Can QoS be dynamically manipulated by the end-device initialization?

Fragiskos Sardis, Massimo Condoluci, Toktam Mahmoodi, Mischa Dohler
Centre for Telecommunications Research
Department of Informatics
King's College London, UK
[fragkiskos.sardis|massimo.condoluci|toktam.mahmoodi|mischa.dohler]@kcl.ac.uk

***Abstract*—The traffic growth as well as the diversity in Quality of Service (QoS) requirements are drastically increasing the challenges in achieving effective network management. The need to handle dynamic QoS requests exacerbates such challenges and dictates novel solutions able to make network management more flexible and adaptable to traffic changes. To this aim, in this paper we propose a novel approach aiming at dynamically managing the QoS within the Software-defined Networks through flow initialization and termination requests from the end-client. Our approach allows management and control planes to be informed as soon as possible about the changes in the traffic requirements of the network, thus introducing a more flexible (re)configuration of the network according to the actual traffic demands. Such flexibility is particularly interesting within the next generation networks that aim to serve diverse set of vertical industries. While the full picture of the network architecture is depicted in this paper, we present the prototype implementation of this novel approach. The latency and overhead as introduced by the proposed approach in this paper are studied using the prototype implementation.**

***Index Terms*—Software-defined Networking, Quality of Service, network resource allocation; QoS Broker.**

I. Introduction

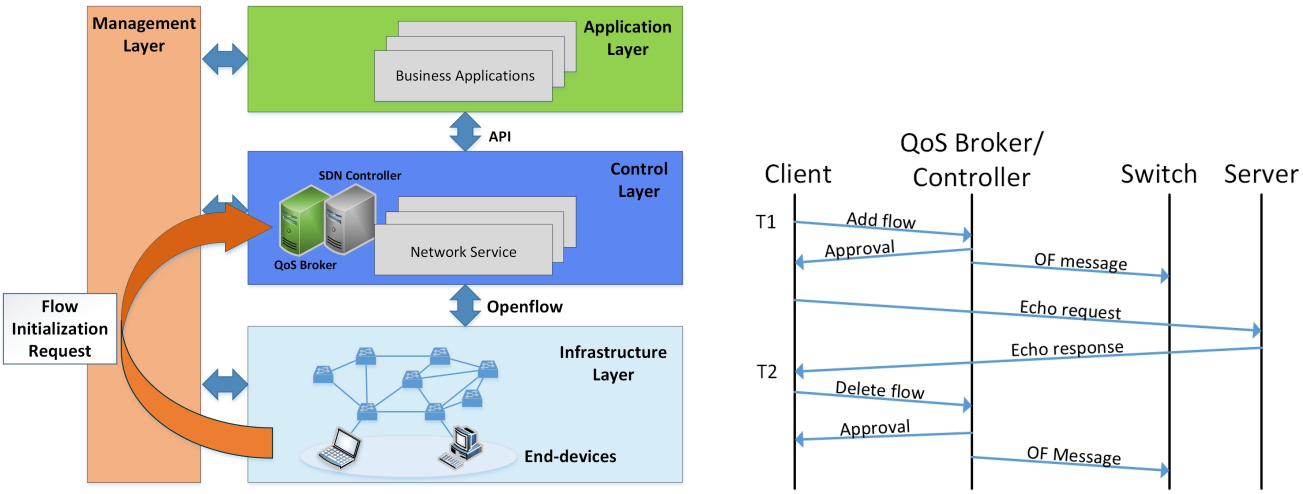
The effective and quick adaptation of resources to the actual traffic demand is one of the main features expected to be effectively handled by next-to-come networks [1]. Guaranteeing *flexibility* in traffic management while simultaneously supporting strict and *dynamic* Quality of Service (QoS) requirements is yet critical and challenging. A step forward in introducing flexibility in network management is represented by the Software-defined Networking (SDN) paradigm [2], where control and data planes that traditionally reside on the same device are decoupled and split into logically centralized network intelligence and an underlying infrastructure which is abstracted. As a consequence, the SDN control is directly programmable. Furthermore, abstracting control from forwarding allows administrators to dynamically adjust network-wide traffic flow to meet varying requirements.[1]

Among the various key features of the SDN, the programmability and agility of the network (re)configurations can significantly ease management of diverse set of QoS requirements. The main architecture design of SDN consists of three layers representing the *infrastructure*, *control* and *application* layers. Alongside the three main layers of the SDN architecture, a fourth vertical layer is interfacing with the main layers of the architecture to provide control and management functions, i.e. the *management layer* [3], [4]. While management functions reside on all planes, in the controller plane, management functions configure the policies that define the scope of control given to the SDN application and monitor the performance of the system. The legacy SDN relies on the management plane to set/manage/modify the rules exploited by network switches to handle QoS [2]. Therefore, end-devices are not able to interact with the network in the control plane and their role is limited to transmit/receive data packets. This will further limits the adaptability of the network, since dynamical QoS changes need to be contracted by the management/control plane, which could directly affect the interval between the time new QoS is requested and the time network is ready to manage it.

To this end, we propose a novel approach for dynamic QoS management in the SDN where the request of a flow initialization is triggered by the end-devices. Within this new design, the flow is initiated with specific QoS request for a particular End-to-End (E2E) communications path by the end-device and will be terminated (again by the end-devices request) upon the termination of the actual communication. Such technique will allow a common ground in providing diverse and dynamic QoS to the range of vertical industries that are sharing the same network infrastructure in the vision of next generation networks. In vertical application domains such as Industry networks, reservation of the E2E communication path with specific QoS that is also performed with high agility is of significant importance. Our proposed approach aims to achieve such agile QoS (re)configuration by allowing end-devices to directly interact with the management/control planes. Furthermore, through explicit flow removal messages sent by the end-devices, our approach allows to dynamically adapt the number of entries in the flow table of switches within the infrastructure layer. The reference architecture for discussion in this paper is shown in Fig. 1, that is depicted based on the Open Networking Foundation (ONF) architecture with the modifications proposed in this paper.

The remainder of this paper is organized as follows. Section II will provide an overview on the QoS management within the SDN and will further discuss the limitations of the current

[1] https://www.opennetworking.org/sdn-resources/sdn-definition



(a) End-device initiated flow initialization

(b) Communication diagram among involved entities

Fig. 1. SDN enhanced by end-device flow initialization.

approaches in the literature. Section III will introduce our proposed solution, whose prototype implementation is detailed in Sec. IV. Section V will discuss the preliminary results for flow initialization achieved with our prototype, while last Section will conclude the manuscript and will focus on the future work.

II. QoS MANAGEMENT IN SDN

A. OpenFlow and QoS management

OpenFlow [5] is the most consolidate communication interface between the control and infrastructure layers. It is used to access and manipulate the configuration of infrastructure devices by installing/removing *rules* in the switches' flow tables. These rules are initially programmed on the SDN controller using northbound Application Programming Interfaces (APIs) (please, refer to Fig. 1) and are then pushed onto infrastructure devices using OpenFlow in order to set up traffic flows on the switches and routers. In this way, the forwarding plan is configured to perform various functions such as conditionally forwarding packets based on their arrival rate at the switch.

The QoS can be managed according to two different solutions: *(i)* through OpenFlow; *(ii)* through the SDN controller that manipulates the rules installed in the switches. Focusing on the first solution, QoS in SDN can be implemented at the infrastructure layer by using two different OpenFlow features that are queues and meters [6]. Packets belonging to a specific flow can be set to a queue on the switch which has a pre-configured transmission rate on an output port. The field associated to the *type of service* in the IP header can then be used alongside queues to implement QoS control mechanisms such as DiffServ. Meters deal with the idea of setting a transmission rate threshold which can be used as a limiter or to trigger other functions once the rate is exceeded. OpenFlow uses meter tables, similarly to flow tables where various meter bands can be defined and packets may be assigned to them. Furthermore, a packet may be assigned to multiple meters, each one performing a different function once its threshold is reached. So, meters in combination with queues can be used to perform complex QoS manipulation functions.

Handling QoS management through such mechanisms has been studied in the literature and a summary of those are given in the following.

B. State of the Art in QoS Management

The provisioning of end-to-end QoS requirement across Internet is yet challenging. It is for example complex for service providers to implement their strategic and business policies dynamically. Differentiated Services (DiffServ), Integrated Services (IntServ) and Multiprotocol Label Switching (MPLS) can provide QoS guarantees at different levels of confidence and granularity, however they still don't allow for a specific application to request a certain level of QoS for E2E communication. In this direction, SDN is giving an opportunity for such flexibility and dynamics. Although the controller in SDN has the ability to define data flows, the QoS management for end-users still remains a challenging task because the SDN controller does not provide QoS policy and management mechanism to define constraint on an E2E flow. In this direction, a framework for enforcing policies in the SDN network is introduced by [7], entitled "PolicyCop". This framework allows SDN to automatically perform network reconfigurations based on statistics gathered from the network elements and their comparison to the service level agreement or the QoS parameters. Although using this scheme, the QoS of a particular flow or aggregation of flows can be dynamically adjusted, an increased packet loss will be initially experienced. The resulted packet loss is studied in [8], which mainly occurs because packets are allocated to different queues on the switches and throughput limits are imposed.

Another QoS management scheme aimed at multimedia traffic prioritisation is "OpenQoS" [9]. The packet header fields are used to identify multimedia traffic. The prioritisation

is achieved by querying the status of all forwarding devices on the network in order to identify the current load on each one. Path computation takes into consideration the load on each forwarding device and routes multimedia packets accordingly in order to satisfy the required QoS. Similar to PolicyCop, in the "OpenQoS" the traffic classification and assignment to a specific QoS level is done based on predetermined policies on how packets should be forwarded. Once a path is calculated, frequent measurements on the load of forwarding devices are used as a feedback mechanism in order to identify if the path should be recalculated.

Compared to PolicyCop, OpenQoS has a more limited scope in terms of QoS mainly because of basis of measurements being the link utilisation on forwarding devices rather than the actual QoS metrics that the multimedia application requires (latency, jitter, throughput, packet loss). PolicyCop, in contrast, uses actual throughput measurements in its experimental implementation and demonstrates the ability to reroute flows when a policy violation occurs. For both of these schemes, policies are used to classify traffic and assign it to a set of QoS parameters that need to be satisfied. Due to the flexibility offered by SDN, it would also be possible to create a module where applications on the clients may send their exact QoS requirements so that detailed of the E2E flow may be created and tighter control of QoS can be applied.

The work in [10] discusses the requirements of a northbound API for SDN that will satisfy real-time dynamic QoS adaptation in Real-Time Online Interactive Applications (ROIA). Authors in [10] argue that ROIA may have varying QoS requirements from the network depending on the state of an application and they use online games and e-learning as examples of how the state of the applications and participant interactions with it translates into higher or lower throughput and latency on the network. They propose a northbound API for SDN that will allow ROIA applications to communicate with the controller and to define the level of service required from the network.

On a more similar ground to the proposed solution in this paper, a QoS management mechanism is discussed in [11], where the authors propose a controller reference architecture that allows client applications to make QoS requests to the controller. These requests are scheduled by the controller to the appropriate network devices based on available topology for controlling the level of service that applications receive in real time. However, we put one step forward in dynamic QoS adaptation over the E2E path by extending the responsibility of requesting communication flows to the client as well as defining the desired QoS for them. This effectively means that by default the network will not provide any connectivity to the clients and no reachability to local or remote hosts is required (with the exception of controller itself).

C. New Opportunities for QoS management in SDN

The capability of SDN to make the network programmable through northbound APIs and its inherent separation of data and control planes presents the opportunity of rethinking how communication is established and carried out. Nevertheless, although control/data splitting aims to achieve network (re)configurability and flexibility, the provisioning of QoS flows pushes new issues and challenges in terms of dynamic QoS management as (potentially large) set of rules for has to be designed for each flow, a-priori. At present, devices do not control the policies and forwarding rules of the networks they are part of, as this is performed in the management plane. This limits the flexibility of the QoS management in current SDN networks in number of ways.

The QoS rules need to be defined by the management plane and installed in the switches through the southbound interface. For example, switches may need to store huge number of rules this may involve performance degradation during packet forwarding as, for each packet, the switch has to search the proper rule among a large set of candidates. The use of timeouts in flows can assist in minimising the number of flows in tables by setting expiry times either relative to the last packet received that matches the flow or absolute time-to-live for the flow.

How flows are installed on the forwarding devices is another important aspect affecting performance of OpenFlow. A packet arriving on an OpenFlow switch that does not match any of the existing flows in the tables, will have its header forwarded to the controller over the southbound interface. This can cause congestion on the southbound interface on a network with a large number of diverse flows. This issue has also been addressed in the literature and opportunities to improve performance are discussed. For example, in [12], and in the context of mobile networks, a redirection-based rule-sharing scheme for SDN switches is proposed, where neighboring forwarding devices can contact each other in order to update their flow tables. This scheme can partially mitigate the effects of fine-grained QoS control on E2E communication which calls for individual flows to be installed on switches for each pair of communicating devices.

III. End-Device Flow Initialization

Current SDN relies on the management plane to set the rules on how traffic is forwarded and QoS is applied. The end-devices are not able to interact with the network in the control and management planes in order to request QoS characteristics and influence how their traffic is being forwarded. However, SDN provides the ability to create such functionality through the use of controller modules. In this paper, we propose a novel approach suitable for dynamic QoS management in SDN where end-devices can request *(i)* QoS parameters according to the application needs and *(ii)* flow instantiation so as to satisfy these requirements. As depicted in Fig. 1(a), we assume a logical connection exists between end-devices and the management/control plane, and can be used to request/set-up as well as to delete flows directly by the end-devices. Such client-initiated flows with requested QoS over an E2E path is of significant importance in various domains and where not only deterministic QoS performance is expected from the

network but also dynamic QoS management is needed[2].

Assuming a network where all devices and software are under a single administrative control (such as SDN), allowing some autonomy at the end-devices could potentially improve the efficiency of the network by: *(i)* enabling strict E2E communication on a need-to-have basis; *(ii)* deterministically distributing the network resources according to client demands; *(iii)* giving fine-grained control over E2E communication paths; *(iv)* hiding devices from the network when they are not communicating by explicitly removing the flow at the controller and involved switches once it is terminated; *(v)* reducing the number of rules to the simultaneously managed by network switches by explicitly removing the flow at the controller and involved switches once it is terminated.

The proposed model here, introduce a new SDN controller module (or a new application) which plays the role of *QoS Broker* in the network. As shown in Fig. 1(b), end-devices should immediately be able to contact the QoS Broker in order to request the level of service they need for their applications. This can be done at different levels of granularity such as at device level or at application level. The request could be in form of a "service level bundle", where QoS metrics are predefined in standardised policies or in the form of "explicit QoS values" for e.g., throughput, latency, jitter and packet loss. The criticality of the communication should also be reported to the controller for prioritization purposes, in particular if network lacks sufficient resources to satisfy QoS for all flows.

A. System-level considerations

For the overall system to function, requests from devices will have to be orchestrated so that flows can be aggregated if possible and resource allocation can be orchestrated efficiently. This will allow QoS requirements by applications to be met without overloading the underlying infrastructure. Although it may be simpler in terms of implementation for the end-devices to request a QoS level rather than individual values for each metric, the latter would allow for a more efficient orchestration and management of resources due to the closer matching that can be achieved between demands and network resources. Another important aspect to consider is security so that only authorised devices can use the network and they can only do so in compliance with access policies in order to protect the functionality and reliability of the network.

B. QoS Broker

On the controller side, the *QoS Broker* module (depicted in Fig. 2) is responsible for allocating network resources on devices and should maintain a view of the network to offer a simple interface for end-devices to request resources. The requested QoS from an end-device for a given E2E communication path should then be translated by the QoS Broker into network paths and flows assigned to the OpenFlow meters and queues. In a simple implementation, the QoS Broker may accept or reject a flow based on current network load, however more complex policies can be used to prioritise the communications based on their criticality. The QoS Broker can decide to satisfy the QoS for a critical flow and demote other flows to the best-effort delivery. The QoS Broker must always be aware of the available resources on the network as well as the status of flows for the purpose of monitoring if QoS is being maintained. Depending on the application, the QoS Broker may offer soft or hard QoS guarantees which should be monitored after flow establishment to ensure guarantees are met. Such monitoring can be achieved by retrieving this information directly from the forwarding devices. However, in our proposed model, the QoS Broker is already in contact with the end devices as well and separate agents on these devices may also assist in confirming the QoS status for their connection. For example an application on a client device may report a QoS failure if the round-trip time does not match the requested latency.

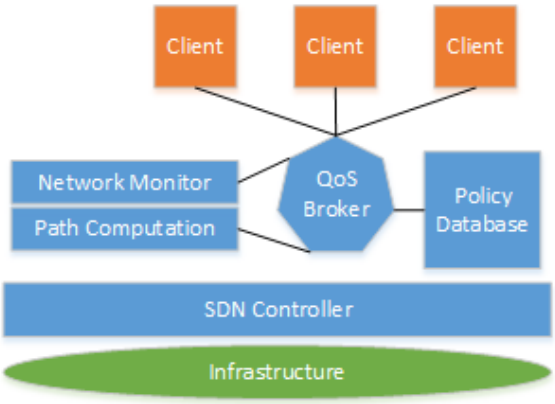


Fig. 2. Our proposed QoS Broker entity with related connections.

C. Design Implications

There are three key aspects and implications of the proposed model that require attention and they are related to the initial delay of establishing communication, the security implications and the additional load on the SDN controller from the client requests. Our approach introduces a delay in the establishment of communication between devices because it implies that a device is not ready to transmit/receive packets at all times. Instead a device will have to first request a communication channel from the QoS Broker, the path will have to be calculated and finally the flow will have to be generated and installed on the controller and forwarded to the OpenFlow switches. This additional overhead occurs once, before establishing E2E communication and does not interfere with the communication afterwards unless the device makes a request to alter the QoS during the communication. In such an event, there will be an initial delay before the QoS is adjusted but as

[2]An example application is exploited within the H2020 VirtuWind project.

also indicated in [8], transport protocol may also be affected by packet retransmissions and therefore wasted network resources as packets are assigned to different queues, especially if they are demoted to lower priority queues.

The security considerations for the proposed model are revolving around the fact that clients may make requests on the network that have the potential of affecting the communication of other devices. Authentication and authorization are two aspects that partially address the problem of admitting devices on the network but there is also a question of trust for devices that get authenticated and are authorized access. For example, a legitimate device may be granted access to the network, however a malicious user or application may make requests for very high QoS that can affect other devices. Security policies may address this problem to an extent by identifying the maximum level of QoS that may be allocated to a particular device or application, however it is evident that strict control is required not only on the hardware and operating system but also on the applications running on these devices.

Finally, it should be noted that potentially the SDN controller may become overloaded with client requests which will cause delays to setting up new flows or modifying the QoS of existing ones, therefore having adverse effects on the network's functionality and performance. Similarly, if QoS is set up per E2E flow, on a large SDN-based network, this may lead to a very high number of flow entries on the controller and switches which also negatively impacts the forwarding performance of the network. However, depending on the granularity of flow definitions and on the algorithms used for flow aggregation and management, it may be possible to partially mitigate the effects of large flow tables on forwarding devices. In the next section we measure the performance implications of this model using a Raspberry Pi. The aim of this experiment is to identify the impact of end-device driven flow instantiation on communication establishment, in terms of latency and overhead.

IV. Prototype Implementation

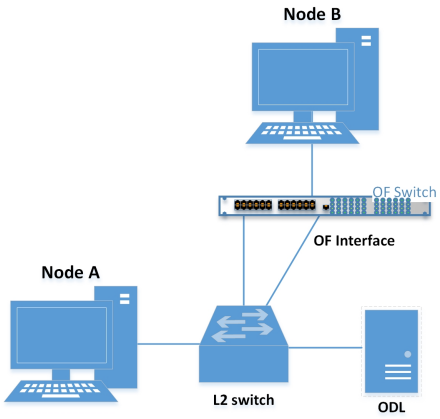


Fig. 3. Setup of prototype implementation

The prototype platform's objective is to measure the overhead introduced to an end-user application from configuring the network before establishing a connection. At this stage the prototype does not dynamically manipulate the QoS of the flows it creates and does not use any matching rules to establish flows explicitly between the IP addresses of the two nodes. Instead, the focus is placed on how long it would take for the application to construct the simplest configuration for a flow, send it to the controller and receive a response from the controller before it starts using the network. Flows are set up on OpenDaylight (ODL) via Representational State Transfer (REST) in JavaScript Object Notation (JSON) format for the parameters using basic authentication. Flows are submitted to the configuration data store in ODL which then calls an Remote Procedure Call (RPC) to install the flow in the operational database.

A. Network Setup

The prototype implementation consists of a Pica8 P-3290 White-Box OpenFlow switch[3], a laptop running ODL Lithium SDN controller[4], two Raspberry Pi 2B[5] nodes serving as the communicating devices and a legacy Layer 2 switch. ODL connects to the management interface of the P-3290 and to one of the Raspberry Pi nodes via the Layer 2 switch. The Layer 2 switch also connects to one of the managed interfaces of the P-3290. Finally, the second Raspberry Pi node connects directly to one of the P-3290 managed interfaces. Fig. 3 illustrates the physical layout of the test platform.

In order to simplify the setup, all devices are configured with IP addresses from the same subnet and P-3290 is configured with a single bridge consisting of the two managed interfaces that connect to the Layer 2 switch and Node B. P-3290 is running in Open vSwitch (OvS) mode with no flows pre-installed on the switch or ODL. During testing, the devices have exclusive control of the network and the controller with no other devices present on the network. The Raspberry Pi is chosen as an example of embedded devices that can be found in sensor networks and its performance in this scenario can give us a better indication of the communication overhead that this approach introduces. This setup choice eliminates the need for using an Dynamic Host Configuration Protocol (DHCP) server along with default flows to allow the devices to acquire and IP address upon joining the network and also ensures that minimal configuration is needed with no previous states stored in the system in order to make it function. Consequently, this speeds up the testing process and eliminates configurations that may impact the system's performance and allows us to focus on the best-case response time we can get from the controller in our setup.

[3]Pica8. Products-Available Switches. http://www.pica8.com/products/pre-loaded-switches

[4]Linux Foundation. Opendaylight https://www.opendaylight.org/lithium

[5]Raspberry Pi Foundation. Raspberry Pi 2 B https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

B. Prototype Software

For testing the functionality of the prototype we use Python 2.7 running on Raspbian Wheezy[6]. We run a Python script on Node B that sets up a listening TCP port with 4KB buffer and accepts incoming connections, places incoming data to a buffer and then discards it. On Node A, we run a Python script that first contacts ODL to set up a flow and then transmits data to Node B. While data is being transferred, the script measures the achieved throughput and we define an acceptable threshold for the throughput which is used to determine if the flow satisfies the application's requirements. If the throughput falls below the threshold then communication is interrupted and the flow is deleted from ODL thus preventing any further communication between the two nodes until the script is reran.

The script running on Node A consists of three main methods. The first method plays the role of an application that is ready to transmit data by generating 1MB of random characters and storing it as a string. The second part of the script contacts the controller and sets up a predefined flow. It measures the time it takes to process the JSON string holding the flow configuration, transmit it to the controller and receive a response from the controller. Once the response from the controller is received, the third part of the script transmits the data stored in the buffer to Node B while measuring the time it takes to send the full buffer. The transmission time is then used to calculate the throughput. This part of the process loops infinitely by invoking each time the first method to generate more data and transmit it. If the throughput result falls below the defined threshold, the loop breaks and a method for deleting the flow from the controller is invoked.

V. Methodology and Results

Performance results are recorded over ten runs. The response time of setting up the flow is measured as the time difference between the transmission of the REST message to the controller over HTTP and the moment that HTTP OK is received on Node A. This approach guarantees that data transmission between the Nodes will not begin before the controller has successfully accepted the flow and allows us to directly measure the overhead of setting up the flow before communication can occur. From the perspective of an application that wants to transmit data, this delay represents the total delay experienced from the moment the data is placed in the buffer until it is ready to be transmitted. Therefore, this delay represents the overhead introduced by the mechanism that generates the flow's parameters as well as the response time of the controller.

The flow used in our testing is a generic flow that forwards traffic as an Layer 2 switch normally would, without introducing any Ethernet and IP matching rules or performance queues. The options for the flow are set in the software using variables rather than storing the entire flow as a single string. To ensure that the transmission loop terminates, we set the threshold to a value of 7 MB/s where due to processing overheads on the Pi, it is likely that it will be violated, resulting in termination of the transmission and flow deletion. Using these settings we measured an average response time of 80 ms for setting up a flow on the controller and 20 ms for deleting the flow over ten repetitions of the test. Using POSTMAN on the same computer running ODL, we measure an average response time of 43 ms for pushing the same flow to the controller which indicates that the extra 37 ms measured on the Pi are introduced by hardware performance and the extra work needed to construct the JSON string for the flow. The novelties of the proposed approach altogether with the main features and results are summarized in Table I.

It is worth noting that the current implementation only measures throughput for a particular application and not the overall throughput achieved by the network interface. In an environment where the application has exclusive use of the network, this is not a problem; however, in scenarios where multiple applications are using the network interface, the performance threshold for a particular application may not be reached due to contention on the local interface rather than due to core network performance. This is evident in our implementation where the Raspberry Pi has Fast Ethernet interface while the P-3290 is capable of Gigabit Ethernet. One way of eliminating such false alarms in a fully functional system would be to implement an additional check on the network in order to confirm that the degradation has occurred due to network load. Alternatively, on the client side a second check can be made on the status of the Network Interface Card to ensure that it is not being overloaded by other applications. A second drawback that becomes apparent through this implementation is that CPU performance on the Pi can also affect the measured throughput and therefore despite the application having exclusive access to the network interface, any applications running in parallel, including the operating system's processes, can load the CPU and affect the performance of the networking stack thus causing the threshold to be violated. Once again, this can be tackled by making a second check on the network to make sure no forwarding devices are overloaded.

VI. Conclusions and Future Work

In this paper we proposed a horizontal approach in SDN where end-devices initiate E2E communication by communicating the required QoS to the network. This is achieved by using a QoS Broker module that is responsible for receiving end-device requests, validating, translating into flows and submitting them to the controller. We discussed the security and performance implications and focused on measuring the impact of the flow setup delay additions to the communication latency of the devices. Delay was measured experimentally using embedded devices as clients that communicate directly with the SDN controller. This setting provided us with the best-case response time without considering any additional delays introduced by the QoS broker. We also demonstrated how QoS measurements on the client side may be used to alter the configuration of the network in order to terminate

[6]Raspberry Pi's Debian Wheezy implementation.

TABLE I
SUMMARY OF THE PROPOSED END-DEVICE TRIGGERED FLOW INITIALIZATION

Novelties	End-device input towards establishing flows and QoS End-device feedback mechanism for QoS failures
Architectural modifications	Inclusion of QoS Broker entity to manage client requests API for the communication between QoS Broker and clients Client access policy database Orchestration and flow aggregation modules for flow table management
Results	80ms avg. flow instantiation time from Raspberry Pi 43ms avg. flow instantiation time from PC running the controller 37ms client overhead in generating the flow request
Concluding Remarks	Controller response time and scalability are important Security should be addressed to prevent network misconfiguration by clients Flow setup time may delay E2E communication initialization

the communication or manipulate the QoS. Our findings show that delay in establishing a communication is dependent on the clients' processing power as well as the amount of information passed from the clients to the QoS Broker.

Based on our findings, the planned future work will focus on the design and prototype implementation of the QoS Broker along with APIs for communication with the clients and the controller. The first aim will be to identify and solve bottlenecks that will increase the overhead in the communication setup phase. Second goal will be the minimization of the response time for setting up flows.

VII. ACKNOWLEDGEMENT

This work has been supported in part by the 5GPP VirtuWind (Virtual and programmable industrial network prototype deployed in operational wind park) Project.

REFERENCES

[1] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob, "Toward an sdn-enabled nfv architecture," *IEEE Communications Magazine*, vol. 53, pp. 187–193, April 2015.

[2] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, pp. 14–76, Jan 2015.

[3] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks." White Paper, April 2012.

[4] Open Networking Foundation, "SDN architecure." ONF TR-502, June 2014.

[5] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE Communications Surveys Tutorials*, vol. 16, pp. 493–512, First 2014.

[6] Open Networking Foundation, "OpenFlow Switch Specification." TS-006, June 2012.

[7] M. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba, "Policycop: An autonomic qos policy enforcement framework for software defined networks," in *IEEE SDN for Future Networks and Services (SDN4FNS)*, pp. 1–7, Nov 2013.

[8] R. Durner, A. Blenk, and W. Kellerer, "Performance study of dynamic qos management for openflow-enabled sdn switches," in *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, (In press).

[9] H. Egilmez, S. Dane, K. Bagci, and A. Tekalp, "Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks," in *Asia-Pacific Signal Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 1–8, Dec 2012.

[10] T. Humernbrum, F. Glinka, and S. Gorlatch, "A northbound api for qos management in real-time interactive applications on software-defined networks," *Journal of Communications*, vol. 9, no. 8, 2014.

[11] K. Govindarajan, K. C. Meng, H. Ong, W. M. Tat, S. Sivanand, and L. S. Leong, "Realizing the quality of service (qos) in software-defined networking (sdn) based cloud infrastructure," in *International Conference on Information and Communication Technology (ICoICT)*, pp. 505–510, May 2014.

[12] M. Ito, N. Nishinaga, and Y. Kitatsuji, "Redirection-based Rules Sharing Method for the Scalable Management of Gateways in Mobile Network Virtualization," in *IEEE Global Communications Conference (GLOBECOM)*, Dec. 2015.