Using Software-Defined Networking for Real-Time Internet Applications

Tim Humernbrum, Frank Glinka, Sergei Gorlatch

***Abstract*—We consider an emerging class of challenging Internet applications called Real-Time Online Interactive Applications (ROIA). Examples of ROIA are multiplayer online games, computation- and interaction-intensive training, simulation-based e-learning, etc. These applications make high QoS demands on the underlying network which involve the number of users and the actual application state and, therefore, vary at runtime. In traditional networks, the reconfiguration possibilities of the network to meet the dynamic QoS demands of ROIA are limited due to the lack of control of the network. The emerging architecture of Software-Defined Networking (SDN) decouples the control and forwarding logic from the network infrastructure, making it programmable for applications. This paper describes the specification, design and implementation of a novel Northbound API for developing ROIA that can use the advantages of SDN. We describe the basic architectural design of the SDN Module which implements the API functionality required by ROIA, and we report experimental testing and evaluation results for its prototype implementation.**

I. Introduction: ROIA and SDN

We consider an emerging class of challenging Internet applications called *Real-Time Online Interactive Applications (ROIA)*. These are networked applications connecting a potentially very high number of users who interact with the application and with each other in real time, i.e., a response to a user's action happens virtually immediately.

Typical representatives of ROIA are multiplayer online computer games, advanced simulation-based e-learning, and serious gaming. Due to a large, variable number of users at runtime, with intensive and dynamic interactions, ROIA make high QoS demands on the underlying network. Furthermore, these demands may change depending on the number of users and the actual application state: e.g., in a shooter game, a high packet loss in a combat state may have fatal consequences on the quality of the game, whereas it is less relevant when exploring the terrain.

Practically all state-of-the-art ROIA use the network on a best-effort basis, because of the lack of control over QoS in traditional networks. This leads to a suboptimal QoS perceived by the end-user, also known as *Quality of Experience (QoE)*. The existing best-effort techniques of controlling the QoS like the reservation of network bandwidth with the Resource Reservation Protocol (RSVP) or DiffServ [1] are mainly static or need to be configured by the network administrator and thus do not fit the dynamically changing demands of ROIA.

This paper addresses the dynamic network demand of ROIA by using the *Software-Defined Networking (SDN)* technology. We aim at enabling ROIA to manage the SDN infrastructure at runtime according to application requirements, thus leading to a higher and more predictable QoE for the end-user.

Figure 1 shows a schematic representation of the general SDN architecture and its interfaces. The *control layer* and its controller separate the application from the network layer. Instead of the complex configuration done by the administrator in traditional networks, in the SDN architecture, an application can perform changes in the network in real time, which should be especially advantageous for ROIA. For this purpose, applications communicate with the SDN Controller by means of a so-called *Northbound API*, whereas a *Southbound API* is used for communication between the SDN Controller and the network switches. While OpenFlow is a de-facto standard for the Southbound API, there is no standard Northbound API between the control layer and the application layer.

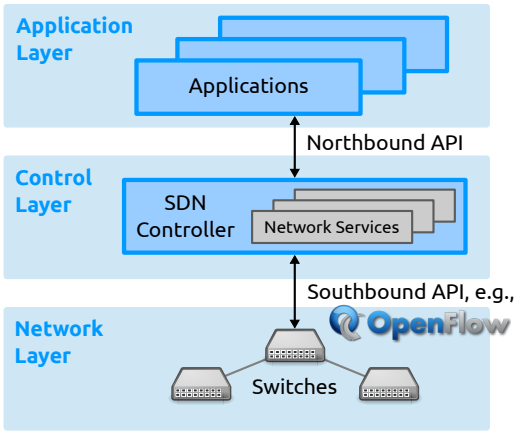

Fig. 1. General schema of SDN, adapted from [2].

The rest of this paper is organized as follows. In Section II, we describe ROIA properties and use them to specify the desired Northbound API functionality. Based on this API specification, the initial design and implementation of the SDN Module implementing the API functionality is presented in Section III. In Section IV, we demonstrate how the SDN Module is used by the ROIA developer. Finally, Section V concludes the paper with an experimental evaluation of the SDN Module.

II. Specification of Northbound API for ROIA

Typically, a ROIA simulates a virtual environment which is conceptually separated into a static and dynamic part. The static part covers, e.g., landscape, buildings and other non-changeable objects. The dynamic part covers *entities* like avatars, non-playing characters (NPC) controlled by the

Manuscript received December 14, 2013; revised January 13, 2014. The authors are with the Institute of Computer Science, University of Münster, 48149 Münster, Germany.
Email: humernbrum@wwu.de, glinkaf@wwu.de, gorlatch@wwu.de

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

computer, items that can be collected by clients, sideboard entries or, generally, objects that can change their state. Therefore, a continuous information exchange about the state of these objects is required between servers and clients.

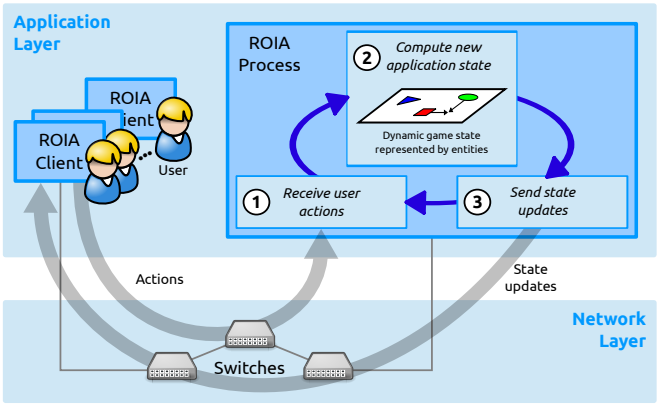

Fig. 2. One iteration of the ROIA real-time loop.

Figure 2 shows the structure of a ROIA based on the client-server architecture. The figure depicts only one *ROIA Process* which serves the connected *ROIA Clients*, but the typical scenario includes a group of ROIA Processes that are distributed among several server machines. In a continuously progressing ROIA, the application state is repeatedly updated in real time in an endless loop, called *real-time loop* [3], [4]. A loop iteration consists of three major steps. At first, the clients process the users' inputs; they are then transmitted in form of actions via the network and are received by the ROIA Process (step ① in Figure 2). The process then calculates a new application state by applying the received user actions and the application logic to the current application state (step ②). As the result of this calculation, the states of several dynamic entities may change. The final step ③ transfers the new, updated application state to the clients.

When designing the real-time loop for a particular ROIA, the application developer deals with several tasks regarding the network [5]. In steps ① and ③, the developer has to organize the network transfer of the data structures that realize user actions and entities. If the application is distributed among multiple machines, then step ② also requires the developer to organize the distributed computation of the application state and necessary communications for updating the state across different processes. Hence, the communication between a ROIA Process and the ROIA Clients or other processes comprises several *data flows* with different demands on network QoS. Different data flows could interfere with each other and, therefore, have to be distinguished in order to assign them different levels of QoS.

Implementing the network communication in a ROIA which makes particular QoS requirements is a challenging task because: a) the developer often has no detailed technical knowledge of networking and the involved protocols, b) the possibilities for specifying QoS requirements in traditional networks are limited, and c) the runtime controlling of the network layer is complex and often requires the intervention of the network administrator. These limitations stand in contrast to the dynamic QoS demands of ROIA. As a result, most ROIA use the network on a best-effort basis and rely on the over-provisioning of the network, which is not cost-efficient since the capacity reserved for peaks in network utilization remains unused most of the time. The use of SDN for ROIA should allow for an effective utilisation of network capacity and, at the same time, simplify the specification of network requirements respecting the dynamic QoS demands of ROIA.

To specify an SDN Northbound API for ROIA, we have extensively analysed ROIA network requirements and how they can be expressed in an API using various metrics, e.g., latency, packet loss, bandwidth, and jitter. Furthermore, technical constraints and expectations from the ROIA developer perspective have been taken into account.

The result of our analysis is a list of desirable features which should be covered by the envisaged API:

1. Since application requirements on network QoS are very dynamic, the API can either update them frequently or specify them in a flexible way.

2. The API allows for specifying network requirements during runtime ahead of time, if the application can anticipate such information in advance.

3. The API can specify different network requirements for different data flow types (e.g., frequent but small packets vs. infrequent but large amount of data, dynamic vs. static data).

4. The data flow types are specified using the API at development time, while the network requirements for a particular data flow type are specified at runtime.

5. The API enables the developer to specify network QoS requirements depending on the direction of data flows.

6. Besides low-level network metrics, the API accepts application-level metrics, e.g., response time, event count and number of synchronized entities, and transparently translates them into metrics understood by the SDN Controller.

7. The API supports multiple data flows types per ROIA and provides an aggregation mechanism for data flows with common requirements.

8. The API can handle the QoS of different data flow types individually.

9. The API allows bandwidth reservations to schedule state migrations and to support short-lived requests/releases of additional bandwidth for particular migrations.

10. The API allows for prioritizing data flows, in particular state synchronization over state migration.

11. The API can specify timing-based requirements on the network, e.g., a certain time frame for transferring data.

III. The API Initial Design

In order to meet the network requirements specified by the ROIA developer, the API implementation attempts to adapt the actual network utilisation of the application, e.g., by prioritizing packets of a particular data flow or by *state migration* which involves switching client connections between servers.

Figure 3 shows the basic architecture design of our Northbound API which includes the following 8 main components:

ROIA Process: the application process which provides (parts of) a ROIA to the connected users. A ROIA process implements the application logic, manages application data and sends application state updates to the connected clients.

Server: a hardware server or virtual machine (VM), able to run a ROIA process. As a ROIA may be distributed across multiple servers for scalability reasons, there are usually

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

multiple ROIA processes for a single instance of a ROIA (for simplicity, only one is shown in Figure 3).

ROIA Client(s): a client connected to one of the ROIA processes. This connection may switch to another process if, e.g., the client accesses entities processed by the other process, causing a state migration of the client's entities.

Network: comprises SDN-enabled switches that are configured by the SDN Controller.

SDN Controller: receives network QoS requests in form of QoS policies from the application via the API and attempts to configure the network resources accordingly.

Real-Time Framework (RTF): is a C++ library which provides programming abstractions and runtime support for ROIA. RTF [6] has been developed at the University of Münster, starting with the European edutain@grid project [7]. While ROIA development is significantly simplified by RTF, network aspects were still managed on a best-effort basis in RTF, which we now improve by employing SDN.

SDN Module: implements the Northbound API; it comes in form of a library which is linked into the ROIA and is integrated with RTF which allows it to manage RTF's network connections in order to meet the requested network QoS. The SDN Module also translates application-level metrics into metrics understood by the SDN Controller.

SLA Manager: implements business models and commitments between ROIA users, providers, and network operators (not shown in the figure). It has an impact on the control decisions of the SDN Controller and monitors achieved QoS in order to trigger business-level actions if necessary. In this paper, we rather focus on the technical aspects of QoS handling and network control between the SDN Controller and the ROIA.

The Northbound API mediates within the generic SDN architecture between the application and the SDN Controller. As stated in API feature 6, the SDN Module translates high-level, application-level metrics into low-level network metrics understood by the SDN Controller. This difference of perspectives is sometimes called *the application-network divide*.

In order to reflect the two different perspectives on the Northbound API – developer vs. controller – our Northbound API is designed as consisting of two parts:

- The *application-level API* (① in Figure 3) between the application code and the SDN Module. It enables the developer to specify how ROIA reports to the SDN Controller about network requirements and achieved QoS. This API takes into account that the communication links are dynamic at runtime, e.g., they can migrate between servers, and it allows for aggregating communication links. The ROIA developer uses this API to formulate his network-related requirements to the SDN Controller.
- The *network-level API* (② in Figure 3) between the SDN Controller and the SDN Module is used by the controller for receiving network requirements from the application and for application management, e.g., rejecting requirements which cannot be accommodated. It is also used to monitor QoS parameters at runtime.

Both APIs provide together a set of metrics to the ROIA developer and SDN Controller which are the basis for managing network resources in ROIA. In this paper, the two APIs are considered together as one Northbound API as long as the distinction between them is not relevant.

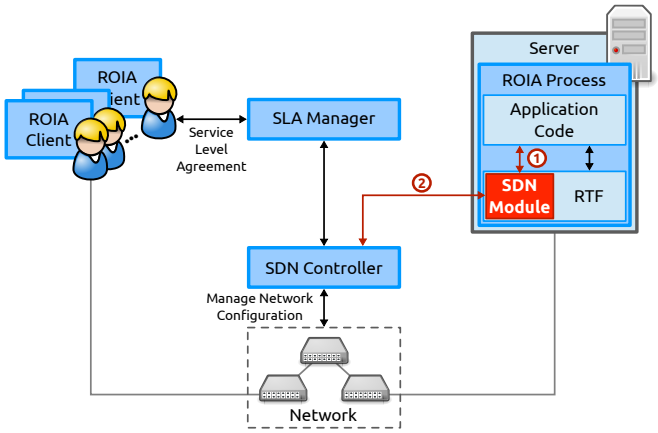

Fig. 3. Basic Architecture: a ROIA process serves connected clients, requests network QoS from the SDN Controller, and reports QoS information to the SDN Controller.

IV. Using Northbound SDN API for ROIA

In our API, a QoS requirement is expressed as a so-called *QoS policy* which is composed of one or several *QoS parameters*. A QoS parameter associates a network metric with a value to be complied with. In Figure 4, the example QoS policy prescribes that not more than 5 % of the data packets sent from a ROIA process to a ROIA client are lost and that a minimum throughput of 2 Mbit/s is achieved. The metrics in a QoS policy must be measurable and influenceable by the SDN Controller. The SDN Module currently provides the following network metrics that meet these requirements: latency in milliseconds (ms), throughput in Bit per second (Bit/s), packet loss in %, and jitter (variance of the transfer time) in ms.

All QoS parameters of a QoS policy apply to one or several *flows*. A flow consists of all data packets from a sender to the same receiver that are allocated to the same logical data flow. This allocation is defined by the application using flow labels. For example, real-time data can be identified with the flow label "1", assets with the label "2", etc. Thus, several data flows can be transmitted via the same communication channel and still be treated as different flows by the network.

Figure 4 shows a practical use case of how the specification of network requirements and their accommodation proceeds. In order to accommodate a QoS policy, the ROIA process sends it to the SDN Controller using methods provided by the SDN Module (step ① in Figure 4). The SDN Controller tries to fulfill the requirements of the QoS policy by adapting the network (step ②), e.g., the controller decides to transmit the data flow between the process and the client through another, faster connection. If the controller is unable to accommodate the desired requirements, it sends a reject message back to the ROIA process (step ③), with those parameters of the QoS policy that could not be fulfilled. Thus, the controller does not have to comply with the complete QoS policy, but may reject some of the QoS parameters, provided that it informs the process about it. This either happens as a direct reaction when receiving a QoS

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

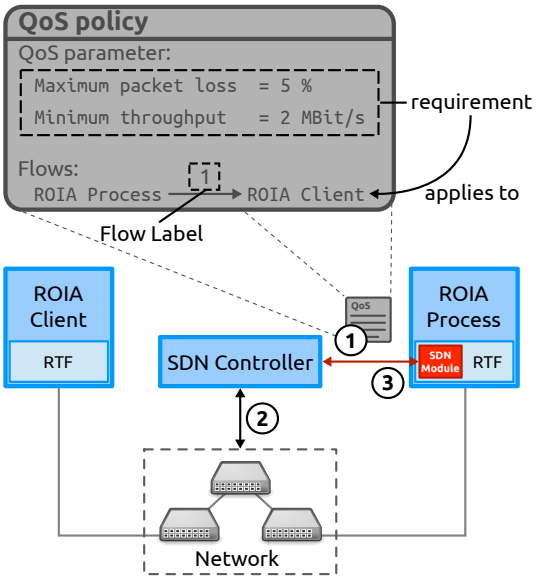

Fig. 4. Specification of network requirements for the communication between a ROIA Process and a ROIA Client.

policy, or at a later moment if the requirements were fulfilled at the beginning and then cannot be fulfilled anymore.

We design the structure of the SDN Module as comprising three components which are used for specification, administration, and communication, correspondingly. Figure 5 illustrates a typical workflow involving these components. For illustration purposes, we consider the previously described scenario of requesting QoS parameters for a particular flow. The ROIA developer uses the data structures of the specification component to define a QoS policy which is passed to the administration component. The administration component packs the QoS policy into a suitable message which is passed to the communication component that serializes the message and transmits it to the SDN Controller.

In the following, we explain the work of the three components of the SDN Module in more detail:

1. The *specification component* offers data structures and functions which are used by the ROIA developer to formulate network requirements. A QoS policy comprises several QoS parameters which apply to specified flows. QoS parameters may have a timeout, after which the SDN Controller does not have to monitor and accommodate the corresponding requirements any more. A QoS policy may contain each parameter type only once. All QoS parameters of a policy are applied to every flow specified in the QoS policy. Therefore, if different requirements have to be met for various flows, this has to be expressed by several QoS policies. In the SDN Module, a flow is uniquely defined by the sender's and receiver's IP address and port, as well as an optional flow label.

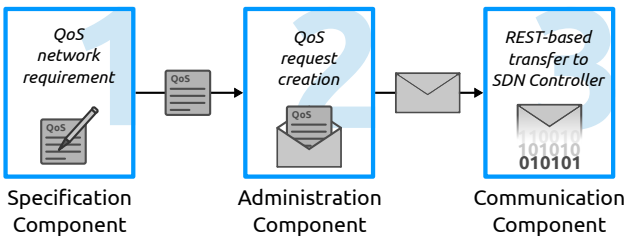

Fig. 5. A typical workflow between the components of the SDN Module.

2. The *administration component* contains the key functions of the SDN Module. Using these functions, the ROIA developer can transmit QoS policies to the SDN Controller or cancel requirements of QoS policies that have already been transmitted to the controller. The administration component, transparently for the user, packs the QoS policies into messages and passes them to the communication component, while bookkeeping is made for requested, rejected, granted and active QoS policies. This way, the ROIA developer can inquire the SDN Module for the current status of QoS policies without having to contact the SDN Controller (which would take a comparatively long time). Also, the administration component is connected to RTF that provides the ROIA monitoring statistics on the number of events, clients, state synchronizations, etc. By using these statistics, the administration component can translate application-level metrics into network-level metrics before passing QoS policies to the communication component.

3. The *communication component* coordinates the connection and communication between the application and the SDN Controller. The ROIA developer never works with this component; it performs its tasks in the background, transparently for the user. One of these tasks consists in serializing and sending messages to the SDN Controller, without blocking the SDN Module. We use a REST-based API implementation [8] to separate the communication from the specific implementation language of the ROIA API and the controller side (e.g., C++ or Java). The fact that REST can use HTTP as transport protocol makes it easy to implement and adapt if some changes in the specification of the API are needed.

V. Tests and Evaluation

In order to evaluate our implementation of the Northbound API by the SDN Module, we conduct several tests which show how the network requirements specified by the ROIA developer using the SDN module are monitored and accommodated by the SDN Controller.

Figure 6 shows our test network topology built using the *Mininet* simulation system [9]. With Mininet, even complex networks with thousands of hosts and switches can be tested without having to assemble a physical network. Our virtual topology consists of six hosts named h1 to h6 and three software switches on the basis of Open vSwitch: s1, s2 and s3. The hosts h1 to h5 are connected to switch s1, whereas h6 is connected to s2. The throughput of the connection between s1 and s2 is set to be limited to 10 Mbit/s, while the throughput of the connections between s1 and s3, as well as between s2 and s3, amounts to maximum 20 Mbit/s.

The virtual network is controlled by our prototype implementation of the SDN Controller which monitors the network utilization and attempts to adapt the network in order to accommodate specified QoS requirements. The controller initially configures the network, such that the switches forward data packets on the shortest path to destination, i.e., packets sent from hosts h1 ... h5 to h6 are forwarded via the direct connection between s1 and s2, which is shown in Figure 6.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

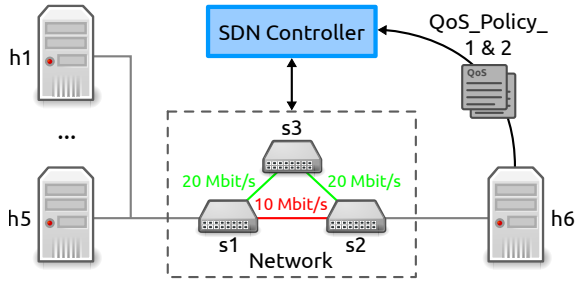

Fig. 6. Test network topology for the evaluation of the SDN Module.

Our test scenario refers to the use case shown in Figure 4. The host h6 can be considered as a virtual machine running a ROIA process which is accessed by ROIA clients running on h1 to h5. During the test, h6 issues two QoS policies for its communication with h1 and h2, named QoS_Policy_1 and QoS_Policy_2, respectively. Both policies specify a minimum throughput of 5 Mbit/s for all data sent to h6 by the corresponding host. In order to send the data and measure the actual throughput, we use the *Iperf* tool [10] which continuously sends randomly created data to a given receiver and calculates the achieved throughput. Our test is divided into eight *measurement intervals*, each of 50 seconds, where the actual throughput is recorded five times every ten seconds.

At the beginning of interval 1, the QoS_Policy_1 is issued, and h1 starts sending data to h6. At the beginning of each next interval, the hosts h2 to h5, in turn, also start sending data to h6, such that, from interval 5 on, all five host are sending data to h6 simultaneously. This increasing load is expected to reduce the maximum throughput available to each host. The goal of this test is to show that the controller reacts to this situation and attempts to fulfill the requirements of QoS_Policy_1, e.g., by redirecting packets from h1 to h6 via s3. We also test what happens if the controller cannot fulfill the specified requirements. For this purpose, after the 5th interval, additional QoS_Policy_2 is issued to the controller. At the beginning of interval 8, the connection between switches s1 and s3 is externally limited to 10 Mbit/s to simulate a higher utilization of this route, which is expected to lead to the rejection of QoS_Policy_2 by the controller.

Figure 7 depicts the measurement results for the described test scenario. The bars represent the average throughput per host in a measurement interval of 50 sec. In addition to the bars for the average values, the curve in Figure 7 illustrates the single measured values for the throughput of h1.

The results of intervals 1 to 5 show that the throughput between h1 and h6 gradually decreases as expected when the number of senders increases, because the connection between s1 and s2 is limited to 10 Mbit/s. All hosts share this available throughput, such that when h4 starts sending data to h6 in interval 4, the throughput of h1 falls below 5 Mbit/s, which is a violation of QoS_Policy_1. After monitoring this, the controller changes the route of the packets which are sent from h1 to h6: it adds new rules to the flow tables of the switches s1, s2 and s3 to redirect the packets via s3. Subsequently, the measured values show an abrupt increase of the throughput of h1 to over 19 Mbit/s. The remaining hosts continue to share the available throughput of the connection between s1 and s2. Due to this adaptation of the network through the controller, the cumulative throughput increases to almost 30 Mbit/s starting from interval 4.

In interval 5, we observe that the measured average throughput between h2 and h6 is 2,15 Mbit/s, i.e., below the specified minimum requirement of QoS_Policy_2 issued at interval 6. In order to fulfill the requirement of QoS_Policy_2, the controller redirects all packets sent by h2 to h6 via switch s3. Therefore, starting from interval 6, both h1 and h2 send data to h6 via s3, i.e., they share the available throughput of this route which is limited to 20 Mbit/s. We observe that the measured throughput between h2 and h6 increases to 6,67 Mbit/s, i.e., the requirements of both QoS policies become fulfilled. At the beginning of interval 8, the results show that the throughput of h1 falls to 4,51 Mbit/s which is caused by the external limitation of the connection between s1 and s3. The controller cannot fulfill both QoS policies simultaneously anymore. Therefore, the controller rejects QoS_Policy_2 and takes back the adaptation made earlier on the network. Thus, packets from h2 to h6 are sent again via the original connection between switches s1 and s2, and QoS_Policy_1 is again fulfilled.

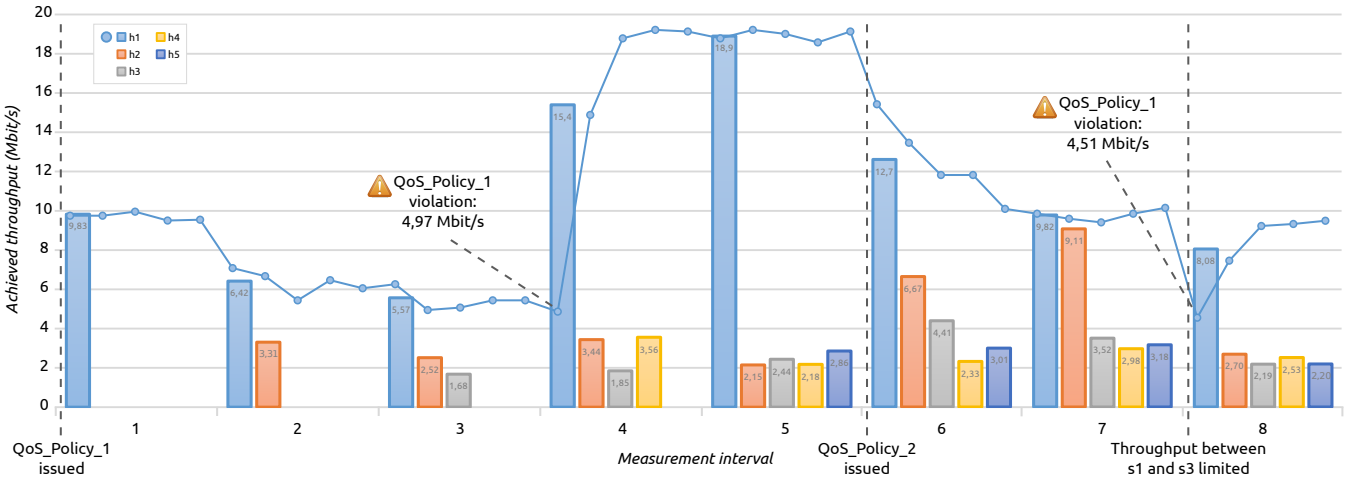

Fig. 7. Results of the functional test of the SDN Module.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

VI. Conclusion and Related Work

This work is motivated by challenging ROIA applications which make dynamic demands on the network, while the state-of-the-art possibilities of influencing the network QoS are mostly static.

Our contribution is a Northbound API for SDN networks which allows the ROIA developers to specify their requirements on the network and communicates corresponding requests to the network controller. This offers a new approach for addressing the dynamic QoS demands of ROIA. We designed and implemented the SDN Controller and the SDN Module that cooperate on monitoring and accommodating QoS by adapting an OpenFlow-enabled network.

Within the SDN community, there have been recent activities towards creating and standardizing a Northbound API. Early implementations like Floodlight's REST-based Northbound API [11] and the Nicira Network Virtualization (NVP) Platform API [12] handle basic functionalities, such as discovering the network topology, static rule programming, and running virtual networks on top of a network infrastructure. Most of these approaches are relatively new and yet few applications use them. Moreover, they still require a detailed knowledge of networking details from the application developer and provide no means for specifying QoS requirements beyond traditional, static techniques like resource reservation with RSVP. Our focus in this paper is on designing an SDN Northbound API for controlling QoS in ROIA with respect to their dynamic network demands. Additionally, the envisaged API provides a higher abstraction of QoS by taking ROIA-specific metrics into account and, therefore, liberates the application developer from specifying low-level network metrics.

By using the SDN Module, the application developer can exploit the advantages of SDN without having detailed knowledge of network infrastructure and protocols and, therefore, can focus on the application design. The transparent translation of application-level metrics into network metrics by the SDN Module provides a more convenient QoS mechanism for the developer than the current SDN approaches. For instance, SDN controllers like Floodlight or NOX [13] provide a Northbound API for manipulating the forwarding behaviour of the network, but addressing dynamic or reactive QoS demands requires direct programming of the corresponding controller. Other approaches like the Nicira NVP API focus on the virtualization or slicing of networks. From the perspective of ROIA, these virtualized networks behave like traditional networks which provide only static techniques for QoS or cause a significant administrative overhead.

The first evaluation of our Northbound API shows that the requirements on the network specified using the SDN Module can be monitored and accommodated by the SDN Controller, leading to a higher and more predictable QoS for ROIA.

Acknowledgements

The authors would like to thank Folker Schamel and Michael Franke from Spinor and Eduard Escalona and Iris Bueno from i2Cat for valuable discussions and sharing their expertise on the design of ROIA and networks. Our research has received funding from the ECs 7th Framework Programme under grant agreements 318665 (OFERTIE) and 295222 (MONICA).

References

[1] S. Vegesna, *IP Quality of Service*. Cisco Press, 2001.
[2] O. N. Foundation, "Software-Defined Networking: The New Norm for Networks," 2012.
[3] M. Joselli, M. Zamith, E. Clua, R. Leal-Toledo, A. Montenegro, L. Valente, B. Feijo, and P. Pagliosa, "An Architeture with Automatic Load Balancing for Real-Time Simulation and Visualization Systems," *Journal of Computational Interdisciplinary Sciences*, vol. 1, no. 3, pp. 207–224, 2010.
[4] L. Valente, A. Conci, and B. Feijó, "Real Time Game Loop Models for Single-Player Computer Games," *Proc. IV Brazilian Symp. on Computer Games and Digital Entertainment*, 2005.
[5] F. Glinka, A. Ploss, S. Gorlatch, and J. Mller-Iden, "High-Level Development of Multiserver Online Games," *International Journal of Computer Games Technology*, 2008.
[6] A. Kumar, J. Etheredge, and A. Boudreaux, *Algorithmic and Architectural Gaming Design: Implementation and Development*. IGI Global, 2012.
[7] J. Ferris, M. Surridge, E. Watkins, T. Fahringer, R. Prodan, F. Glinka, S. Gorlatch, C. Anthes, A. Arragon, C. Rawlings, and A. Lipaj, "Edutain@grid: A business grid infrastructure for real-time on-line interactive applications," in *Grid Economics and Business Models*, ser. Lecture Notes in Computer Science, J. Altmann, D. Neumann, and T. Fahringer, Eds. Springer Berlin Heidelberg, 2008, vol. 5206, pp. 152–162.
[8] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.
[9] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.
[10] "Iperf. `http://iperf.sourceforge.net`," 2013.
[11] "Floodlight OpenFlow Controller. `http://www.projectfloodlight.org/floodlight`," 2013.
[12] Nicira, "Network Virtualization Platform (NVP) White Paper," 2013.
[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, pp. 105–110, 2008.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)