IoT devices may request many services having diverse QoS requirements according to a given time. Therefore, to distinguish between these different services, $3GPP$ defined the set of characteristics for 9 $QCIs$ as presented in [18].

Table 1 is specifically based on this standard. QoS requirements vector consists of different flow specification like QCI value (integer), priority, service type, PDB (ms), PER (between $10^{(-2)}$ to $10^{(-6)}$). QCI vector can be obtained from the $sFlow$ protocol [19] which provides the consumable resources statistics of IoT servers for the controller using the $sFlow$ Agent and the $sFlow$ Collector.

Unlike [5], [19], Table 1 is performed by taking into account values mentioned above since few applications can be delay-sensitive while having non-bandwidth guaranteed (service type = non-GBR). Previous works present acceptable loss rates ($PER = 10^{(-2)}$) and guaranteed bandwidth (service type = non-GBR). Since the default routing is moderately sensitive to load balancing, then the later type of traffic, in case of low PDB value, can be directly sent in order to avoid delay.

With respect to controller, we consider the following classifier classes according to a fine QoS-aware control admission. Therefore, IoT traffic can be grouped into 3 classes:

1. *Delay-centric* (D-centric): mission-critical or event-driven applications.
2. *Bandwidth-centric* (B-centric): associated with continuous traffic, (query-driven and real-time monitoring).
3. *Best-effort* (Be-centric): which consists of general applications such as non-real time monitoring.

According to this second class values and packets specification, chosen path, from all recorded ones must satisfy the traffic needs. Thus, this fine grained path selection also reduces harmful congestion within switches for flows directly sent (default route) from the IoT gateway. The overall system design and proposed controller architecture are illustrated respectively in Fig. 2 and Fig. 3.

Fig. 2 depicts 3 separate components distributed on two layers (edge and control layers). A heuristic algorithm called "*Simulated Annealing*" (SA) is used to find the approximate optimal solutions. According to edge layer, an IoT gateway is used to perform *classScpt* classification algorithm based on Table 1 entries. In regard to control plane, a *SA-based QoS* routing algorithm performs candidate paths selection with QoS constraints such as delay, bandwidth and packet loss rate. The appropriate path is finally chosen by an admission control algorithm according to current traffic load. In fact, path selection with multiple constraints in an IoT network communication is an *NP-complete* problem [10].

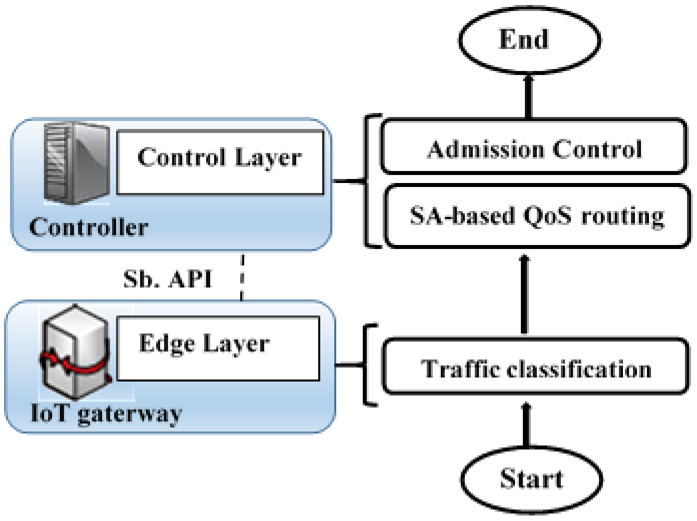


Fig. 2. *EFQM* at a nutshell.

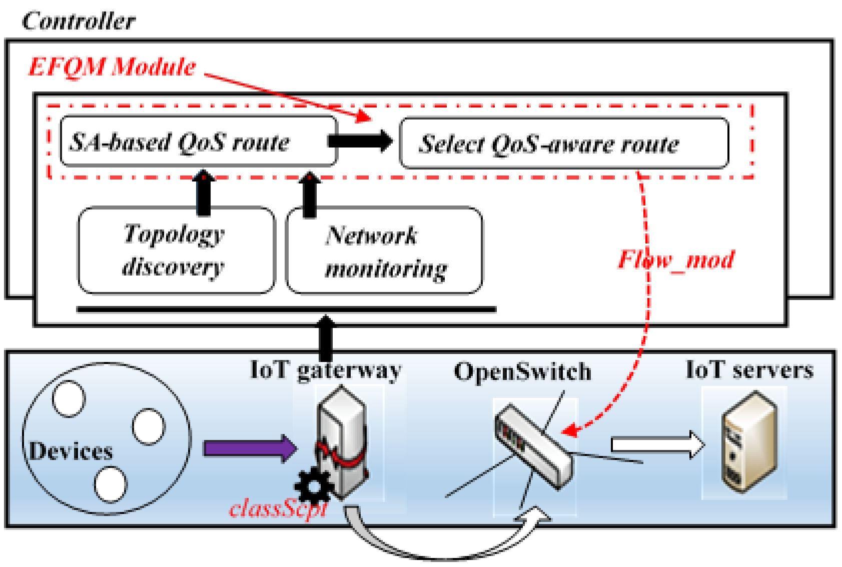


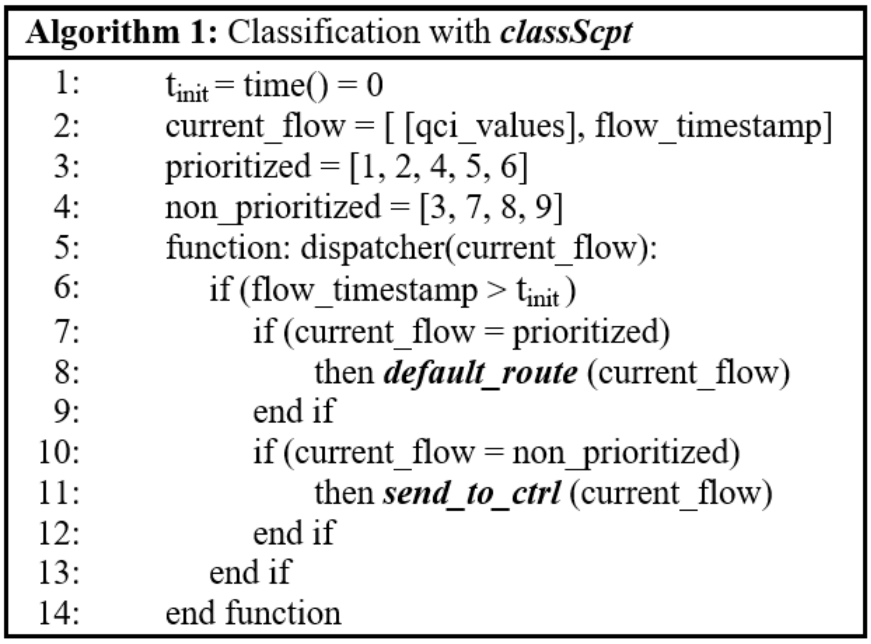
Fig. 3. *EFQM* controller architecture.

4.2 *EFQM* architecture

The proposed controller architecture in Fig. 3 shows the different components in detail with traffic flows processing from *Perception layer* to *Application layer*. The *classScpt* script gives the classification level allowing to ensure both, a good completion time for high latency packets and prevent controller overloading. The *classScpt* can send packets directly to the network layer (white arrows) or ask the controller for adequate *QoS* fulfillment (black arrows). It should be noted that when candidate paths are obtained from *SA*-based *QoS* routing algorithm, the best path is selected by an admission control component. Afterwards, the suited *Flow_mod* message (rule) is sent by controller to switches for processing packets of concerned flows.

Therefore, we avoid intentional flows deletion in order to limit unnecessary losses while respecting the flows deadlines. A detailed *classScpt* algorithm processing is proposed in the next section. This algorithm gives a basic flows classification in edge layer, according to Table 1 class model. For instance, **Algorithm** 1 depicted in Fig. 4 illustrates packets dispatching steps from the time they attempt *IoT* gateway in edge layer.

Note that *IoT* gateway is *SDN-enable* therefore it can communicate with the controller via southbound *API*. Once the traffic reaches this level, two choices are possible: either route the traffic directly to the *OpenFlow* switches, or, contact the controller for adequate routing rules (*Flow_mod* message).



Algorithm 1: Classification with ***classScpt***

```
$t_{init}$ = time() = 0
current_flow = [ [qci_values], flow_timestamp]
prioritized = [1, 2, 4, 5, 6]
non_prioritized = [3, 7, 8, 9]
function: dispatcher(current_flow):
    if (flow_timestamp > $t_{init}$)
        if (current_flow = prioritized)
            then default_route (current_flow)
        end if
        if (current_flow = non_prioritized)
            then send_to_ctrl (current_flow)
        end if
    end if
end function
```

Fig. 4. Traffic classification algorithm

4.3 Traffic classification

Upon receiving message from perception layer, whatever the traffic class, the *IoT* gateway, with *classScpt*, looks for the traffic corresponding class *QCI* values ("*prioritized*" or "*non-prioritized*"). The timestamp is used to ensure dissimilarity between flows. If the flow is *prioritized* (*QCI* value belong to 1, 2, 4, 5, 6) then the message is sent to next corresponding switch through the shortest path (*default_route* algorithm), else, the message is encapsulates within a *Packet_In* message and sent to the controller (*send_to_ctrl*) for appropriate path computation.

A controller by having a global view of network statistics information (topology and measurement), *SA*-based *QoS* routing algorithm and *EFQM* module, computes and selects the path that is most suited with respect to packet requirement. Afterwards, *EFQM* installs the response with the *Flow_mod* message on track switches along choosing path. Finally, effective traffic routing is performed without any intentional flow deletion.

A couple of functions that are used by *SA*-based routing algorithm and *EFQM QoS*-aware admission control are illustrated from **Equation 1** to **Equation 5**. Furthermore, Table 2 describes the meanings of different parameters that are used in Eq. 1, Eq. 2, Eq. 3, Eq. 4, and Eq. 5.

Table 2. Key nomenclatures.

C_p	Cost of path P
W_x	Weight for x QoS requirement
MR_p	Miss Rate for metric x
$p(C_p, C_x, t)$	Probability to accept new path x
ABW_p	Available bandwidth according to fixed routing path P
e_i	i^{th} link in the routing path P
c_i	e_i link capacity
b_i	Current bandwidth load on e_i
a_i	Available bandwidth on e_i

$$C_P = W_d \frac{(P_d - R_d)}{R_d} + W_j \frac{(P_j - R_j)}{R_j} + W_l \frac{(P_l - R_l)}{R_l}. \tag{1}$$

$$W_x = \frac{MR_x}{MR_d + RM_j + RM_l}. \tag{2}$$

$$MR_x = \frac{(flows\ that\ can\ not\ meet\ requirement\ x)}{(flows\ in\ pList)} \tag{3}$$

$$p(C_P, C_X, t) = \begin{cases} 1 & C_X < C_p \\ e^{\frac{-c|C_X - C_P|}{t}} & C_X \geq C_p \end{cases} \tag{4}$$

$$ABW_P = \min_{e_i \in P} a^i \ ; \ a_i = c_i - b_i \tag{5}$$

EFQM QoS-aware routing flowchart is illustrated in Fig. 5. It combines *SA*-based *QoS* routing algorithm and an admission control function. According to flow *QoS* requirement and source/destination IP addresses, *EFQM* uses Dijkstra's algorithm to compute the shortest path P_s then the cost of P_s named C_{ps} using **Equation 1**.

The used metrics are *delay* (d), *jitter* (j) and *loss rate* (l). Path P_s is stored in a list named *pList*. P_s consists of an initial solution of SA. An iteration value t is set and decreases whenever t is not null. The cost function computing needs the weights W_x of each metric x (**Equation 2**). If t is canceled and no path is accepted, a new neighbor N is determined and the process is repeated. Path acceptance probability is determined by **Equation 4**. If it exists a path X which cost improves the cost of P_s, then X replaces P_s. This process is the basis of this algorithm and it is repeated until t is equal to 0. So each time an improved path is found, it is appended to *pList*.

Finally, *pList* is built based on potential candidate paths. In fact, SA algorithm avoids being trapped in local optima but does not eliminate the possibility of oscillating indefinitely by returning to previous visited paths. The list *pList* is consequently defined to avoid this paths revisited. SA can be replaced by the

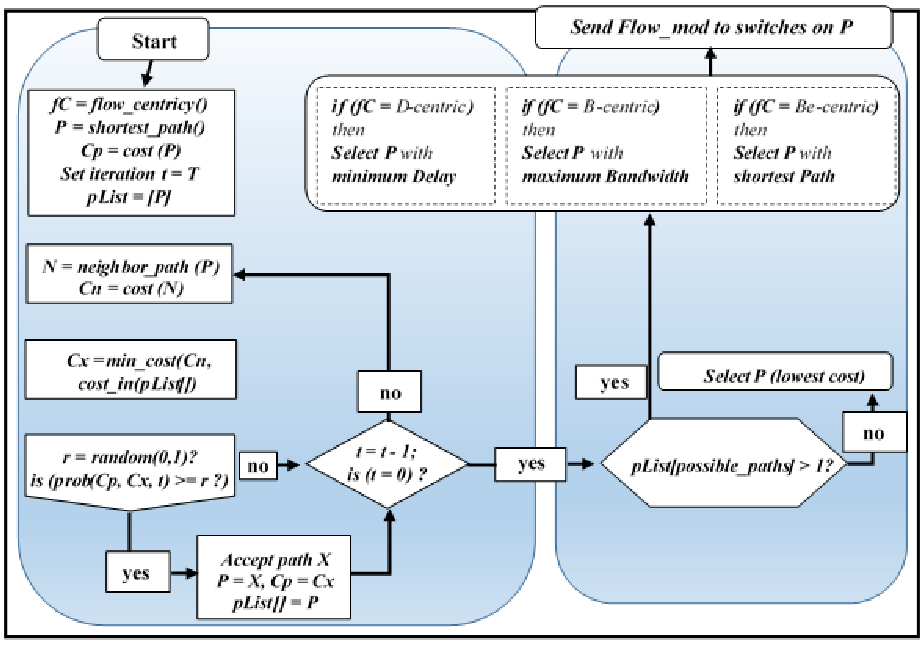

Start
fC = flow_centricy()
P = shortest_path()
Cp = cost (P)
Set iteration t = T
pList = [P]
N = neighbor_path (P)
Cn = cost (N)
Cx = min_cost(Cn, cost_in(pList[])
r = random(0,1)?
is (prob(Cp, Cx, t) >= r ?)
no
yes
Accept path X
P = X, Cp = Cx
pList[] = P
t = t - 1;
is (t = 0) ?
no
yes
pList[possible_paths] > 1?
yes
no
Select P (lowest cost)
if (fC = D-centric) then Select P with minimum Delay
if (fC = B-centric) then Select P with maximum Bandwidth
if (fC = Be-centric) then Select P with shortest Path
Send Flow_mod to switches on P

Fig. 5. Flowchart of proposed EFQM

tabu search algorithm if the state space was larger. *tabu search* can also minimize the size of *pList* with an automatic memory-based reaction mechanism.

In fact, a suitable path is chosen among candidate paths within *pList*. This choice is crucial since all paths are improving paths. Therefore, the best one that meets the needs of the current centric traffic will be the selected path.

Therefore, according to *EFQM*, each packet is optimally forwarded in order either to minimize end-to-end delay, or increase bandwidth, or reduce contention to satisfy resource limits of *IoT* server. This is suitable specifically for network with limited resources.

The implementation setup and *EFQM* performance evaluation is presented in the Section 5. *EFQM* is compared to *AQRA* [5] according to overall flow end-to-end performance and system runtime evaluation. MR_x consists of miss rates for metric x requirement, as shown in (**Equation 3**). For any given link e in path P with capacity c and available bandwidth a, the "*Available Bandwidth*" (*ABW*) of a routing path P is computed by (**Equation 5**).

5 *EFQM* evaluation

Our experimental testbed is based on "*Ryu*" *SDN* controller [20] and "*Mininet-Wifi*" [21]. *Ryu* is an *OpenFlow* based controller which provides python language based application development. According to topology discovery, we use a

Ryu module/library called *topology*. A *python* graph library *networkX* is used for network view. The proposed system is simulated within *Ubuntu* 18.04.1 *LTS*.

The deployed testbed network consists of three *OpenFlow* core switches, two *OpenFlow* edge switches, 15 *OpenFlow-enabled* access points connected to 20 end devices accessing the network via WiFi (*IEEE* 802.11*n*). Three stations included in network act as application servers with different services requirements. We used *iPerf* an active measurement tool [22] in order to generate test traffic and measure the performance of the network. *iPerf* enables to get, for each test, the reports of loss, bandwidth and other parameters.

The performance evaluation is done in two steps: the overall end-to-end performance in terms of delay and loss rate and total system runtime. Firstly, we evaluate the transmission time for *prioritized* and *non-prioritized* traffic. For *prioritized* traffic, the measurement is the total transmission delay for packets sent with *default_route* function of *classScpt*. In contrast, for *non-prioritized* flow delay consists of time required to route a flow using the controller specifications. Secondly, we assessed the runtime estimation in *EFQM* process from source to servers.

EFQM is compared with *AQRA* [5] in relation to overall end-to-end flow performance and system runtime. Table 3 shows the experimental result. The end-to-end flow performance of *EFQM* by considering the default route is 7.92% better than *AQRA* in terms of *delay*. Nevertheless, *AQRA* gives an enhanced packet loss rate (reduced by 8%) than *EFQM*.

The end-to-end flow performance of *EFQM* with history reduces that of *AQRA* with history by 21.23% and 23.52% in terms of *delay* and *packet loss* rate, respectively. This is due to the fact that, in *AQRA* [5], sending packets with very high priority is affected not only by the waiting time for routing decision coming from the controller but also by the network degradation comes from frequent low priority packets deletion.

Regarding to default route, even if it does not guarantee all *QoS* requirements, it fulfills *delay* and better escapes bottlenecks given the limited size of low-priority data. In addition, elephant flows that are more suitable to overload the network are optimally managed in terms of *QoS* by *EFQM*, therefore less interference by sending priority flows are noticed. This situation explains losses reduction as illustrated in Table 3 by *EFQM* with history.

Table 3. End-to-end flow performance.

Overall end-to-end performance	**Delay (ms)**	**Packet loss rate (%)**
AQRA (with history)	89.10	0.051
EFQM (default route)	82.04	0.056
EFQM (with history)	70.18	0.039

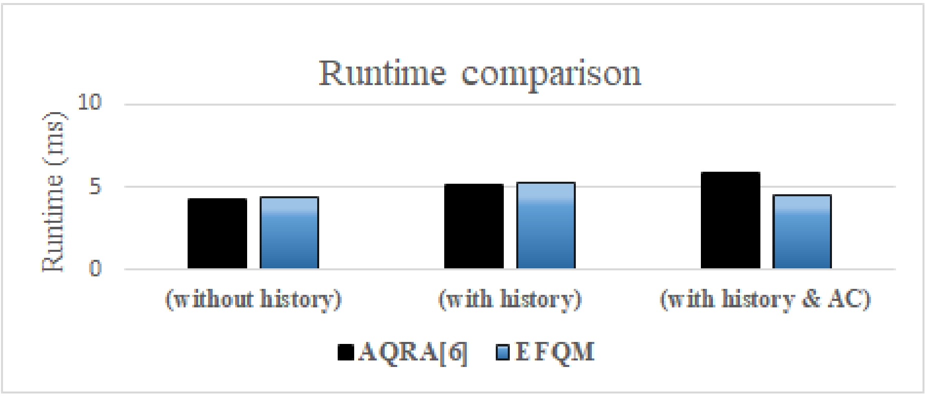


Fig. 6. Runtime comparison

The total runtime of $EFQM$ is computed by subtracting from the end-to-end delay, the time between the gateway and the controller (T_{gc}) plus time between controller and the servers (T_{cs}) as described in equation (**Equation (6)**).

$$T_{runtime} = T_{end\_to\_end} - (T_{gc} + T_{cs}) \tag{6}$$

Fig. 6 illustrates the runtime comparison of proposed $EFQM$ and $AQRA$ [5] according to use or not of $pList$ (history), of admission control algorithm (AC) or none of them. Compared to $EFQM$, $AQRA$ [5] proposes best runtime in two scenarios: reduce runtime by 0.96% with history and 1.38% without history. This is due to $classScpt$ processing time which exists in any of these scenarios. However, $EFQM$ decrease $AQRA$ [5] runtime by 23.29% when we consider history and AC. Indeed, time used by the $AQRA$ controller model to drop packets at edge level increases it processing time due to the waiting of next flow receive for applying the new control rules and path change processing.

In addition, The AC processing latency increases this runtime delay due to packets deletion in edge layer. This situation occurs when there are multiple successive low-priority flows or multiple flows with the same priority coming at the same time to edge equipment. It should be noted that $EFQM$ does not control incoming flow as long as it arrives at the controller. Note that by sending directly packets, $EFQM$ avoids overloading the controller as well as considerably reduces local buffer (gateway) utilization rate.

To the best of our knowledge, $EFQM$ gives a good QoS-aware approach that outperforms previous studies. Indeed, it takes into account 3 metrics to cover a wide performance aspect of IoT traffic, in contrast to former works such as [7], [9], [11] that consider just 1 metric like delay; or 2 metrics like [10].

6 Conclusion

This paper illustrates a new flow QoS management mechanism for SDN-based IoT network. $EFQM$ proposes a framework which aims to reduce flow processing delay and congestion caused by frequent packets deletion. Therefore, it limits flow deletion process by fixing two sorting levels for better performance.

Firstly, $EFQM$ separates vulnerable latency, loss sensitive and very high priority flows to others. These flows are sent directly to avoid delay constraints. The remaining traffic flows are sent to a fixed controller. A second level of sorting based on flows specific requirements is applied after the computation of the overall enhanced paths. Our evaluation results have shown that, $EFQM$ (default route) outperforms $AQRA$ with history in terms of end-to-end delay performance.

Furthermore, the end-to-end flow performance of $EFQM$ with history reduces $AQRA$ with history by 21.23% and 23.52% according to delay and packet loss rate, respectively. Finally, by considering history approach and AC, $EFQM$ runtime decreases by 23.29% compared to $AQRA$ runtime. However, $AQRA$ gives best packet loss rate (reduced by 8%) than $EFQM$ (default route) and decreases $EFQM$ runtime in two scenarios: 0.96% with history and 1.38% without history.

We plan to compare $EFQM$ and $AQRA$ under different conditions and scenarios, for instance, when the topology is highly dynamic with more or less switches in the data plane.

References

1. C. Pham, A. Rahim and P. Cousin.: Low-cost, Long-range open IoT for smarter rural African villages. in Proc. IEEE ISC2, pp. 1-6, Trento, 2016.
2. M. R. Seye, M. Diallo, B. Gueye and C. Cambier.: COWShED: Communication Within White Spots for Breeders. in Proc. IEEE ICIN, pp. 236-238, France, 2019.
3. E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou.: Software-defined networking (SDN): Layers and architecture terminology. IRTF, ISSN: 2070-1721, RFC 7426, pp. 1-35, Jan. 2015.
4. N. McKeown et al.: OpenFlow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, (Mar. 2008).
5. G. Deng and K. Wang.: An Application-aware QoS Routing Algorithm for SDN-based IoT Networking. in Proc. 2018 IEEE ISCC, pp. 186-191, Natal,2018.
6. B. Oh, S. Vural, N. Wang and R. Tafazolli.: Priority-Based Flow Control for Dynamic and Reliable Flow Management in SDN. in IEEE Transactions on Network and Service Management, vol. 15, no. 4, pp. 1720-1732, Dec. 2018.
7. Sulthana SF, Nakkeeran R.: Performance Analysis of Service Based Scheduler in LTE OFDMA System. Wireless Personal Communications. vol. 83(2), pp. 841–854, 2015.
8. K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, M. Thottan, J. Rexford, A. Vahdat.: Measuring control plane latency in sdn-enabled switches. in Proc. ACM SIGCOMM SOSR, pp. 1-25, USA, 2015.

9. X. Guo, H. Lin, Z. Li and M. Peng.: Deep Reinforcement Learning based QoS-aware Secure Routing for SDN-IoT. In IEEE Internet of Things Journal, Vol. 7, pp. 6242-6251, 2019.
10. A. Montazerolghaem and M. H. Yaghmaee.: Load-Balanced and QoS-Aware Software-Defined Internet of Things. In IEEE Internet of Things Journal, vol. 7, no. 4, pp. 3323-3337, 2020.
11. M. Jutila.: An adaptive edge router enabling internet of things. IEEE Internet of Things Journal, vol. 3, no. 6, pp. 1061-1069, 2016.
12. S. Jeong, D. Lee, J. Hyun, J. Li and J. W. Hong.: Application-aware traffic engineering in software-defined network. 19th APNOMS. pp. 315-318, Seoul, 2017.
13. I. Gravalos, P. Makris, K. Christodoulopoulos and E. A. Varvarigos.: Efficient Network Planning for Internet of Things With QoS Constraints. In IEEE Internet of Things Journal, vol. 5, no. 5, pp. 3823-3836, 2018.
14. 3GPP, "Quality of service (QoS) concept and architecture," TS 23.107, Last accessed: 2020-05-29.
15. N. Mesbahi and H. Dahmouni.: Delay and jitter analysis in LTE networks. in Proc. WINCOM, pp. 122-126, Fev, 2016.
16. Z. Qin, G. Denker, C. Giannelli, P. Bellavista and N. Venkatasubramanian.: A software defined networking architecture for the internet-of-things. In Proc. IEEE NOMS, pp. 1-9, Krakow, 2014.
17. Haikal Amira, Badawy Mahmoud, Ali Hesham. : Towards Internet QoS Provisioning Based on Generic Distributed QoS Adaptive Routing Engine. The Scientific World Journal, vol. 2014, pp. 1-29, 2014.
18. Mamman Maharazu, Zurina Mohd Hanapi, Abdullah Azizol, Muhammed, Abdullah.: Quality of Service Class Identifier (QCI) radio resource allocation algorithm for LTE downlink. PLOS ONE Journal. vol.14(1), pp. 1-22, 2019.
19. sFlow.org, *www.sflow.org.*
20. Ryu, Component-based software defined networking framework, *https://github.com/faucetsdn/ryu.*
21. Mininet-wifi, Emulator for Software-Defined Wireless Networks, *https://github.com/intrig-unicamp/mininet-wifi.*
22. iPerf, The ultimate speed test tool for TCP, UDP and SCTP, *www.iperf.fr.*