Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Self-adaptive QoS-management Framework for Highly Dynamic IoT Networks

AVEWE. BASSENE[1] and BAMBA GUEYE[2].
[1]Université Cheikh Anta Diop de Dakar, Senegal (e-mail: avewe.bassene@ucad.edu.sn)
[2]Université Cheikh Anta Diop de Dakar, Senegal (e-mail: bamba.gueye@ucad.edu.sn)

Corresponding author: A. Bassene (e-mail: avewe.bassene@ucad.edu.sn).

ABSTRACT IoT infrastructure makes great demands on network control methods for dynamic and efficient management of massive amounts of nodes. Software-Defined Networking (SDN) enables to handle dynamically network traffic as well as flexible traffic control in real-time. However, while providing flexibility and scalability, SDN-based architecture still remains ineffective to self-adapt with respect to network topologies with more or less switches in the data plane (highly dynamic topology). Having a centralized control plane is not an acceptable situation because that would represent a single point of failure in the network. Using multiple controllers that ensure flexibility and high availability would be a solution; meaning that if one controller has problems and fails, the other would be ready to take over and control the network. Thus, having a single controller raises the problem of scalability while multiple controllers call for a distributed states management problem. To overcome such issues, we propose *EFQM++*, a self-adaptive framework for highly dynamic network topology changes. By leveraging SDN controller topology discovery mechanism, *EFQM++* improves flow end-to-end transmission delay. It tackles flexibility and scalability related to a single point of failure problem and gives distributed states management solutions in large scale IoT networks. *EFQM++* reduces up to 6% and 13% the average delay in contrast to previous works like $EFQM$ and $AQRA$, respectively.

INDEX TERMS Software-Defined Networking, Internet of Things, Performance, Quality of Service, Clustering.

I. INTRODUCTION

THE Internet of Things (IoT) is rapidly gaining ground in the scenario of modern wireless telecommunications. The basic idea of this concept is the pervasive presence around us of a variety of things or objects such as Radio-Frequency IDentification ($RFID$) tags, sensors, actuators, mobile phones, etc. These devices can be attached and/or revoked from the network every second which makes the IoT network highly dynamic, evolutionary, and frequently changing over time.

Software-defined networking (SDN) is a constantly progressive technology that offers more flexible programmability support [1]. The Open Networking Foundation defines SDN as an adaptable and manageable emerging model that deals with the dynamic aspect of today's network applications. However, there are some scalability issues in large IoT network and SDN-based IoT architecture still lacks to adapt to such a wide and dynamic network (network that include several devices) topology changes [2]. Topology discovery mechanism (TDM) is a critical component of any SDN architecture and often the most used criterion to improve network performance in IoT environments. An efficient TDM with the SDN concept of network and programmability could be key elements for the implementation of a suitable adaptive quality of service (QoS) support. The deployment of distributed SDN controllers raises challenges regarding to TDM and distributed states management problems in large scale IoT network [3]–[7].

We think that, the reduction of the TDM delay and the management of control plan devices depending on on-demand application requirements can be a smart strategy to deal with such a changing network topology. We propose *EFQM++*, a flexible framework that autonomously makes adaptive QoS management decisions reliant on monitoring network topology status, clustered controllers load and the IoT application traffic requirements. *EFQM++* consists of a group of two entities; it is able to collect both network topology and controller status information in one entity and then join it to the second entity which interacts with server application to fulfill current traffic QoS requirements. Finally, the framework decides the best network topology change to apply accordingly.

The first entity must imperatively be linked with an efficient TDM for good experimental results. To ensure a suitable control entity for our framework, the performances of the two SDN controllers, RYU [8] and $ONOS$ [9], were measured using the *Mininet* SDN simulation environment. The second entity is based on QoS-aware algorithm used in $EFQM$ [10] and interacts with application Layer. This algorithm is an heuristic algorithm based on "$Simulated\ Annealing$" (SA) used to find the approximate optimal path according to multiple constraints (delay, bandwidth, best-effort).

The rest of the paper is organized as follows. Section II reviews related works. Section III presents the TDM in SDN in general and introduces our self-adaptive QoS management framework architecture. Section IV tests controllers performances in different dynamic testbeds scenarios. Section V evaluates *EFQM++* performance. Finally, a conclusion and some open research perspectives are given in section VI.

II. RELATED WORK

Several works have been investigating the TDM problem [5]–[7] to face SDN-based architecture shortage.

Authors in [2] propose an architecture that autonomously make QoS related decisions based on network topology, distributed controllers status and application requirements from the business layer. However, any implementation is proposed to test proposed architecture.

Peros et al. [11] propose an SDN framework to support dynamic QoS for the IoT network. However their QoS aware approach is based on a given default QoS profile values instead of current IoT network traffic state. Tangari et al. [12] give approach for heterogeneous applications resources monitoring with frequent network state updates. Their resources monitoring way is seated on a network with static topology.

Deng et al. [13] propose $AQRA$ for SDN-based IoT network to fulfill a multiQoS 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. In $EFQM$ [10], authors propose an enhanced QoS management approach, that reduces, among others, spent time within the control plane (CP). However, both $AQRA$ and $EFQM$ are limited by the fact that they treat the problem with a static consideration of the QoS management aspect - all decisions relating to this were invariant from the network topology view and all along the experimental time. On top of that, they consist of a centralized CP.

In contrast to previous works, *EFQM++* promotes a use of several QoS metrics and gives an approach that deals with QoS decisions over time in a dynamic network topology. It manages QoS decisions related to both the real-time traffic demand as well as topology changes. As opposed to works in [2], *EFQM++* proposes an implementation to test the proposed self-adaptive framework architecture. To answer challenges of reliability, scalability and fault tolerance, our framework runs under a clustered distributed controller in the emulated setting of $SDNs$.

III. SELF-ADAPTIVE QOS-MANAGEMENT FRAMEWORK

This section provides a provides a comprehensive review on TDM in SDN controllers in general. The proposed framework architecture with it different components and abilities are explained. The TDM is considered to be the key element for the $SDN\ CP$ scalability problem since it tests its abilities to cope with constant topology changes. In an SDN-based network, the CP is responsible for making decisions on how packets should be forwarded by one or more network devices and pushing such decisions down to the network devices for execution [1]. To be able to give a good forwarding decision, the knowledge of the network topology is necessary. The deployment of the TDM in SDN is well described in [2].

A. PROPOSED FRAMEWORK ARCHITECTURE

The architecture of the *EFQM++* consists of two components interacting between different SDN architecture levels.

Fig. 1 depicts the framework architecture components along with the communication channels that connect them. The key idea of this work is to alleviate scalability concerns by extending the responsibilities of the CP in level above (management) in order to relieve the load on the controller. The control layer provides information about network topology status and devices statistics.

The Management layer hosts the key elements of our framework: a database and a QoS-management Application. This layer monitors and stores information about network topology and manages QoS based on QoS-aware management algorithm in [10]. *EFQM++* is then able to : (1) detects new devices that join the network, (2) gets the real-time topology updates information, (3) gives availability of control devices, (4) provides current application traffic requirements, (5) calculates QoS-aware paths between pairs of devices,(6) route traffic according to the topology state, the controllers load and the current application demands. The rest of this section presents *EFQM++* communication mechanism approach. This communication mechanism gives factors to deal with scalability for distributed states management problem.

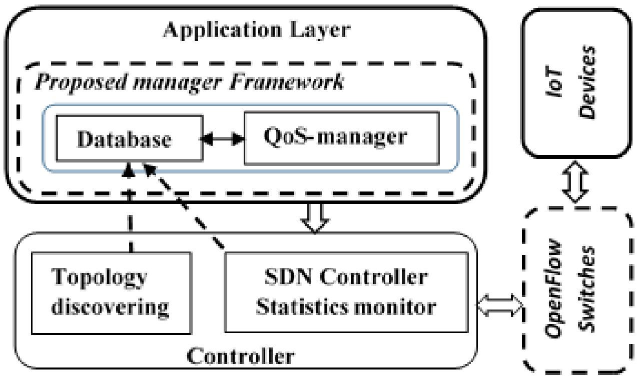

FIGURE 1. Self-adaptive QoS-aware framework architecture for SDN-based IoT network

B. EFQM++ PROCESSING APPROACH

This section describes the *EFQM++* framework components and evaluates its end-to-end transmission delay compared to $EFQM$ [10] and $AQRA$ [13].

The framework presents a self-adaptive QoS management solution that deals with highly dynamic SDN-based IoT networks. The proposed design is first focused on the monitoring of the $SDN\ CP$ status in a distributed environment. For this purpose, the solution consists of three elements associated with a database: a Collector, a Monitoring Agent and a QoS-aware algorithm given in [10].

The QoS-aware algorithm is a SA-based algorithm used to find the approximate optimal path solutions based on topology information and on-demand application data (priority).

On-demand application data is managed as follows: according to the flow priority (QCI) value and collected controllers statistics information (load), a binary decision is made regarding to switch between the least loaded controller and a default master of concerned switch. If current flow is *delay-centric*, then it is forwarded through the least loaded controller. Otherwise, the default master controller is used. These tasks are delegated from controllers to the manager framework components via the northbound API. Once paths and controllers state information are available, the framework processes the packets promptly with necessary resources (topology information) and decisions to gain the final destination. Thus, the framework with a global view of the network, computes and selects the path that is most suited with respect to packets requirement. The manager framework ensures the success of the distributed states management to tackle problems related to scalability.

Fig. 2 shows this communication mechanism. A detailed processing is described in Algorithm 1. In our test we limit to a cluster of three instances for the control plane.

Our framework, with information on the number of controllers in the cluster (i), each controller statistics (C_Load) and traffic priority (Flow priority), processes as shown in Algorithm 1.

Upon a flow arrives from a switch in CP to its master controller, if the flow is *delay-centric*, the master controller of the corresponding node is ordered (with a *Packet_Out* message) to send the path with minimum-delay corresponding to the *packet_In* message header. The controller then updates entries on the switches along dedicated path. By contrast, if the flow is not *delay-centric* then a *Packet_In* message is forwarded to the manager framework in the application layer. The manager framework picks the least loaded controller from the cluster and send it information about the adequate QoS-aware path (*maximum-bandwidth* or *best-effort*) from dedicated paths list. The controller then pushes the flow rules to the switches in data plane (*Packet_Out* message). Thus, traffic is forwarded to our manager framework only if it is not *delay-centric* and if there are more than one controller instances as described in Algorithm 1.

Algorithm 1: Communication processing in *EFQM++*

```
REQUIRE: $i$, $C\_Load$; $Flow\_priority$
When a packet comes to a switch in data plane
A Packet_In message is sent to the master controller
if $i$ is greater than or equal to 1 then
    if the flow is delay-centric then
        Packet_Out: Returns the path with
        minimum-delay and updates the flow entries
        on switches along corresponding path
    else
        A Packet_In message is forwarded to the
          manager framework. The framework picks
          the least-loaded controller in cluster.
        Send path with maximum-Bandwidth
          information to the chosen controller.
        Packet_Out: Returns the path with
        maximum-Bandwidth then pushes the flow
        rules to the switches in data plane.
    end if
end if
```

We adopt this approach because it has been shown that a single control plan component is good enough to cover transmission delay requirements in most topologies [14]. Although, in clustering mode, the default master can an-

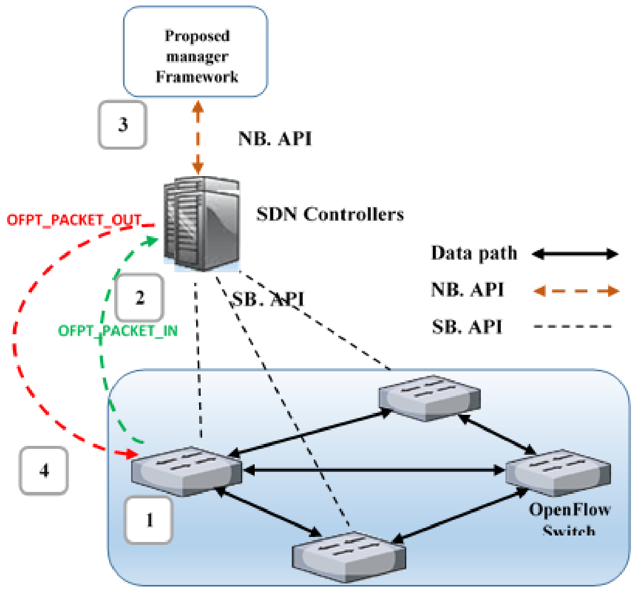

FIGURE 2. Communication mechanism within *EFQM++* Framework

swer delay concerns, while the least loaded is chosen to avoid/reduce congestion that drastically impacts throughput. Thus, *EFQM++* gives availability and robustness since a distributed cluster of controllers can coordinate to provide resilience and fault-tolerance, which is required if any of the controllers fails [15].

The collector component is developed as a core application of $ONOS$ and it designed for a clustered environment. It collects the real-time statistics of individual controller system metrics that include CPU, memory, disk I/O, network I/O. It then provides an aggregation service to compute the overall load of the entire CP. It is based on the *read* plug-ins of *collectd* system and the "*write_onos*" Python plug-in module to send the real-time statistics to listener service of the collector. This proposed collection solution gives real-time load statistics of individual controllers and the entire $ONOS$ cluster [16]. These results can be helpful to identify controllers load.

The Monitoring Agent pushes the network topology status to the centralized PostgreSQL database.

The performance evaluation is done in two steps: firstly, the overall end-to-end performance in terms of delay when the underlying OpenFlow switches are connected to one controller. Secondly, when the underlying OpenFlow switches are connected to multiple controllers in a clustered mode. Indeed, $ONOS$ supports running multiple controllers in a clustering mode where they share state among each other. Furthermore, when the underlying OpenFlow switches are connected to more than 1 controller, they determine which controller should be the master and which should be the standby/slave. This is very useful for fault tolerance and high availability purposes [9]. The RYU controller too has the advantage of being able to work in a distributed manner [17]. The next section illustrates performances comparison between these two previous SDN controllers, which represent the “brains” of the network.

IV. EXPERIMENTAL SETUP

The TDM is crucial to evaluate controller performance. This section aims to compare $ONOS$ and RYU controllers TDM efficiency to deal with highly dynamic network. Our testbed is deployed using different virtual machines (VM) hosted on *VirtualBox Version 6.0.4 r128413*. The PC is x64-based with *Intel(R) Celeron(R) CPU N3060* 2.4 GHZ and 8 GBytes of memory. Among these three VM, we deploy $ONOS$ controller (".ova" VM), RYU controller and its requirement packages and a virtual network infrastructure for simulation *Mininet*.

The *Mininet-Wifi* [18] emulator also enable a dynamic network topology and mobility. In addition, it supports several built-in basic network topologies like Minimal topology, Single topology, Linear topology, Tree topology, etc. [19]. The first testbed environment executes the $Mininet$ command to extend the network and its underlying topology at running time.

The ultimate speed test tool *Iperf* [20] is used for network performance measurement. In SDN, *Iperf* can create data traffic and report parameters as bandwidth, throughput, jitter, and packet loss between two nodes in both directions.

Our first testbed evaluates the statistics related to the controllers performance. An scalable IoT topology is generated in a dynamic manner to mimic real highly dynamic network topology. We conceive a custom topology with 6 different steps with growing numbers of switches and hosts as shown in Table 1.

Our evolutionary proposed testbed topology group together a well-ordered set of real topologies ranging from

TABLE 1. Scenario Table for Experiment.

Steps	Number of Switches	Number of Hosts
0	1	2
1	2	3
2	3	4
3	6	7
4	12	13
5	Link for mesh topology	13
6	Add a host to generated Switches	24

a linear topology to a more complex mesh topology with respectively 12 switches and 24 hosts. This testbed steps evolves for each SDN controller aim to evaluate at least two major QoS performance parameters - the efficiency of the TDM of each SDN controller - the scalability, to mention the controller capability to handle the growth of the network.

In this first testbeb, a mesh topology was created in running time using mininet command. This means four layer of switches aligned in a tree topology with links connecting switches to get the mesh topology. In each step, we evaluate performances of controllers under both topologies while considering metrics like latency, throughput and jitter.

Our second test environment is deployed in a distributed way to support multiple controller instances. It is about comparing transmission delay between proposed framework and literature in various control plan status.

The experimental setup uses a cluster of 3 SDN controllers in the CP. Firstly, while all controllers are added to all switches/devices, we bring down two controllers to let the remaining one become the master. This standalone mode is used for a first comparison step. Secondly, we enable the cluster by starting the stopped controllers and perform the second evaluation. The switches are connected to the master controller with the IP addresses indicating the standby controllers for each connected switch. The clustered controllers perform load balancing by distributing the number of connected OpenFlow switches between instances of the controllers in the cluster.

The test mode is using an incremental number of switches, the deployed testbed network consists of 3 OpenFlow core switches, 2 OpenFlow edge switches, 15 OpenFlow-enabled access points connected to 20 end devices. Since we are interested in the dynamic aspect of the data plane, the switches are increased by 15 per test until a maximum limit of 90 switches. The total number of iteration is counted as 6 in both standalone and clustering mode.

V. PERFORMANCE EVALUATION

The most important component of any SDN controller architecture is the TDM. TDM is a service that typically runs continuously in the background of all SDN controllers and remains a master of all decision-making. It is therefore important to know requirement performance and load it imposes on the chosen controller. We evaluate these controllers performance in terms of some QoS parameters such as delay, throughput, and jitter while deploying a predefined dynamic topology.

A. CONTROLLERS PERFORMANCE IN DYNAMIC IOT TOPOLOGY

This section first defines testbed performance parameters before evaluating metrics under previous proposed steps for the two SDN controllers - $ONOS$ and RYU. We have conducted each test with a 10 number of iterations to have optimal average measurement results. The test will run for 100 seconds, in each of these 10 iterations. For each test, the network performance parameters (Average Delay, Throughput, and Jitter) are measured to evaluate the performance of the controllers.

Delay Measurement: the Delay is the time it takes for traffic to leave the sender and arrive at the destination. It can measure based on the Round-Trip Time (RTT). The average RTT is in milliseconds (ms) and can be measured using *ping* tool between two hosts, with a total of 10 ICMP packets.

The average latency values all along with the different steps described in Table 1 is observed in Fig. 3. The X-axis illustrates the average latency of packets, while the Y-axis depicts various network topology status. We can notice the average latency variation from the initial topology before any change to mesh topology with hosts adding step (final step).

The results in Fig. 3 show that the average latency of $ONOS$ is initially about 0.099ms compared to 0.211ms for that of RYU. From the initial static single topology to the next step, the latency increases faster for RYU than for $ONOS$. For steps 1 ($stp1$) and 2 ($stp2$) of Table 1, we observe the results when adding a switch with one host. We note that the average latency increases by a relatively small value for $ONOS$ (0.013ms). In contrast to $ONOS$, the latency generated by switch adding operation is considerably too high for RYU, about 3 times that of $ONOS$. The reason is that RYU takes more time for updating new switch while $ONOS$ built-in mechanisms support updating components in running time. This allows a flexible adding functionality approach to the controller.

The average latency value for $ONOS$ stay constant when the number of switches is greater than or equal to 12 (from 0.139ms to 0.175ms). In contrast to $ONOS$, RYU latency is increasing with the topology state. This means that, RYU controller is more sensitive to dynamic topology changes than $ONOS$, even when it just a simple host adding step ($stp6$). From $stp6$, any latency variation is notice for $ONOS$, meaning that $ONOS$ is less or insensitive in terms of latency during host adding operation. Therefore, any host discovery mechanism is needed for improved latency when using $ONOS$ controller. Even less if $ONOS$ can be used to leverage this host discovery mechanism without overhead.

Throughput Measurement: it defines the quantity of useful data that can be transmitted per unit time. The throughput in $Mbps$ can be measured using *Iperf* command between *Iperf* client and server.

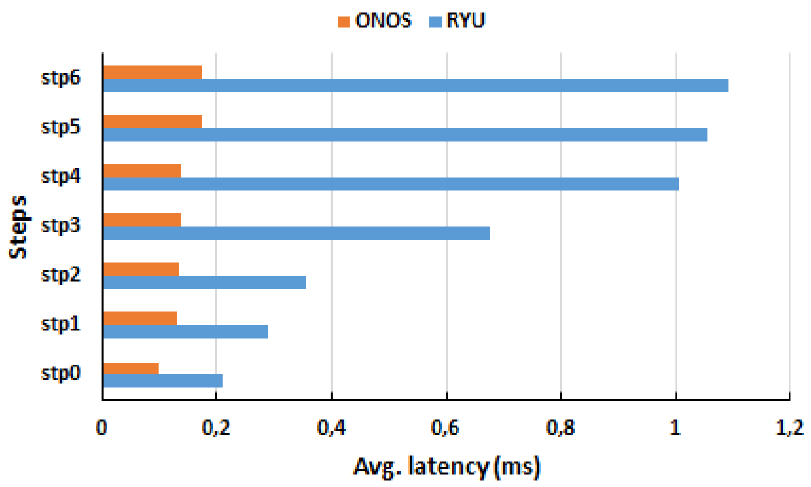

FIGURE 3. Average latency for highly dynamic SDN-enable IoT network

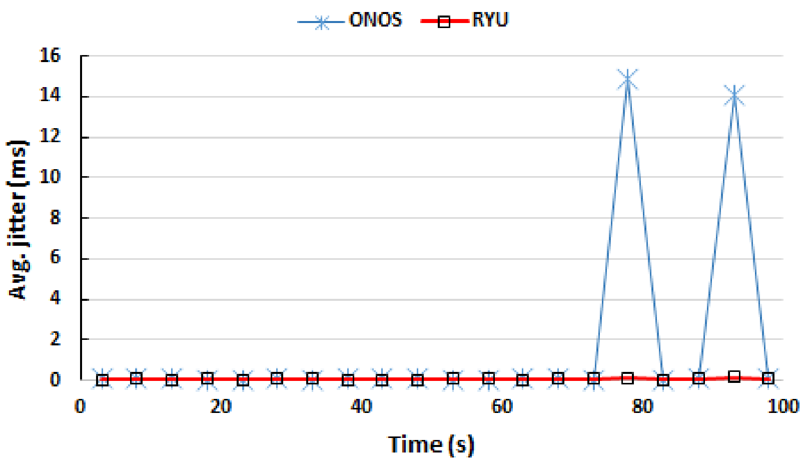

FIGURE 4. Average jitter for highly dynamic SDN-enable IoT network

The throughput evaluation results are ploted to observe it variation over time. This figure has not been inserted. The throughput values for $ONOS$ and RYU controllers kept stable during testbed deployment scenarios. The average throughput for $ONOS$ is around 11 $Gbit/s$ while RYU gives around 8.4 $Gbit/s$. In contrast to RYU, $ONOS$ controller gives the best throughput performance. This is because of built-in mechanisms of $ONOS$ deals with connecting/disconnecting components while the controller is running. Moreover, it has also an inherent support for very large scale networks as mentioned in [21]. One possible explanation for a lowest values of throughput for RYU would be the congestion in the network switches due to switches adding operation, which results in network performance degradation. In fact, RYU is a resource demanding controller which uses CPU and RAM utilization to optimum and thus results in degraded performance in presence of increasing number of nodes.

Jitter Measurement: it refers to the variation in latency of packets.

Fig. 4 illustrates the results of jitter parameter. The jitter is the latency variation and does not depend on the latency (response time or RTT). The jitter result for all steps is relatively too low (under 0.1 ms) for both $ONOS$ and RYU controllers, except at times $78s$ and $93s$. Thus, we observe that jitter do not vary considerably, which refers to a reliable connection. Indeed, the jitter value is particularly essential on network links steadiness because a high jitter can break the connection. The average jitter variation recorded at times $78s$ $(14.9$ $ms)$ and $93s$ $(14.1$ $ms)$ is related to links disconnection and re-connection when adding switches in steps $stp3$ and $stp4$.

The testbed results show that the performance of $ONOS$ in terms of throughput is better than that of RYU. In terms of latency, the latency generated by $ONOS$ is smaller than that of RYU while recorded jitter values, this shows that all of these controllers can give reliable connection support in a highly dynamic network environment. The general experimental results show that $ONOS$ outperforms RYU in all along with testbed steps, granting flexible, dynamic, and scalable network topology re-configurations. From this analysis result, $ONOS$ controller is adopted for proposed framework control plan device since it exhibits the best experimental results showing that it is able to respond to requests more promptly under various traffic loads.

B. $EFQM$++ EVALUATION

$EFQM$ as well as $AQRA$ are based on a static state of network topology while considering QoS management decisions most related to generated traffic. In addition, these proposals also consist of a centralized entity that can increase the risk to be a single point of failure [2].

In this section, we compare the performances of $EFQM+$ $+$ with those of $EFQM$ and $AQRA$ in both single as well as multiple controllers environment to mention the scalability problem in distributed systems. The main purpose of this evaluation is to show how QoS management decisions based on both generated traffic and the dynamic aspects (load and control plane statistics) of the underlying network topology could help improve the end-to-end transmission delay. A well synchronous distributed control plane such as the one used here gives reliability, robustness and better transfer delay.

Our second testbed setup is previously described in section IV. Our framework application uses ONOS controller RESTAPI to get the global network topology details and controllers load. The framework consists of a Python console application with a database. the information from the controller and data plane nodes is stored using PostgreSQL docker container.

The experimental results for this second testbed in both standalone as well as clustering modes can be observed in Fig. 5. It compares flow end-to-end transmission delay among *EFQM++*, $EFQM$ [10] and AQRA [13]. From Fig. 5, we notice that the values of the end-to-end delay when working in standalone mode as well as clustering mode is small owing to its optimum features values (low latency, broadband and reliable connection) which contribute to the improvement in the QoS.

Nevertheless, when working in standalone mode, we can observe that *EFQM++* improves end-to-end transmission delay of $EFQM$ and $AQRA$ by an average values of 6.68% by $13,36\%$, respectively. The reason for this is that, the low values of throughput in RYU-based environments (like $AQRA$ and RYU) when failover (disconnection and reconnection) occurs at data plan level. In fact, all OpenFlow switches have multiple paths for each source/destination pair of nodes. When a link fails, the controller, based on the pre-established paths, monitors the status of each port of each OpenFlow switch. The controller decreases transmission rate of the port that exceeds the rate threshold to switch the flow with minimum rate to another path. Decreasing transmission delay has drastically impacted delay for low throughput environments ($EFQM$ and $AQRA$ delay in Fig. 5).

The end-to-end delay with a single controller increases compared to clustering mode when the number of OpenFlow switches is larger than 35. Indeed, a single controller creates a performance bottleneck when the number of *Packet-In* requests towards it increases i.e. when number of switches is up to 35 (as illustrated in Fig. 5).

Clustering mode reduces the end-to-end delay by an average values of 6.72% and 12.25% compared to $EFQM$ and $AQRA$, respectively. These results verify the fact that, the support of the clustering feature results in reducing the end-to-end delay as mentioned in [22]. Clustering mechanism also gives better scalability and performance by enabling a load balancing function to distribute the load evenly across the nodes between the source and destination. Load balancing reduces congestion and leads to average good delay in transmission time (around 76.368 ms). In addition, with clustering mode, the processing time for *packetIn* messages is reduced [23], this could explain latency reduction when number of switches is greater than 35 (see Fig. 5). However, the latency increases with the number of switches when this number is less than 35, this may be having extra synchronization overhead in distributed controller clustering. Compared to *EFQM++* framework and $EFQM$, $AQRA$ suffers from delays in both clustering and standalone mode. This delay is due to the network degradation caused by SDN switches transmission disruption, which leads to packets lost and a delay extension as shown in [10]. This delay is more noticeable if several paths changes occur, especially in a low bit rate transmission network.

VI. CONCLUSION

We proposed *EFQM++*, a self-adaptive QoS-management framework of SDN controllers for highly dynamic IoT networks with more or less switches in the data plane. Distributed controllers are configured to adapt the network according to on-demand applications and available control plan resources. The experimental results show that *EFQM++* reduce the average transmission delay by 6.72% and 13.25% compared to $EFQM$ and $AQRA$, respectively.

According to on-demand applications, we switch from default master to new master thanks to a centralized entity to guarantee delay for *delay-centric* traffic. Stoping or idling least loaded controllers instead of switching between them promotes better energy-aware, even if we can achieve acceptable response time. Reducing the energy consumed is becoming an increasingly challenging research direction [24]. Therefore, we plan to explore energy consumption as well as to satisfy predefined flow response time in a distributed clustering environment based on SDN controllers.

REFERENCES

[1] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O.Koufopavlou.: RFC 7426: Software-defined networking (SDN), "Layers and architecture terminology", IRTF, ISSN: 2070-1721, pp. 1-35, 2015

[2] I. Bedhief, M. Kassar, T. Aguili, L. Foschini and P. Bellavista, "Self-Adaptive Management of SDN Distributed Controllers for Highly Dy-

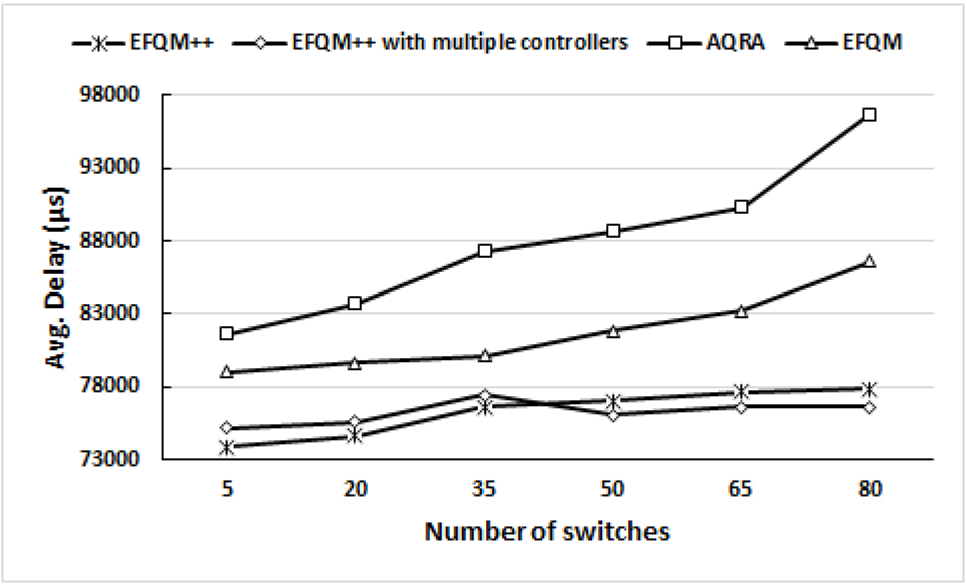

FIGURE 5. End-to-end flow performance comparison.

namic IoT Networks", In Proc. 15th IWCMC, pp. 2098-2104, Morocco, 2019

[3] F. Bannour, S. Souihi, and A. Mellouk, "Scalability and reliability aware sdn controller placement strategies", In Proc. IEEE CNSM, pp. 1–4, Tokyo, 2017

[4] K. S. Sahoo et al., "On the placement of controllers for designing a wide area software defined networks", In Proc. IEEE TENCON, pp. 3123-3128, Penang, 2017

[5] S. Khan, A. Gani, A. W. A. Wahab, M. Guizani, and M. K. Khan, "Topology discovery in software-defined networks: Threats, taxonomy, and state-of-the-art", IEEE CS&T, vol. 19, no. 1, pp. 303–324, 2017

[6] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in software defined networks", In Proc. IEEE ICSPCS, pp. 1–8, Gold Coast, QLD 2014

[7] Hasan, Dana & Othman, Mohamed, "Efficient Topology Discovery in Software Defined Networks: Revisited", In PCS, Vol. 116, pp. 539-547, 2017

[8] RYU Documentation, *https://bit.ly/3bbSOMb*. Last accessed 11 Nov 2020

[9] SDNHUB, ONOS Tutorial, *http://sdnhub.org/tutorials/onos*. Last accessed 2 Dec 2020

[10] A. Bassene, B. Gueye, "An Enhanced Flow-based QoS Management within Edge Layer for SDN-based IoT Networking", In EAI AFRICOMM, December 2020.

[11] S. Peros, H. Janjua, S. Akkermans, W. Joosen, and D. Hughes, "Dynamic QoS support for IoT backhaul networks through SDN", In Proc. IEEE FMEC, pp. 187–192, Barcelona 2018

[12] G. Tangari, D. Tuncer, M. Charalambides, Y. Qi, and G. Pavlou, "Self-adaptive decentralized monitoring in software-defined networks", In IEEE TNSM, vol. 15, no. 4, pp. 1277–1291, 2018

[13] G. Deng and K. Wang: "An Application-aware QoS Routing Algorithm for SDN-based IoT Networking," IEEE ISCC, pp. 00186-00191, Natal, 2018

[14] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem", In Proc. ACM FWHTSDN, pp. 7–12, Finland 2012

[15] A. S. Muqaddas et al., "Inter-controller traffic in ONOS clusters for SDN networks", In Proc. IEEE ICC, pp. 1-6, Kuala Lumpur, 2016

[16] Shah, Syed et al., "An adaptive load monitoring solution for logically centralized SDN controller", In Proc. APNOMS, pp. 1-6, Kanazawa, 2016

[17] A. L. Stancu et al., "A comparison between several Software Defined Networking controllers", In Proc. TELSIKS, pp. 223-226, Serbia, 2015

[18] Mininet-Wifi, Emulator for SDN Networks, *https://bit.ly/3tp3QpN*. Last accessed 24 Oct 2020

[19] Islam, Md Tariqul & Islam, Nazrul & Refat, Md., "Node to Node Performance Evaluation through RYU SDN Controller", In WPC, Vol. 112, pp. 555–570, 2020, 10.1007/s11277-020-07060-4.

[20] Dugan, J. et al., "iPerf-The ultimate speed test tool for TCP, UDP and SCTP", línea]. *https://iperf.fr*. Last accessed 23 May 2020

[21] L. Mamushiane et al., "A comparative evaluation of the performance of popular SDN controllers", In Proc. WD, pp. 54-59, Dubai, 2018

[22] J. Ali et al., "QoS improvement with an optimum controller selection for software-defined networks", PLoS One, vol. 14, no. 5, pp. e0217631, 2019

[23] Abdelaziz, Ahmed et al., "Distributed controller clustering in software defined networks", PLOS ONE, Vol 12, pp. e0174715, 2017

[24] F. Balde et al., GreenPOD, "Leveraging queuing networks for reducing energy consumption in data centers", In Proc. ICIN, pp. 1-8, Paris, 2018