Minimizing latency of critical traffic through SDN

Joan Meseguer Llopis
Orange Polska
Warsaw, Poland
Joan.Llopis@orange.com

Janusz Pieczerak
Orange Polska
Warsaw, Poland
Janusz.Pieczerak@orange.com

Tomasz Janaszka
Orange Polska
Warsaw, Poland
Tomasz.Janaszka@orange.com

Abstract— **Internet of Things becomes one of the main trends in development of future networks. It introduces new challenges to network operator, like exponential growth of the traffic, large amount of network nodes to manage or variety of services. Some IoT services include the control of critical processes such as remote manipulation of a robotic arm in tele-surgery. In such processes, keeping very low end-to-end latency in the communication is crucial. The critical traffic generated becomes important and thus it needs special treatment, especially in situations when latency may be affected (e.g. network congestion or network node malfunctioning). In this work we analyzed what are the technical possibilities to manage traffic delay for important data. We selected Software Defined Networking technology to manage the end-to-end IoT traffic. An implementation based on OpenDayLight SDN controller allowed to build a solution that is able to identify in real-time the routing path with minimum delay and to route the important data traffic to this path. In order to monitor the latency, a probe packet is sent over each path and its trip time is measured. Simulations based on changes of the physical delay on links forming paths confirmed the effectiveness of our approach. They also showed the conditions, where the solution can be applied.**

Keywords—Internet-of-Things; Software Defined Networking; Critical Traffic; Latency Management; End-to-End Delay

I. INTRODUCTION

Recent studies estimate more than 20 billion of connected IoT devices in the near future [1]. This plethora of devices will inject a huge amount and diversity of data to telecommunication networks. IoT introduces specific nature of the traffic, which is characterized by large amount of devices generating data, using of several network technologies to access network devices or nodes and enabling real-time system response in case of specific situations. The proper management of IoT traffic and keeping QoS objectives for some services becomes a critical task. It seems to be difficult to resolve those issues in existing telecommunication networks. Routing of the traffic in existing networks is static and based on fixed rules, which are predefined and cannot be flexibly adapted to the changing traffic and network behavior. Prioritization of the traffic may be also insufficient.

Telecom operators deal with management and quality challenges using several solutions. But in practice the most simple and effective approach to mitigate negative consequences of traffic growth is providing overestimated network capacity. It allows to carry the traffic assuming, that even if e.g. congestion happens, the traffic is held with only slight and temporary QoS degradation.

But for some IoT services such solutions are not sufficient. When IoT service is applied for processes like remote manipulation of a robotic arm in tele-surgery or road monitoring, the transmitted traffic may require special treatment for dedicated periods. Taking the above examples it can be direct realization of part of an operation by a doctor in a remote location or detection of a car accident. Both require higher video quality and minimal latency. If such situation happens, data become critical for a certain period of time, during which they need to be managed with special treatment. Important data treatment solution should allow to carry traffic with guaranteed quality parameters, and not only using overestimated capacity or prioritization.

One of the parameters of important data treatment solution, which we selected for analysis in this paper, is end-to-end delay. Delay can be managed with regards to all network flows related to data identified as important during certain period of time. The presented solution analyzes available network paths and dynamically calculates delay associated with the paths. Network management system redirects the traffic to the path with minimal delay. It also monitors the current status of the network and in case of change of path's delay applies alternative paths for the flows requiring special treatment. If IoT service stops demanding special treatment, the traffic management is returning to normal conditions. Considering emerging technologies to manage networks like Autonomic Network Management [2], SDN [3], Network Function Virtualization [4], Network as a Service [5] and others, we selected SDN as it has global view on the network and is able to orchestrate end-to-end traffic. We wanted to verify if SDN can be effectively used for important data treatment solutions. We chose OpenDayLight Lithium as SDN controller and Mininet as network simulation environment.

The paper is structured as follows: in section 2, the delay minimization method is described. In section 3, we described the architecture of the solution and test environment used for the evaluation of the solution. Results of evaluation and its further discussion are presented in section 4. Finally, section 5 provides main conclusions resulting from this work and proposes some guidelines for future work in this area.

II. DELAY MINIMIZATION METHOD

The method of delay minimization for IoT data flows between IoT nodes follows the concept of MAPE (Monitor-Analyze-Plan-Execute) control loop [6]. In our method, the process is made of several SDN operations and it has been decomposed into three main functions: a) path resolving, b) delay tracking and c) delay management. Path resolving and

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

delay tracking functions run in parallel and they are merely responsible for monitoring activities. On top of them the delay management function consumes the monitored information for analysis and control activities.

In this paper, the term *path* refers to network connectivity elements (virtual and/or physical) – typically composed of network nodes and links – that form the way followed by a data flow exchanged between two IoT hosts (i.e. device, gateway, server, etc.). In our approach network nodes refer to OpenFlow switches. Within a path the edge nodes are labeled as *Head* node and *Tail* node while the nodes in between as *Middle* nodes. Nodes are connected by one or more *links*. This is shown in Fig. 1.

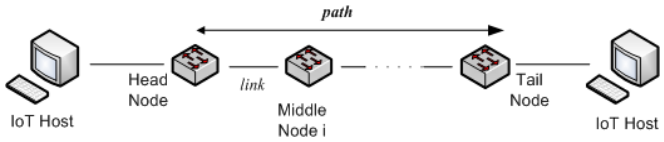

Fig. 1. Illustration of a path between two hosts

A. Path Resolving

This function is in charge of discovery of the underlying network topology between IoT hosts. More precisely, this function creates a catalog of available paths between the Head and Tail nodes. In order to build a catalog – among the whole inventory of hosts, nodes and links visible by the SDN controller – this function picks up: a) the "hosts" representing the IoT source and destination nodes of data flows; and b), the Head and Tail nodes common for all the available paths. Afterwards, the catalog of paths is built and it contains a list of links (sequentially ordered) between the Head node till Tail node (see Fig. 1). Each link is characterized by the *source* and *destination* elements, and contains information like node and termination point identifiers (i.e. port). The path catalog returned by this function resembles a data tree format like shown in the Fig. 2.

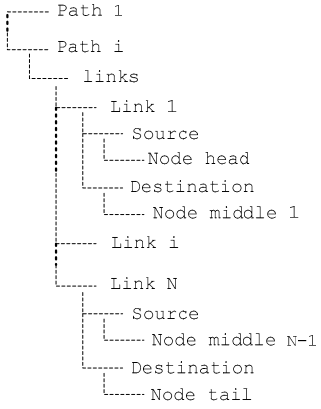

Fig. 2. Tree representation of a path catalog

In our work this module does not intend to be the key element of the solution. The algorithm used for computing paths, that connect the pair of hosts, follows a brute-force search approach that is only capable to discover paths in networks with a ring-like topology. Therefore, the algorithm iterates over the list of links that the SDN controller keeps in its topology inventory. The core part of the algorithm performs iteratively the following steps in order to build a path:

1) `SET` *current_tail* value to Head node
2) `GET` next link from Controller's catalog of links
   a) `SET` *current_link* value to this link
3) `IF` *source* of *current_link* `EQUALS TO` the *destination* of *previous_link*
   a) `THEN`
      i) `ADD` link to the *current path*
      ii) `IF` *source* of *current_link* `EQUALS TO` the Tail node
         (1) `THEN`
            (a) `ADD` *current path* to the catalog
      iii) `ELSE`
         (1) `SET` *current_tail* value to *destination* of *current_link*
4) `ELSE`
   a) `SET` *previous_link* value to *current_link*
   b) `GOTO` point *2*

B. Delay Tracking

This function aims to measure the (near) real-time latency of network paths. That is an estimation of the delay that a data flow would suffer between the Head and Tail nodes of monitored path. The used approach consists of a similar technique to the one proposed by K. Phemius et all in [7]. It is based on sending "probe" packets periodically through the path and measuring their travelling time. This approach is pictorially shown in Fig. 3.

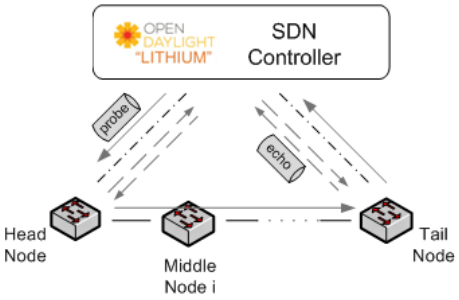

Fig. 3. Path delay measurement approach

The application that implements this function runs on top of the SDN controller. It firstly sends the probe packet to the Head node, which immediately routes the packet to the port where the given path begins. The packet travels across the links and Middle nodes of this path until reaches the Tail node. At this point the packet is forwarded up to that application through the controller (solid line arrows in Fig. 3). Packet's total travelling time is calculated by subtraction of packet's transmission timestamp (TS) to the timestamp of its arrival back at the application. But, in order to calculate the final delay value, the time spent by the packet in the controller-switch-controller trip (a.k.a southbound latency) must be discounted from the total travelling time. This time is measured by sending "echo" packets to the Head and Tail nodes (dashed line arrows in Fig. 3) and calculating the timestamp difference from packet's sending to its later reception. The final path's latency is calculated as follows:

$$Delay_{final} = \left(TS_{probeRx} - TS_{probeTx}\right) - \frac{\left(TS_{echoHeadRx} - TS_{echoHeadTx}\right)}{2} - \frac{\left(TS_{echoTailRx} - TS_{echoTailTx}\right)}{2}$$

In the above calculation we assume that southbound link latency is symmetrical, so that it equals to the half of echo packet's round trip time (RTT). In order to allow the probe packet to follow the way defined by a path the appropriate programming of the flow rules within the network nodes (i.e. OpenFlow switches) is needed. Delay Tracking function also generates some basic statistics (e.g. moving average) for their further exploitation by the Delay Management function.

Our approach differentiates from the one proposed in [7] mainly in the following aspects:

- The design of their approach limits the delay measurement to the link level. Since we aim to monitor latency at the level of paths – for a finite number of flows – which are composed of one or more links, this may cause a bigger overhead in traffic for southbound interface as well as in processing for the controller. Note that their technique is based on sending a probe packet for each link. In opposite to it, we send only one packet for the entire path.
- For the measurement of southbound communication delay we use the echo packets while they reused the built-in mechanism for statistics gathering by sending requests to OF switches. Our approach seems to be more realistic since it is closer to the actual probe packet life process (from its sending to its reception back at the application).
- Two different SDN controllers are used at each project. Floodlight [8] was used in their approach whereas the OpenDayLight [9] is used here.

C. Delay Management

This function aims to apply the minimal possible delay for the flows requiring special treatment between two network endpoints. This function receives as input parameters the identifiers (MAC or IP address) of these endpoints. Current implementation of the function is triggered on-demand. Once the functionality and performance of the method is proven, several solutions to identify critical traffic requiring special treatment may be used. Logic of Delay Management basically performs a delay sensitive routing, which means that it makes (near) real time decisions to select the most appropriate path among the available ones that will keep the latency as low as possible. Therefore this function is iteratively invoked. As an input it uses the topology information gathered by the Path Resolving function together with the monitored delay provided by the Delay Tracking function. Further analysis maps existing available paths with their associated current delay for a given flow. The decision which path the flow has to follow is taken by applying the standard max/min policy. Thus, the path with minimum delay is chosen and the consequent OpenFlow routing commands are applied via the SDN controller. The delay minimization function is decomposed into the following steps which are iteratively executed:

1) `GET` catalog of paths for the pair of MAC/IP addresses from Path Resolving function
2) `GET` current delay for each path from Delay Tracking function
3) `FIND` *path k*, $delay_k = Min\{d_1, d_2, \dots, d_i\}$
4) `IF` *path k* `IS NOT` the current one
   a) `THEN` apply data-flow-path-selection action
      i) Program the path in the tail and head node direction
   b) `ELSE`
      i) Do nothing

III. Architecture and Test Environment

The architecture of the testbed used for the evaluation of the presented concepts is shown in Fig. 4. In this architecture we can differentiate two main elements: the SDN controller (executed on Server 1) and the virtual network (modeled on Server 2).

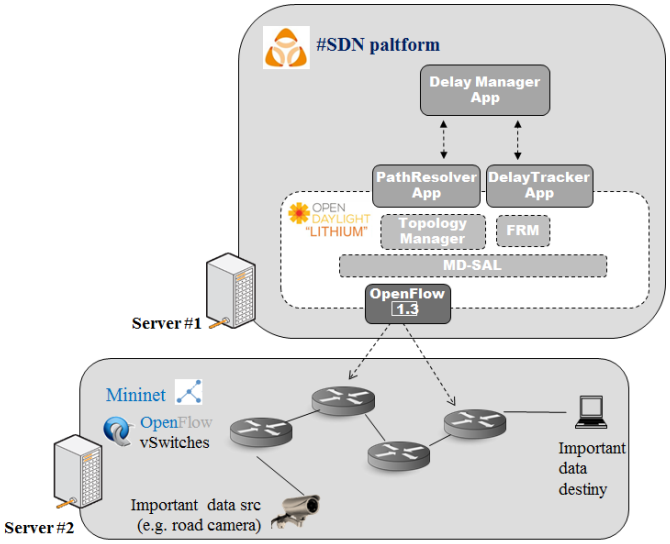

Fig. 4. Testbed architecture

A. SDN Controller: OpenDayLight and northbound applications

In Server 1 all the SDN operations are performed. For the SDN core functionalities the OpenDayLight (ODL) tool is used. The ODL can be seen as a middleware platform aiming to integrate network services such as load balancing or firewall with underlying network infrastructure. It is composed of several modules which are also known as features. The main ODL modules used in our solution are: Topology Manager, which is used for collecting information about underlying network nodes, links or hosts; Forwarding Rules Manager (FRM) that processes OpenFlow programming requests to switches; Service Abstraction Layer (SAL) that is used for consuming internal ODL services by the developed northbound applications; and OpenFlow plugin that finally applies the OpenFlow commands to switches, where OpenFlow 1.x protocol [10] is implemented. On the top of the core of the ODL we deployed three main northbound applications that realize delay management:

1) Path Resolver (PR) application: this is merely a northbound application of the ODL controller. It implements the path resolving function described in pervious section. All topology information necessary for building the path catalog is retrieved due to the SAL and Topology Manager. Its functionality is exposed over JAVA functions that are later used by the Delay Manager application.

2) Delay Tracker (DT) application: this is another SDN northbound application performing the Delay Tracking function. Within it, the packet processing service of SAL module is used for sending and receiving probe and echo packets. Additionally, the FRM and the OpenFlow plugins are used for setting the route of probe packets through the emulated network. It also exposes JAVA interfaces to share path latency statistics with other applications.

3) Delay Manager (DM) application: this is SDN northound application, which is an umbrella to the above two. It consumes the PR and DT exposed functionalities for the realization of Delay Managment function. In addition, it also uses the OpenFlow and FRM modules for applying path computation decisions by programing the switches' forwarding table(s).

B. Network Emulation: Mininet

Mininet [11] is a well-known network emulation tool within the OpenFlow community. Although it is not convenient for conducting performance tests it is suitable for the functional evaluation of SDN operations. With Mininet one can build up its own desired topology with basic transmission network parameters: bandwidth, links' delays, queues' lengths, etc. In our work two different topologies were generated: a linear and a ring.

IV. Simulations and Discussion

For the evaluation of the above presented concepts simulations have been split into two different test scenarios. First, we describe the results from the tests performed to evaluate the Delay Tracking function. Next, we show the results from the simulations applied to the overall end-to-end Delay Minimization method.

A. Delay Tracking Evaluation

For the evaluation of the DT module the Mininet network has been configured to build a topology like the one shown in Fig. 5.

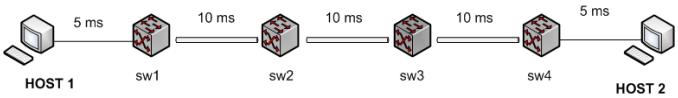

Fig. 5. Emulated network for Delay Tracking function evaluation

In this topology there is only one path between Host 1 (H1) and Host 2 (H2) which is composed of 3 links: switch1-switch2 (SW1-SW2), SW2-SW3 and SW3-SW4. The end-to-end nominal latency should be the sum of nominal latency of each link between H1 and H2. In this case, we assume that the latency values are fixed (as shown on Fig. 5) and in our calculations we also assume no delay introduced by the packet processing in each switch. Thus, for the latency values of the links, the final end-to-end nominal latency would be 40 ms. And, for the path SW1-SW2-SW3-SW4 this would be 30 ms.

In order to check the accuracy of the latency calculated by the Delay Tracking mechanism we compare it to the ping utility from terminal H1 to H2. In order to properly compare both measurements, the ping result is divided by two and from this value the nominal latency of links H1-SW1 and SW4-H2 are subtracted.

$$Ping_{final} = \frac{Ping}{2} - Latency_{h1sw1} - Latency_{sw4h2}$$

For this test scenario two different test cases have been considered. In the first one, the latency in the network shown in Fig. 5 is random, that is, the physical latency of link SW2-SW3 suffers from random changes throughout the simulation runtime. In the second one, the latency is deterministic. The applied variations to that link are 30 ms of magnitude following a harmonic pattern with a progressive period decrease (firstly 10 seconds, then 8, and so forth). Such variations allow us to check the DT behavior under sharp and fast latency changes. In order to achieve random and harmonic variations of link's latency the Linux *tc* command has been used. In both cases the delay sampling period of DT is set to half second (i.e. F_S= 2 Hz).

The chart in Fig. 6 shows the difference between the ping tool and the DT application.

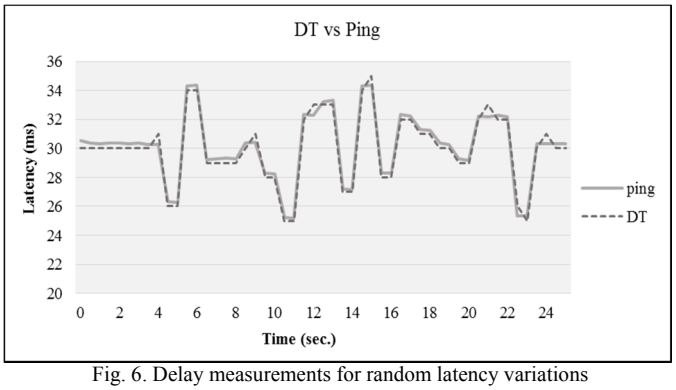

Fig. 6. Delay measurements for random latency variations

In this chart, the dashed curve represents the delay measurements gathered by the DT while the solid one represents the measurements from the ping utility. As we can observe, both latency measurements are almost the same. The dashed curve shows that DT is able to follow in quite high accuracy the (random) hops of path's latency. In fact, the maximum deviation of DT's measurements from the nominal value and from the ping tool is 1 ms and 1.35 ms respectively. The average value for this deviation is 0.231 ms and 0.352 ms respectively.

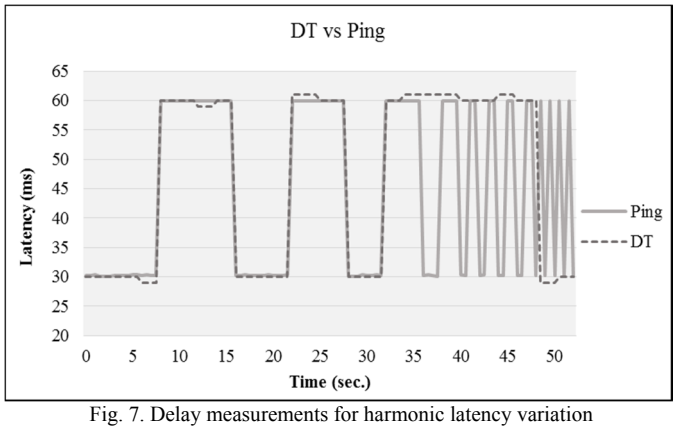

Fig. 7. Delay measurements for harmonic latency variation

For the test case with harmonic latency variations, its result is shown in Fig. 7. In this chart, we can observe that the DT performs well, especially at the three first harmonics of SW2-SW3 link's delay variations (i.e. second 0-35). However, after second 35 the latency variations reach the Nyquist frequency ($F_S/2$) where it can be noticed the information loss effect. From that moment, the dashed line shows that DT is not able to follow latency changes. From second 35 to 48, DT takes samples just at the harmonics' peak and then only the valleys are noticed.

Besides the comparison of DT and ping mechanisms we considered worthy to analyze the latency of the control channel (i.e. controller's southbound interface). That latency chart is presented in the Fig. 8. Here, we show the round-trip-time (i.e. echo_packet_RTT) of echo packets in dashed line (i.e. CTRL-SW-CTRL trip) and the one-way-trip-time of the probe packet (i.e. probe_OWTT) in solid line.

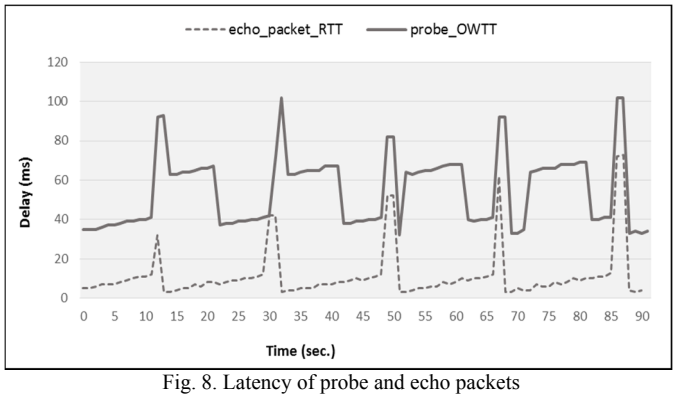

Fig. 8. Latency of probe and echo packets

Contrary to what one may think, the dashed line shows us that southbound interface latency is, in fact, far from being constant. This is clearly shown with the sharp peaks of delay suffered by probe and echo packets. It can be observed that latency peaks occur regularly with a period of 20 seconds approximately but with progressively increasing amplitudes. This fluctuation can be due to concurrent operations realized over southbound interface which may cause interface overload episodes. From the above chart we can assume that such operations are performed periodically by the controller. Moreover, such operations can be related to network flow statistics collection, topology knowledge update, etc.

B. E2E Latency Minimization Evaluation

This section describes the latency minimization method evaluation. In this case, the Mininet software has been reprogrammed to build a network like the one shown in Fig. 9.

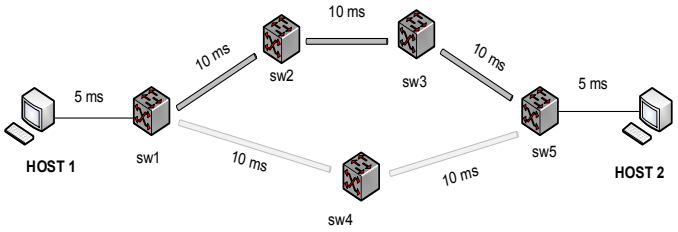

Fig. 9. Mininet network for latency management evaluation

In the above topology, the nominal latency from SW1 to SW5 should be 20 ms following the path 0 (lighter grey colorized links) and 30 ms following the path 1 (darker grey colorized links). Therefore, following a typical shortest path algorithm, path 0 would be the default path for every communication handled between HOST 1 and HOST 2. As in previous test scenario, here we also emulate a network where the links suffer from latency variations. Under such conditions, the DM module should (on-the-fly) program the OpenFlow switches' routing tables to force the flow to follow the fastest path (i.e. with lower delay) between hosts. Hence, in order to check the efficiency of the end-to-end Latency Minimization method we compare the measured delay for the flow between HOST 1 and HOST 2 when the DM module is activated against when it is deactivated.

The data flow from HOST 1 to HOST 2 is emulated via a simple ping between both hosts. The result of this ping is captured within a *csv* file in order to observe the delay suffered by the transmitted packets. Two simulation shots of 60 seconds long have been launched. One with the DM module turned off and the second one when the module is turned on. The test scenario is similar to the one for the evaluation of the delay tracker mechanism. However, this time the latency variations have 100 ms of magnitude for the deterministic test case. In both (random and deterministic) test cases, the latency variations are applied to the link that connects SW1 with SW4.

Results of both simulation shots have been presented on one chart for the easier comparison (see Fig. 10 and Fig. 11). The light grey line shows that when the DM module is turned off, the flow generated by the ping suffers the same delay changes of path 0 (i.e. light grey dashed line). On the other hand, when the DM is activated, just after a new delay peak occurs in path 0 it detects that path 1 becomes the fastest one. As a consequence, it immediately redirects the flow generated by the ping to path 1 as described in section II-C. Path switching actions are shown with the darker grey colorized curve. The algorithm reacts on the latency changes with a delay of 1 sampling period of magnitude. This is the reason for the peaks encountered at this curve. Finally, the light and dark grey dotted lines represent the linear trend of the latency which allows us to visualize the efficiency of the proposed solution.

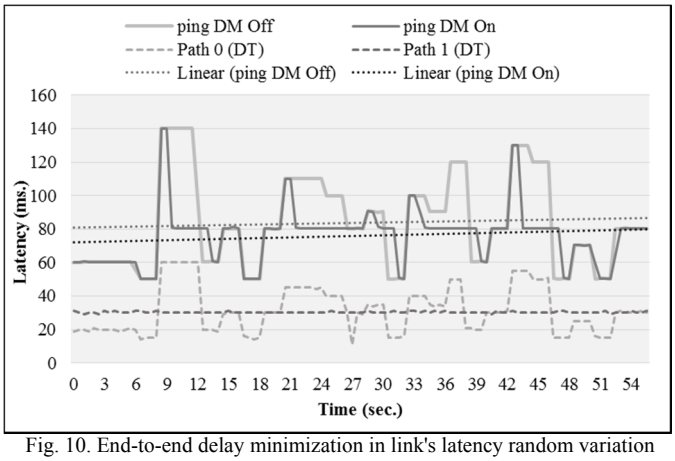

Fig. 10. End-to-end delay minimization in link's latency random variation

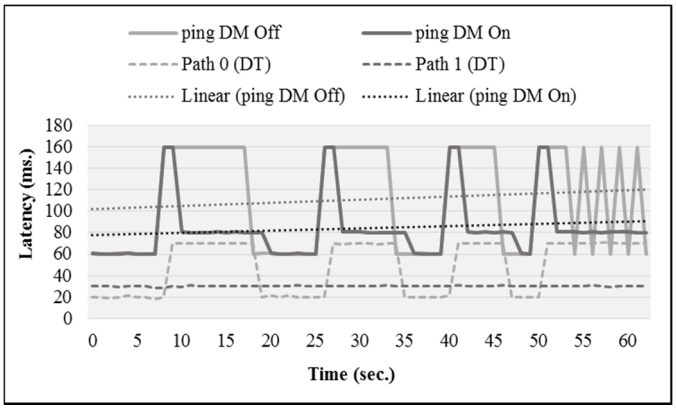

Fig. 11. End-to-end delay minimization in link's latency harmonic variation

The chart in Fig. 10 shows how the function reacts on a totally uncertain scenario. In this case, when the DM is turned off, the average latency is 85 ms. If the function is activated the average latency is 75 ms, a 10 ms reduction in latency.

As for the deterministic test case (see Fig. 11) two different observations can be extracted. The first one is a noticeable latency 'savings' achieved especially for slower latency changes (e.g. second 0-50 in Fig. 11). Indeed, the dotted lines clearly show the advantage of having activated the Latency Minimization function. In this case, when the function is turned off the traffic suffers from the worst case scenario and so average latency is 108.8 ms from second 0 to 50. And when the function is enabled the averaged latency is 78 ms (30.8 ms less). In fact, this means 18 ms above the 60 ms of delay from ideal scenario (i.e. shortest path free of latency changes), achieving reduction of 63.1 % comparing to the worst case scenario. The second observation is related to the strong dependence of such routing algorithms on the sampling frequency and in consequence on the impact on final routing decisions. This can be well noticed in the time interval 55−65 seconds. Here, the frequency of the latency variations goes beyond the sampling frequency of DT. Within these delay samples the loss of information effect occurs. Indeed, the measured delay for path 0 is constant (70 ms) though it should have similar shape to light grey solid curve. For that reason the DM has the perception that the latency for this path is all the time greater comparing to the latency of path 1. That is why path1 remains selected within this time interval.

V. Conclusions

In this work we successfully demonstrated the possibility to manage the end-to-end delay between network nodes using SDN management of flows. The proposed service benefits from its SDN nature which gives a top level view of the managed underlying network. In case of network failure or congestion (especially under conditions of lack or wrong network over-provisioning), our solution is able to discover changes in topology and path delays and to redirect the traffic to minimize end-to-end delay between nodes. Our solution is prepared for IP networks which are under control of OpenDayLight controller. In the test scenario we were able to reduce the latency by 63.1 % comparing to shortest path based routing, which is typically based on number of hops. We have also checked the efficiency of such real-time routing applications on different time scales of path's latency variability. We observed that in fast variations – i.e. close to the sampling frequency of DT – latency perceived by the DM can be far from reality, which may lead to wrong forwarding decisions. Increasing the sampling frequency may be a potential solution. In practice we observed that such solution would add noticeable overhead on southbound interface as shown in Fig. 8. Hence, future work could involve a solution that would dynamically change sampling frequency considering real-time or statistical load related information of controller's southbound interface. This would imply further analysis of controller's performance under different network conditions (load, number of nodes, etc.). Further work could also include adaptation of our solution to various topologies (e.g. mesh) with various network sizes (i.e. number of nodes, number of network instances), as well as evaluation of the effectiveness for different types and loads of critical flows. Analysis of coexistence of several instances of the network, managed by different SDN controller instances, should allow to identify the scaling capability of the proposed solution.

References

[1] Gartner, Inc., "Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015", Press Release, Stamford, Conn., November 10, 2015.

[2] ETSI GS AFI 002 V1.1.1 (2013-04) – "Autonomic network engineering for the self-managing Future Internet (AFI); Generic Autonomic Network Architecture (An Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management)".

[3] Open Networking Foundation, "OpenFlow/Sofrware-Defined Networking (SDN)," https://www.opennetworking.org/.

[4] "Network Functions Virtualization - An Introduction, Benefits, Enablers, Challenges & Call for Action", ETSI NFV Whitepaper #1.

[5] Recommendation ITU-T Y.3512 (08/2014) "Cloud computing - Functional requirements of Network as a Service".

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing", Computer 36, Issue 1, pp 41—50, January 203.

[7] Kevin Phemius, Mathieu Bouet, "Monitoring latency with OpenFlow", *CNSM*, 2013, 9th International Conference on Network and Service Management (CNSM).

[8] Project Floodlight, "http://www.projectfloodlight.org/floodlight/"

[9] OpenDaylight platform "https://www.opendaylight.org/".

[10] O.N.F., "Openflow switch specification - version 1.3.0," June 2012.

[11] Mininet: An Instant Virtual Network on your Laptop. http://mininet.org/.