A Northbound API for QoS Management in Real-Time Interactive Applications on Software-Defined Networks

Tim Humernbrum, Frank Glinka, and Sergei Gorlatch
University of Muenster, 48149 Muenster, Germany
Email: {t.hume, glinkaf, gorlatch}@uni-muenster.de

Abstract —Real-Time Online Interactive Applications (ROIA), e.g., multiplayer online games and simulation-based e-learning, make high Quality of Service (QoS) demands on the underlying network. These demands depend on the number of users and the actual application state and, therefore, vary at runtime. Traditional networks have very limited possibilities of influencing the network behavior to meet the dynamic QoS demands, such that most ROIA use the underlying network on a best-effort basis. The emerging Software-Defined Networking (SDN) technology decouples the control and forwarding logic from the network infrastructure, making the network behavior programmable for applications. This paper analyses ROIA requirements on the underlying network and describes the specification and design of an SDN Northbound API that allows ROIA applications to specify their dynamic network requirements and to meet them at runtime using SDN networks.

Index Terms—Quality of service, network management, software-defined networking, soft real-time applications, real-time framework (RTF)

I. Introduction: ROIA and SDN

Real-Time Online Interactive Applications (ROIA) 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, simulation-based e-learning, and serious gaming. Due to a large, variable number of users, with intensive and dynamic interactions, ROIA make high *Quality of Service (QoS)* demands on the underlying network. Furthermore, these demands may continuously 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 QoS, whereas it is less relevant when a player is 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 traditional techniques of controlling the QoS like the reservation of network bandwidth with the Resource Reservation Protocol (RSVP) or DiffServ [1] are mainly static and thus do not fit the dynamically changing demands of ROIA.

In this paper, we discuss the use of the emerging *Software-Defined Networking (SDN)* technology [2] to address the dynamic network demands of ROIA: SDN enables applications to manage the network behavior at runtime, thus leading to a higher and more predictable QoE for the end-user. For this purpose, the control logic in SDN is decoupled from the network infrastructure and configured by a centralized *SDN controller* which has a global view of the network.

An open, actively studied problem for SDN is the design of the so-called *Northbound API* which should precisely define how an application communicates with the SDN controller. We focus in this paper on developing an SDN Northbound API for the emerging ROIA internet applications. For the so-called Southbound API that connects the SDN controller with the network infrastructure, standardized solutions like OpenFlow exist; this API is not considered in the paper.

In the following, we describe the common structure of ROIA applications and start to systematically develop our specification of a Northbound API for ROIA by analyzing the most important ROIA scenarios regarding network requirements (Section II). In Section III, we list the desired API functionality, followed by an experimental study about the relation between application-level and network-level QoS metrics in Section IV. Section V presents our initial design of the SDN Module which implements the specified functionality of the Northbound API for ROIA applications. Section VI describes how the SDN Module is used by the ROIA developer, followed by an evaluation of the new SDN Northbound API and its implementation in Section VII.

II. ROIA Scenarios for Network QoS

A typical ROIA application is conceptually separated into a static and dynamic part. The static part includes, e.g., landscape, buildings and other non-changeable objects. The dynamic part includes entities like avatars, non-playing characters (NPC) controlled by the computer, items that can be collected by players or, generally, objects that can change their state. A continuous

Manuscript received March 15, 2014; revised June 12, 2014.
This work was supported by the EC's 7th Framework Programme under grant agreements 318665 (OFERTIE) and 295222 (MONICA).
Corresponding author email: t.hume@uni-muenster.de.
doi:10.12720/jcm.9.8.607-615

information exchange about the state of dynamic objects is required between servers and clients as the application is running.

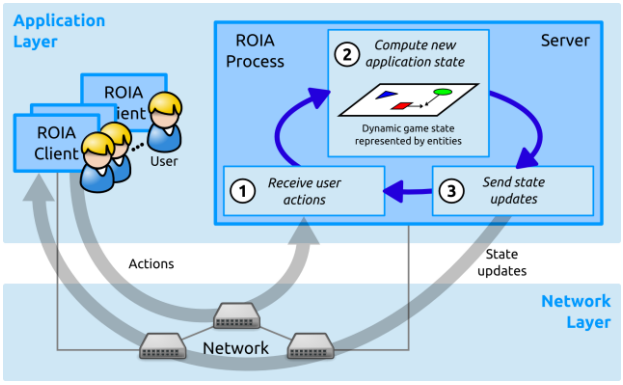

Fig. 1. Structure of a ROIA and its real-time loop.

Fig. 1 shows the structure of a ROIA; it 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 infinite loop, called real-time loop [3]. A loop iteration consists of three major steps as follows. At first, the clients process the users' inputs which are then transmitted in form of actions via the network and received by the ROIA Process (step ① in Fig. 1). The process 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 of the loop 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 [4]. In steps ① and ③, the developer has to organize the network transfer of the data structures that realize user actions and state updates. If the application is distributed among multiple machines, then in step ② the developer has to manage the distributed computation of the application state and the necessary communications for updating the state across different processes. Moreover, 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 may interfere with each other and, therefore, have to be distinguished to assign different levels of QoS to them.

Implementing the network communication in a ROIA with particular QoS requirements is a challenging task because: a) the developer often has no detailed technical knowledge of the involved networks and their 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 still 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. Our goal is to use SDN for an effective utilization of network capacity and, at the same time, simplify the specification of network-related QoS demands of ROIA.

In the following, we analyze three typical application scenarios that demonstrate how the network demands of ROIA depend on the actual application state and the number of users.

First-Person Shooter (FPS) is a computer game scenario where the players move in a 3D game world, see the world from their own perspective, and fight against enemy players. Games like Battlefield 3 require an especially high update rate for a fluent game flow, which is achieved, e.g., by using the UDP protocol. When using UDP, packets may get lost or arrive in the wrong order. The higher the loss rate, the poorer the extrapolation of the future enemy positions; it may eventually become impossible for a player to target his enemies, i.e., the game becomes unplayable [5]. This negative effect becomes less relevant if a player has no contact to other players, e.g., when exploring the terrain. Therefore, the FPS's QoS demand depends on the player's position and the degree of interactivity with other players and can change quickly during runtime.

Real-time strategy (RTS) games like StarCraft or Command & Conquer usually start with a phase where players build a basis, after which they enter combat. In this process, the game's network requirements change dynamically, because numerous interactions of the units engaged in combat require considerably more bandwidth than when building a basis.

In *e-learning applications*, participants usually follow the lecture of a tutor by video stream, watch a digital blackboard, and chat with other students or with the tutor. Compared to games, the data amount per client in E-learning is much higher, especially when streaming video and audio data [6]. For that, a distribution among several resources can help, but it requires a suitable network configuration and eventually may congest certain parts of the network. For a higher number of participants, it may become impossible for the servers to serve all participants, i.e., additional connections must be refused. Scalability is achieved either by replicating the application across multiple network sites, e.g., located closer to the clients, or by using multicast [7].

The three scenarios above show that during runtime, the network requirements of ROIA may change, depending on the environment and the situation of the user. The state-of-the-art possibilities to fulfil the network requirements are mostly static, involve high administrative costs and, therefore, stand in contrast to the dynamic QoS demands of ROIA. It is possible to either reserve bandwidth along a specific route in the network, e.g., using the Resource Reservation Protocol

(RSVP) [1] which requires a time-consuming configuration of the corresponding route components, or to priorities data packets for pre-defined traffic classes. While the reservation is complex and rather static, the second approach lacks an adequate support from the network hardware.

III. SDN Northbound API Specification

Fig. 2 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 [8] is a de-facto standard for the Southbound API, there is yet no standard Northbound API between the control layer and the application layer.

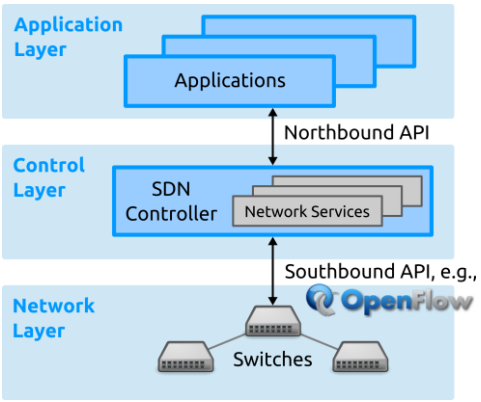

Fig. 2. General schema of SDN, adapted from [2].

In the following, we analyze how the network QoS demands of ROIA can be expressed using an SDN Northbound API, in the following called API. The result of our analysis is a list of desired features of the envisioned API, written as **API Feature X:…**. We discuss metrics specified by the API and different data flow types for the transmission via the network, together with the typical requirements on them. Moreover, technical constraints and expectations from the ROIA developer's perspective are discussed.

The three ROIA networking scenarios described above demonstrate that their network requirements may change depending on the application state. Therefore, the first desired feature of the Northbound API is as follows.

API Feature 1: The API should enable the application developer to update the application requirements on network QoS frequently and/or specify them in a flexible way.

While the application requirements may change quickly, e.g., within seconds, the network may not be able to adjust its resource allocation and packet forwarding rules within this timescale. Therefore, the API should allow the application to compensate for the slow adaption of the network, e.g., by specifying future requirements in advance based on application-specific knowledge and providing them to the SDN controller.

API Feature 2: The API should enable the developer to specify network requirements in advance if the application is able to anticipate such information.

The application data flows, for which the QoS requirements are specified, typically fall into classes defined during development time. Typical examples are: *state synchronization*, *state migration*, and *asset transfers*. In ROIA, state synchronization is used to update the application state from one server or client to all other ROIA participants (clients and servers). State migration is used in multi-server ROIA for scaling them to high user numbers. For example, in an online multi-player game employing multiple servers, each managing a dedicated zone of the virtual world, if an entity moves from one zone into another, then the complete state of the entity must be migrated from the server responsible for the original zone to the server of the new zone [9]. While state synchronization omits non-changed data from the transmission, state migration requires a full serialization and transmission of the migrated entity.

Different data flows generate different network requirements: while state synchronization is very timing-sensitive and usually cannot be rescheduled, the start of state migration can be delayed, but it must be completed as soon as possible after it has started. Asset transfers like texture or sound downloads have more relaxed timing characteristics and can be flexibly scheduled to the optimal point in time, e.g., when the network has free capacity.

API Feature 3: The API should enable the developer to specify different network requirements for different data flows (e.g., state synchronization vs. asset transfers).

In ROIA, the contents of transfers depend on the flow direction: transfers from server to client include object positions, property changes, state data and entity creation/destruction; transfers from client to server include input data (e.g., desired movements of characters) and client-side predicted physics. Modern ROIA use many kinds of data flows during state synchronization. For example, the Shark 3D game engine [10] has about 20 standard internal software components, each one transferring a different kind of synchronization data.

API Feature 4: The API should enable specifying network QoS requirements depending on the direction of data flows (client-to-server and server-to-client).

While servers and clients regularly send state synchronization packets, the amount and size of packets are unknown and can hardly be predicted. Moreover, ROIA developers often have no detailed technical knowledge of the involved networks and protocols and, therefore, tend to specify their requirements as "transfer packets as fast as possible", "need as much bandwidth as

possible" or "no jitter is desired" - which are not suitable for a pro-active management of the network capacity. Therefore, the API should provide application-level metrics which are understandable and convenient for the developer.

API Feature 5: The API should liberate the developer from specifying low-level network metrics, e.g., bandwidth, jitter and latency. Rather application-level metrics like response time should be offered and then automatically translated to low-level metrics understood by the SDN controller.

When sending state synchronization updates to multiple clients, a server sends different data for each client connection and entity. Correspondingly, the ROIA developer might tend to specify his network requirements on a per-client connection basis. However, certain network limitations, e.g., the maximum capacity of the switches' flow tables, may require a configuration based on aggregated flows: e.g., 10 state synchronization flows to users with the same bandwidth and latency requirements may be combined to a single aggregated flow that requires 10x the bandwidth with the same latency.

API Feature 6: While supporting multiple data flows with different network requirements, the API should also provide an aggregation mechanism for flows with common requirements.

In a ROIA, a state migration that has started should be completed very quickly, since entities being migrated cannot be updated. Therefore, suitable bandwidth must be available for the data transfer.

API Feature 7: The API should allow bandwidth reservations for scheduling migrations, or support requests/releases of additional bandwidth for particular migrations.

A priority handling of state migration vs. synchronization within limited bandwidth scenarios is desirable yet difficult. If synchronization receives a higher priority, then migrations may take a long time, thereby leading to non-updated entities. If the two have the same priority, a slightly lower but still good state synchronization QoS may be better than giving the migrated player a temporarily very poor QoS. Assigning state migration a higher priority than synchronization may lead to a very poor QoS for all players. Which approach is better cannot be decided technically, but rather depends on the current application state of a ROIA.

API Feature 8: The API should allow flexible prioritization of data flows, e.g., state synchronization over state migration.

The term *asset* is used within the gaming industry to refer to textures, geometries, animations, images, sound and videos which compose the content of a game. Assets belong to the static part of the game, i.e., no state synchronization or migration is required for them. However, clients must have local access to them, e.g., load textures and geometries for rendering a new zone that the client is entering. Assets are typically much larger than the program code, e.g., several GB for assets as compared to several MB for the code. A new approach is the on-demand download of assets concurrently to the game processing in the background. For this, the assets are placed in a repository, e.g., by the game developer. While the client starts the game on his computer with a minimal set of assets, e.g., required to render the starting zone for new players, the remaining assets are fetched in the background without disturbing the game processing. Furthermore, the client is able to reload new assets if they have been placed in the repository [11].

API Feature 9: The API should be able to place timing-based requirements on the network, e.g., transfer a particular amount of asset data within a certain timeframe.

IV. Experiments with QoS Metrics

In this section, we present an experimental study with a shooter-like online game about the relation between the low-level and the application-level QoS metrics as discussed in API Feature 5. The low-level metrics include the CPU load and bandwidth on the server. The major application-level metric used to measure the game's QoS is the response time: how long it takes until a movement command sent by a client to the server is applied by the server to the game state and the result is rendered on the client's display. Academic studies of commercially successful shooter games show that the player experience is greatly affected by even modest (100 – 150 ms) delay in response time. The response time is a more accurate metric than the plain network latency as it measures also the delays within the server-, client- and networking application code.

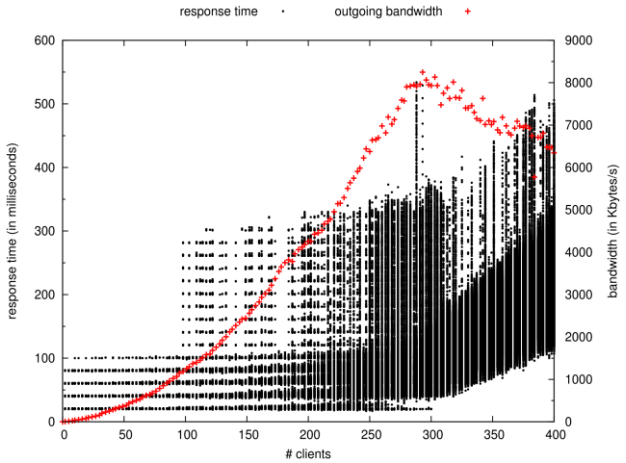

Fig. 3. Response time and bandwidth consumption of an example game.

Fig. 3 shows the response time (bars) observed on the clients and the bandwidth (curve) consumed by the server in relation to the number of participating clients. Each of the bar points represents the perceived response time of a client (altogether about 3 million measurement points). The observed response time is very good (below 100 ms) for <100 clients. From that number on, clients sometimes perceive higher response times to their inputs, up to 300 ms, until 200 clients are reached. For even larger number

of clients, the response time peaks increase to 550 ms which is an unacceptable value. However, the bandwidth consumed by the server shows a steady growth for up to 270 clients.

In our next experiment, Fig. 4 shows the time spent on the server for one iteration of the real-time loop (so-called tick duration), depicted by bar points, as compared to the CPU load observed on the server, depicted by the curve. We observe that the server is able to keep the targeted update rate of 25 updates per second, i.e. tick duration of 40 ms, for <270 players. For larger numbers of players, the clients issue too many actions which cannot be processed by the server quickly enough to keep the targeted update rate. This saturation of QoS is achieved at a CPU load of <90 %. Therefore, again, the low-level metric "CPU load" does not precisely express the QoS behavior.

From this analysis, it follows that there is no server-side reason for the increased response time on >100 clients as observed in Fig. 3; rather the network becomes congested by the increasing number of packets. This is a typical situation where an SDN Northbound API should allow the application to monitor application-level QoS metrics like response time and manage the programmable network accordingly.

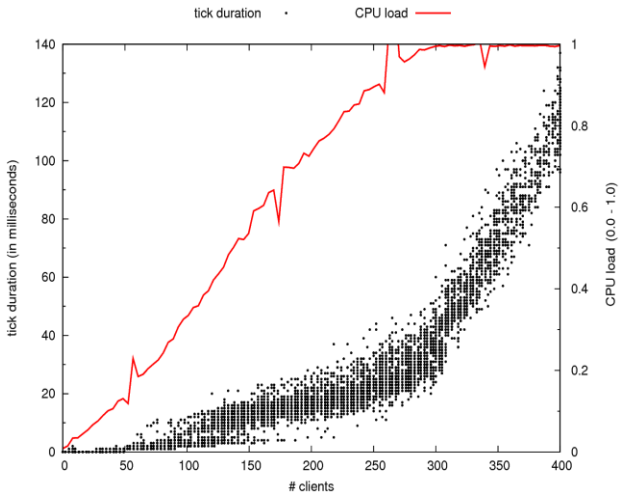

Fig. 4. Server-side processing time for one real-time loop iteration (tick duration) and average CPU load in an example game.

V. SDN Northbound API: Initial Design

The advantage of SDN is that the Northbound API implementation can employ the SDN controller to dynamically accommodate the QoS requirements specified by the ROIA developer. This is done by adapting the network, e.g., by prioritizing packets of a particular flow or by redirecting flows if certain routes in the network become congested. In contrast to the traditional networking scenario shown in Fig. 1 where ROIA has no possibility of influencing the network, Fig. 5 shows the SDN scenario for ROIA, with our Northbound API realized by the so-called SDN Module. The scenario includes the following eight components:

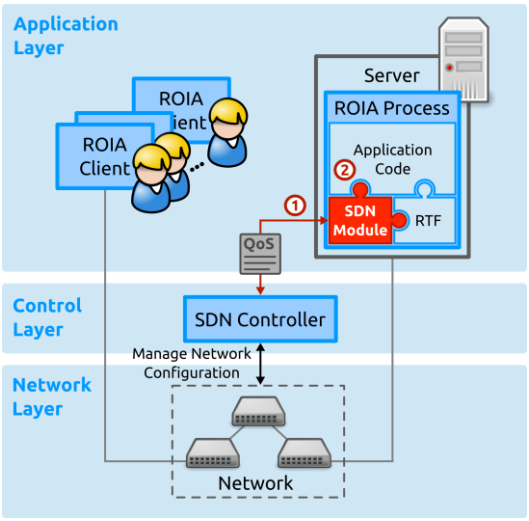

Fig. 5. Architecture of SDN networking for ROIA.

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, able to run a ROIA process. As a ROIA may be distributed across multiple servers for scalability reasons, there are usually multiple ROIA processes for a single application instance (for simplicity, only one is shown in Fig. 5).

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 migration of the client's entities.

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

QoS policy: a data structure that expresses the network requirements for specific data flows.

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): a C++ library for development and runtime support of ROIA. RTF [12] has been developed at the University of Münster, starting with the European *edutain@grid* project [13]; it offers access to high-level application metrics like response time, in-application event count, entity count, entity positions, etc., which are translated by the Northbound API into network-level metrics understood by the SDN controller.

SDN Module: implements the Northbound API; we implement it as a C++ library which is linked into the ROIA and is connected 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 (e.g., response time) into network-level metrics by evaluating application statistics provided by the RTF. We deliberately omit Northbound

API-related issues that are currently addressed by other research projects, e.g., access control, fairness, accountability, and resource scheduling issues.

We partition our Northbound API in two parts which reflect the different perspectives of ROIA developers and the SDN controller. While the controller can only affect network metrics that it is able to monitor, e.g., by evaluating the packet and byte counters of flow tables, ROIA developers often have no technical knowledge about networking details and, therefore, cannot map application's QoS demands to network-level metrics. The both parts of the SDN Northbound API are shown in Fig. 5 and cover:

1) A base API (① in Fig. 5) which offers generic network control functions to applications; we call it *network-level API*. This API connects the SDN controller and the SDN Module and is used by the controller for receiving network requirements from the application and for network management, e.g., rejecting requirements which cannot be accommodated.

2) An application-level API (② in Fig. 5) which targets the ROIA developer's point of view and hides low-level network details. It enables the application developer to specify how an application reports to the SDN controller about its network requirements and the achieved QoS.

VI. Using SDN Northbound API for ROIA

In our SDN Northbound 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 Fig. 6, the example QoS policy prescribes that <5 % of the data packets from a ROIA process to a client are lost and that >2 Mbit/s throughput is achieved. The metrics in a QoS policy must be measurable and influenceable by the SDN controller. The SDN Module currently supports the following network-level metrics: latency in milliseconds (ms), throughput in Bit per second (Bit/s), packet loss in %, and jitter in ms.

All QoS parameters of a QoS policy apply to one or several data 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.

Fig. 6 shows a practical use case of how the specified network requirements are accommodated. 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 Fig. 6). The SDN controller attempts 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 policy, or at a later moment if the requirements were fulfilled at the beginning and then cannot be fulfilled anymore.

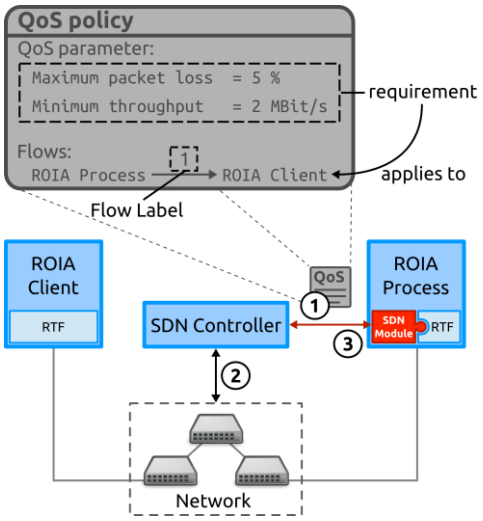

Fig. 6. Specification of network requirements for the communication between a ROIA Process and a ROIA Client.

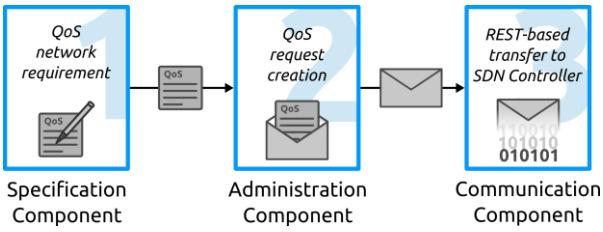

Fig. 7. Typical workflow between the components of the SDN Module.

We design the structure of the SDN Module as comprising three components which are used for specification, administration, and communication, correspondingly. Fig. 7 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:

- 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.

- The *administration component* implements 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 and are no longer needed, e.g., because of a changed application state. The administration component, transparently for the developer, 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. Therefore, the ROIA developer can inquire the SDN Module about 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 run-time ROIA monitoring statistics on the number of events, clients, state synchronizations, etc. By using these statistics, the administration component translates application-level metrics into network-level metrics before passing QoS policies to the communication component.
- 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 developer. 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 [14] 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.

VII. 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.

Fig. 8 shows our test network topology built using the Mininet simulation system [15]. 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.

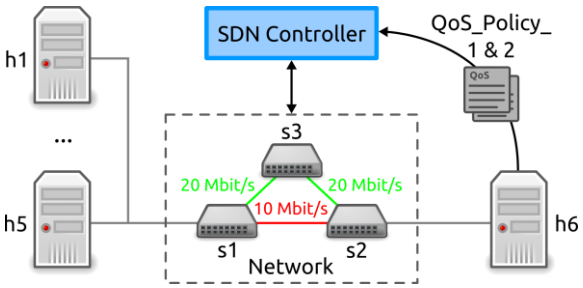

Fig. 8. Network topology for testing the SDN Module.

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, as shown in Fig. 8.

Our test scenario refers to the use case shown in Fig. 6. 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 [16] 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.

The test scenario is as follows. 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.

Fig. 9 depicts the measurement results for the described test scenario. The bars represent the average throughput per host in the measurement interval of 50 sec. In addition to the bars for the average values, the curve in Fig. 9 illustrates the single measured values for the throughput of h1.

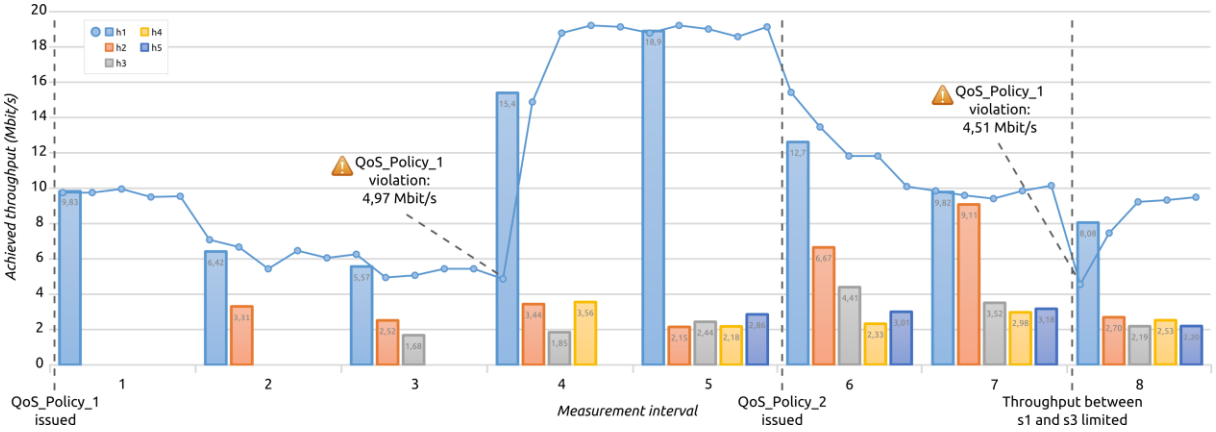

Fig. 9. Results of the functional test of the SDN Module.

The results in the 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 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.

VIII. Conclusion

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 focus is on a Northbound API for SDN networks: it allows ROIA applications to specify their requirements on the network and communicates corresponding requests to the SDN controller, which tries to accommodate them by reconfiguring the network. This offers a new approach for addressing the dynamic QoS demands of ROIA.

Within the SDN community, there have been recent activities towards creating and standardizing a Northbound API. So far, early implementations can be found that handle basic functionalities such as queries, state reporting or rule programming. Examples are Floodlight's REST-based Northbound API [17] and the Nicira Network Virtualization Platform (NVP) API [18]. They require a detailed technical knowledge of the network infrastructure and protocols. Participatory networking (PANE) [19] offers a Northbound API which delegates read and write authority from the network's administrators to the end users and applications which can reserve guaranteed minimum bandwidth, issue path control requests, or configure access control.

While the mentioned Northbound APIs provide applications with management functionality for the network, they fall short in two aspects: a) not all network metrics that are important for ROIA are supported; and b) their interfaces are rather low-level from the ROIA developer's point of view. Therefore, our goal is two-fold: 1) analyze and motivate additional networking metrics that should be supported by a network-level API, and 2) develop an additional layer of abstraction with the application-level API which supports the ROIA developer in configuring and monitoring the network.

Our specification (desired features) of the Northbound API is based on a detailed analysis of ROIA scenarios and their requirements on the underlying network. Our prototype of the SDN Module implements the specified API functionality and cooperates with an SDN controller on monitoring and accommodating QoS by adapting an SDN-enabled network. The SDN Module can be used together with libraries and frameworks which provide application statistics and access to their network connections, like RTF. The SDN Module can communicate via the network-level API with any controller which provides a suitable Northbound API for reporting network metrics, e.g., with the PANE controller [19].

ACKNOWLEDGMENT

We would like to thank our project partners, especially Folker Schamel and Michael Franke from Spinor GmbH (Munich) and Eduard Escalona and Iris Bueno from i2Cat Foundation (Barcelona) for valuable discussions and for sharing their expertise on the design of ROIA applications and SDN controllers.

REFERENCES

[1] S. Vegesna, *IP Quality of Service*, Cisco Press, 2001.
[2] O. N. Foundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.
[3] M. Joselli, M. Zamith *et al.*, "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] F. Glinka, A. Ploss *et al.*, "High-level development of multiserver online games," *International Journal of Computer Games Technology*, 2008.
[5] T. Beigbeder, R. Coughlan *et al.*, "The effects of loss and latency on user performance in unreal tournament 2003," in *Proc. ACM Network and System Support for Games Workshop*, 2004.
[6] T. Szigeti, "Quality of service network design considerations for cisco telepresence systems," *Technical Services Newsletter*, 2011.
[7] P. Zhang, W. Liu, *et al.*, "A study of video-on-demand learning system in e-learning platform," in *Proc. CSSE '08*, vol. 5, 2008, pp. 793–796.
[8] N. McKeown, T. Anderson, *et al.*, "Open Flow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[9] C. Ghosh, R. P. Wiegand, *et al.*, "An architecture supporting large scale MMOGs," in *Proc. 3rd International ICST Conference on Simulation Tools and Techniques*, Brussels, Belgium, Belgium: ICST, 2010, pp. 2:1–2:8.
[10] Spinor GmbH. (2013). Shark 3D. [Online]. Available: http://www.spinor.com
[11] J. Aydelotte and A. Ebrahimi, "Game streaming: A planned approach," *Game Development Tools*, 2011.
[12] S. Gorlatch, F. Glinka, *et al.*, "Designing multiplayer online games using the real-time framework," in *Algorithmic and Architectural Gaming Design: Implementation and Development*, A. Kumar, J. Etheredge, *et al.*, Eds. IGI Global, 2012, pp. 290–321.
[13] J. Ferris, M. Surridge, *et al.*, "Edutain grid: A business grid infrastructure for real-time onine interactive applications," in *Grid Economics and Business Models, ser. Lecture Notes in Computer Science*, J. Altmann, D. Neumann, *et al.*, Eds. Springer Berlin Heidelberg, vol. 5206, 2008, pp. 152–162.
[14] 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.
[15] B. Lantz, B. Heller, *et al.*, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, New York, NY, USA: ACM, 2010, pp. 1–6.
[16] Iperf. (2013). [Online]. Available: http://iperf.sourceforge.net.
[17] Floodlight Open Flow. Controller. (2013). [Online]. Available: http://www.projectfloodlight.org/floodlight
[18] Nicira, "Network virtualization platform (NVP) white paper," 2013.
[19] A. D. Ferguson, A. Guha, *et al.*, "Participatory networking: An API for application control of SDNs," in *Proc. SIGCOMM 2013, ACM*, 2013, pp. 327–338.

Tim Humernbrum received his Diploma degree in computer science from the University of Münster (Germany) in 2013. He then started his PhD studies under the supervision of Sergei Gorlatch in the research group parallel and distributed systems at the University of Münster. His main research interest includes high-level programming models for distributed systems.

Frank Glinka received his computer science degree from the University of Muenster in 2006 and is now a research associate at the department of computer science in the group of Prof. Sergei Gorlatch. He has worked as a work package leader for the European research projects edutain@grid and OFERTIE covering the topic of real-time application services and is currently completing his PhD thesis titled "Developing Grid Middleware for a High-Level Programming of Real-Time Online Interactive Applications".

Sergei Gorlatch has been Full Professor of Computer Science at the University of Münster (Germany) since 2005. Earlier he was Associate Professor at the Technical University of Berlin, Assistant Professor at the University of Passau, and Humboldt Research Fellow at the Technical University of Munich. Prof. Gorlatch has about 200 refereed publications in renowned inter-national journals and conferences. He has led several international research and development projects in the field of parallel, distributed and Grid computing, funded by the European Commission, as well as by German national bodies. Sergei Gorlatch holds MSc degree from the State University of Kiev, PhD degree from the Institute of Cybernetics of Ukraine, and the Habilitation degree from the University of Passau (Germany).