Improving QoS in Real-Time Internet Applications: From Best-Effort to Software-Defined Networks

Sergei Gorlatch
University of Münster, Germany
Email: gorlatch@uni-muenster.de

Tim Humernbrum
University of Münster, Germany
Email: humernbrum@uni-muenster.de

Frank Glinka
University of Münster, Germany
Email: glinkaf@uni-muenster.de

***Abstract*—Real-Time Online Interactive Applications (ROIA), e.g., multiplayer online games and simulation-based e-learning, are emerging internet applications that 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 behaviour to meet the dynamic QoS demands, such that most ROIA use the underlying network on a best-effort basis. The emerging architecture of Software-Defined Networking (SDN) decouples the control and forwarding logic from the network infrastructure, making the network behaviour programmable for applications. This paper analyses ROIA requirements on the underlying network and describes the specification of an SDN Northbound API that allows ROIA applications to specify their dynamic network requirements and to meet them using SDN networks.**

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)* [2] technology to address the dynamic network demands of ROIA: SDN enables applications to manage the network behaviour 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 centralised *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 defines precisely how an application communicates with the SDN Controller. We focus in this paper on the SDN Northbound API for the emerging ROIA internet applications.[1]

In the following, we systematically analyse the most important ROIA scenarios regarding network requirements which contribute to the specification of a Northbound API for ROIA (Section II). In Section III, we specify the desired API functionality features, followed by an experimental study about 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.

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

Figure 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 Figure 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.

[1]For the so-called Southbound API that connects the SDN Controller with the network infrastructure, standardised solutions like OpenFlow exist; this API is not considered in the current paper.

978-1-4799-2358-8/14/$31.00 ©2014 IEEE

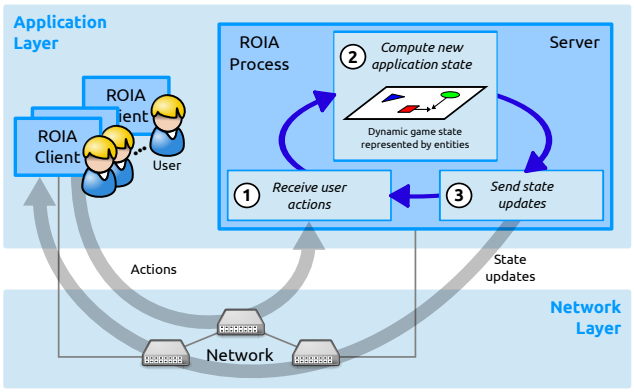

Fig. 1. Structure of a ROIA and its real-time loop.

In the following, we analyse 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 genre where the players move in a 3D game world, see the world from their own perspective, and fight against enemy players. Shooters 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 [4]. 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 [5]. In that case, 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. A scalable solution is achieved either by replicating the application across multiple network sites, e.g., located closer to the clients, or by using multicast [6].

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 prioritise 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. ROIA Network QoS and API Specification

In the following, we analyse 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 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 synchronisation*, *state migration*, and *asset transfers*. In ROIA, state synchronisation 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 a 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 [7]. While state synchronisation omits non-changed data from the transmission, state migration requires a full serialisation and transmission of the migrated entity.

Different data flow types generate different network requirements: while state synchronisation 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 comparatively freely 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 flow types (e.g., state synchronisation 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.

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 synchronisation packets, the amount and size of packets is unknown and can hardly be predicted. Moreover, ROIA developers often have no detailed technical knowledge of networking and the involved 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 used which are then automatically translated to low-level metrics understood by the SDN Controller.

When sending state synchronisation updates to multiple clients, a server sends different data for each client connection and entity. 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 synchronisation 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 unique 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. synchronisation within limited bandwidth scenarios is desirable yet difficult. If synchronisation 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 synchronisation QoS may be better than giving the migrated player a temporarily very poor QoS. Assigning state migration a higher priority than synchronisation 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 prioritisation of data flows, e.g., state synchronisation over state migration.

IV. Experiments with QoS Metrics

In this section, we present an experimental study about the relation between the low-level and the application-level QoS metrics as discussed in API Feature 5. Our experiments with an FPS-like game investigate the relation between ROIA QoS and low-level metrics like 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 FPS 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.

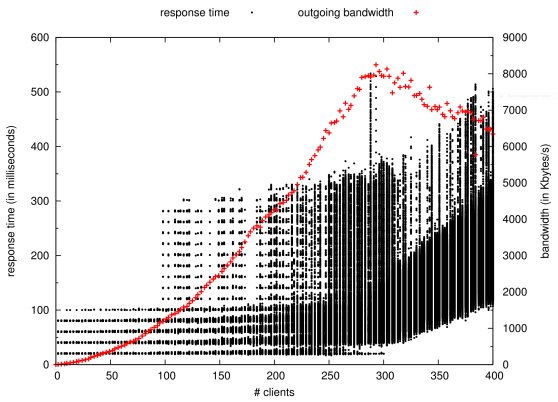

Fig. 2. Response time and bandwidth consumption of an example game

Figure 2 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 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. Thus, Figure 2 demonstrates that the bandwidth metric does not precisely mirror the changes in QoS observed by the user.

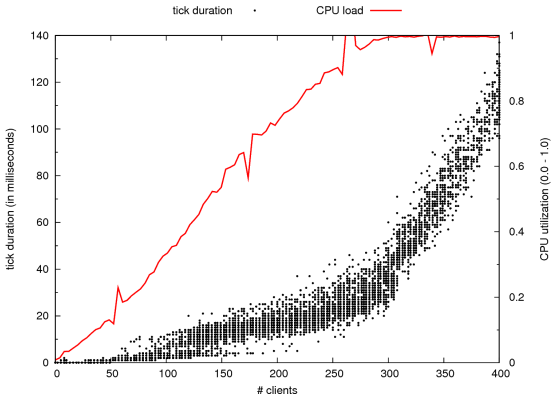

Fig. 3. Server-side processing time for one real-time loop iteration (tick duration) and average CPU load in an example game

In our next experiment, Figure 3 shows the time spent on the server for one iteration of the real-time loop (so-called tick duration), depicted by points, depending on 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 utilisation of 90 %. Therefore, again, the low-level metric "CPU load" does not precisely express the QoS behaviour.

From this analysis, it follows that there is no server-side reason for the increased response time on >100 clients as observed in Figure 2; 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.

V. SDN Northbound API: Initial Design

The advantage of SDN is that the Northbound API implementation employs the SDN Controller to accommodate the QoS requirements specified by the ROIA developer. This is done by adapting the network, e.g., by prioritising packets of a particular flow or by redirecting flows if certain routes in the network are congested. In contrast to the traditional networking scenario shown in Figure 1 where ROIA has no possibility of influencing the network, Figure 4 shows an SDN scenario for ROIA with our Northbound API realised by the so-called SDN Module. The scenario includes the following eight 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, 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 Figure 4).

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 are configured by the SDN Controller.

QoS policy: a data structure that expresses the network requirements for specific data flows. A QoS policy 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 $<5\,\%$ of the data packets from a ROIA process to a client are lost and that >2 Mbit/s throughput is achieved. The SDN Module currently supports the following network metrics: *latency* in milliseconds (ms), *throughput* in Bit per second (Bit/s), *packet loss* in %, and *jitter* in ms.

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 for development and runtime support of ROIA. RTF [8] has been developed at the University of Münster, starting with the European edutain@grid project [9]; 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.

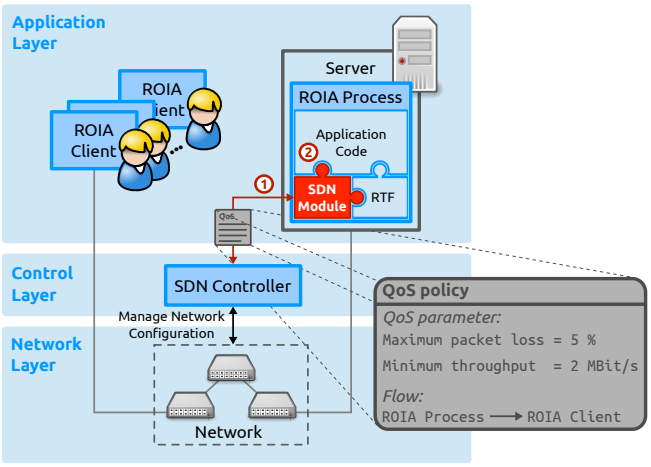

Fig. 4. Architecture of SDN networking for ROIA: a ROIA process sends a network QoS request to the SDN Controller which tries to accommodate the request by adapting the network

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 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 Figure 4 and cover:

a) a base API (① in Figure 4) 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 application network management, e.g., rejecting requirements which cannot be accommodated. We focus on network control functionality that is particularly important for ROIA, i.e., metrics that should be controlled and monitored in order to manage the QoS of ROIA. 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.

b) An *application-level API* (② in Figure 4) 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. 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 promising approach for addressing the dynamic QoS demands of ROIA.

Within the SDN community, there have been recent activities towards creating and standardising 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 [10] and the Nicira Network Virtualization Platform (NVP) API [11] which are relatively new and still few applications use them. Furthermore, they require a detailed technical knowledge of the network infrastructure and protocols. *Participatory networking (PANE)* [12] offers a Northbound API which delegates read and write authority from the networks administrators to the end users and applications which can reserve guaranteed minimum bandwidth, issue path control requests, set rate-limits 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 a ROIA developer's point of view. Therefore, our goal is two-fold: a) analyse and motivate additional networking metrics that should be supported by a network-level API, and b) 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., the PANE controller [12].

VII. Acknowledgements

We would like to thank our project partners, especially Folker Schamel and Michael Franke from *Spinor GmbH* and Eduard Escalona and Iris Bueno from *i2Cat Foundation* for valuable discussions and sharing their expertise on the design of ROIA and SDN controllers. Our research has received funding from the EC's 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 *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] T. Beigbeder *et al.*, "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003," *Proceedings of ACM Network and System Support for Games Workshop*, 2004.
[5] T. Szigeti, "Quality of Service Network Design Considerations for Cisco TelePresence Systems," *Technical Services Newsletter*, 2011.
[6] P. Zhang *et al.*, "A Study of Video-on-Demand Learning System in E-learning Platform," *CSSE '08*, vol. 05, pp. 793–796, 2008.
[7] C. Ghosh *et al.*, "An architecture supporting large scale MMOGs," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '10. ICST, Brussels, Belgium, Belgium: ICST, 2010, pp. 2:1–2:8.
[8] S. Gorlatch *et al.*, "Designing Multiplayer Online Games Using the Real-Time Framework," in *Algorithmic and Architectural Gaming Design: Implementation and Development*, A. Kumar *et al.*, Eds. IGI Global, 2012, pp. 290–321.
[9] J. Ferris *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 *et al.*, Eds. Springer Berlin Heidelberg, 2008, vol. 5206, pp. 152–162.
[10] "Floodlight OpenFlow Controller. `http://www.projectfloodlight.org/floodlight`," 2013.
[11] Nicira, "Network Virtualization Platform (NVP) White Paper," 2013.
[12] A. D. Ferguson *et al.*, "Participatory networking: an API for application control of SDNs," in *Proceedings of the ACM SIGCOMM 2013*. ACM, 2013, pp. 327–338.