A Flexible GraphQL Northbound API for Intent-based SDN Applications

Fetia Bannour
ENSIIE & SAMOVAR
Evry, France
fetia.bannour@ensiie.fr

Stefania Dumbrava
ENSIIE & SAMOVAR
Evry, France
stefania.dumbrava@ensiie.fr

Damien Lu
ENSIIE
Evry, France
damien.lu@ensiie.fr

***Abstract*—In a Software-Defined Networking (SDN) architecture, Northbound, Southbound, and East-Westbound APIs are used to describe how interfaces operate between the three SDN planes, namely the data, control, and application planes. Apart from the standardization of the Southbound interface, for which OpenFlow has emerged as the widely-accepted standard, there is to date no open and vendor-neutral standard for the Northbound and East-West interfaces to provide the required interoperability between different SDN controller platform designs. This paper addresses the lack of a well-defined standard for the Northbound API that is used for the interaction between the applications and the SDN controllers, by proposing a GraphQL-based Northbound API design for the SDN controllers in the context of large-scale deployments. Our proof-of-concept methodology was validated and evaluated for an intent-based routing application that we designed on top of the ONOS controllers. When compared to ONOS's native REST API, our Northbound API model proved efficient in optimizing different performance metrics (i.e the number of requests, the request execution time, and the throughput) on both synthetic and real-world network topologies (like Renater and China Telecom) that are emulated using Mininet.**

I. Introduction

SDN [1], [2] is an emergent network architecture paradigm that allows openness, programmatic management, flexible control, and resource optimization. Compared to traditional networking, SDN has the potential to enable rapid innovation, automation, and to adapt to the dynamics of today's applications. In the SDN architecture, the control logic is decoupled from the underlying hardware devices. Also, network intelligence is logically centralized in software controllers [3]. These aim to ensure a global and consistent view of the network, by handling communications between network applications and data plane devices using open interfaces, referred to as Application Programming Interfaces (APIs).

To simplify network management in SDN and facilitate the tasks of network application developers, there has been an increasing interest in designing suitable *abstractions* for the network topology and the underlying physical network infrastructure. These have been extensively studied when developing interfaces for the Southbound API (between the control and data planes), in the context of standardizing the OpenFlow communication protocol [4]. However, little attention has been devoted to studying suitable abstractions for both the Northbound API (between the control and application planes) and the East-Westbound API (between the distributed SDN controllers in the control plane). In particular, addressing the lack of a standardized Northbound (NB) interface, relies not only on finding a suitable abstraction of the network topology (the global network view), but also on defining a proper API (transport language) to ensure an efficient NB communication between the control and application planes. As there is no standardized NB interface, some of the current SDN controllers propose their own ad-hoc Northbound APIs, while many other controllers use the Web-based REST API. Given the importance of the NB interface for SDN application developers, we aim to leverage the potential of GraphQL-based [5] approaches to ensure a more efficient and scalable communication between the SDN controllers and applications.

In this work, we propose a proof-of-concept methodology for building a GraphQL-based API as an SDN Northbound API. The advantages of using the GraphQL API over REST in SDN include the increased performance and flexibility of requests and the rapid application development. Some of these advantages are also discussed in [6]. Our approach is implemented on the open-source ONOS controller [7]. It is evaluated for an off-platform intent-based routing application that we designed on top of ONOS and which leverages ONOS's intent framework. Generally, Intent Based Networking (IBN) is an emergent concept that offers users the ability to express what they want to achieve in a high-level manner rather than how to achieve it, thereby easing network management.

Outline: The rest of the paper is organized as follows. In Section II, we review the NB API designs used by state-of-the-art SDN controller platforms, and works comparing RESTful and GraphQL APIs. In Section III, we present the proposed methodology for building our GraphQL-based NB API model for SDN controllers. We also describe the implementation of our GraphQL API and an off-platform intent-based routing application on ONOS. Section IV provides an experimental analysis comparing the performance of our GraphQL API with that of ONOS' native REST API for our designed application on different test scenarios. Finally, Section V summarizes our approach and outlines possible directions for future work.

II. Background

A. Northbound APIs in SDN

The Northbound API, which is considered as one of the key components of an SDN architecture, represents the communication channel between the control plane and the

978-1-6654-0601-7/22/$31.00

application plane. More specifically, it provides a common abstraction of network functions to the upper layers, namely the SDN applications and management systems. The latter use this interface to consume the network services, configure the network dynamically, and dictate the behavior of the network, regardless of the underlying network infrastructure hardware.

Contrary to the Southbound API, the standardization of the Northbound API is an ongoing topic of discussion, given the different SDN application requirements. Hence, many Northbound API designs have been adopted by SDN controller platforms. They can be classified into two categories [1].

The first category involves simple and primitive APIs that are directly linked to the controller's internal services. These implementations include low-level specialized ad-hoc APIs that are proprietary and controller-specific, as well as Web-based APIs like the widely-used REST API. The latter has been adopted by current SDN controllers like Floodlight and ONOS. The second category includes higher level APIs that rely on domain-specific programming languages, such as Procera and Pyretic [8], to provide a wide range of powerful abstractions that make it easier for application writers to develop software modules and SDN-enabled programs.

B. REST and GraphQL APIs

APIs provide protocols that can serve as a contract, describing the data that web services accept and return to clients.

1) REST API: RESTful APIs conform to the REST architectural style guidelines (uniform interface, client-server decoupling, statelessness, cacheability, layered system architecture) and allow to interact with RESTful web services. The most common REST operations are GET, POST, PUT and DELETE, used to read data from resource URIs, create, update, and, respectively, destroy resources.

2) GraphQL: GraphQL is novel language for implementing Web-based, client-driven APIs, which allows to specify a common abstraction layer (schema) between the client and the server. The GraphQL Schema comprises object types, which contain a collection of fields, each with their own type and with values from back-end data stores (obtained using resolver functions). As the language supports arbitrary nesting, it is especially tailored to compactly describe hierarchical data that needs to be fetched by queries or modified through mutations. The language's flexibility allows the client side to exactly specify the needed information, without impacting the server-side. This avoids under- and over-fetching problems, common to REST APIs, and leads to more agile front-end development.

3) REST and GraphQL Comparison: Recent research works focused on comparing REST and GraphQL. In [9], Seabra *et al.* conducted a study in which they compared these techniques in terms of performance, but their work has been carried out in a different context. Thus, their results cannot be directly applied to our SDN approach. Another work found in [6] studies the differences in implementation between these two architectural techniques. Specifically, the authors perform an experiment that investigates the time taken to implement requests on a Web server. Based on this study, the authors conclude that implementing GraphQL queries is less time consuming than implementing REST queries, especially when performing complex queries with multiple parameters.

Essentially, the GraphQL syntax makes it easy to understand the code and requires less effort to specify the parameters. Hence, the GraphQL queries proposed by the study participants are all correct, unlike the proposed REST queries.

On the other hand, a single GraphQL query can possibly call several other resolvers allowing to increase the complexity of the queries, without multiplying the REST queries (as they are limited to a single function) or increasing the difficulty of implementing the queries. In the REST case, the risks of errors are particularly high at the syntax level, and the queries can quickly become long and difficult to read.

III. The Proposed GraphQL-based Northbound API for SDN Controllers

We design a generic method that is agnostic to the controller choice. To this end, we mount a GraphQL API server intermediary between the SDN controller and the external application.

A. Workflow Overview

The SDN architecture is fully implemented in the virtual machine accessible on the local network through its IP address. In fact, access to SDN network information, from the external off-platform application, is done through the Northbound interface via REST or GraphQL, as can be seen in Fig. 1.

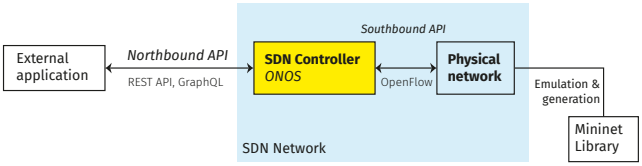


Fig. 1: Pipeline overview

The external application is independent from the ONOS controller and only has access to the information transmitted via its API endpoint. In particular, we focus on the application in charge of intent installation. Given a pair of hosts, this application will install the rules necessary to route the packages passing between them. We propose a dynamic method which propagates updates to and from the network.

B. Design of a GraphQL Northbound API

We build a GraphQL server serving as an intermediary between external applications and the ONOS controller. Thus, when the GraphQL server receives requests from a given application, it will execute them through the ONOS CLI. However, this gives additional latency to each GraphQL query, since it must pass through the GraphQL server before reaching the ONOS controller. This latency must be taken into account when comparing our GraphQL API with the REST-based one. Hence, we build, on the same GraphQL server, an alternative REST API, which uses the ONOS CLI commands, exactly like GraphQL. The latency due to the intermediate server is, thus, identical between the two APIs.

Note that the GraphQL and REST servers are, like the external application, independent of the ONOS controller. The implementation of the GraphQL server relies on encoding the JSON responses obtained from the ONOS CLI as GraphQL types. Based on this, we build the general GraphQL schema, by defining the types of queries, mutations, and resolvers.

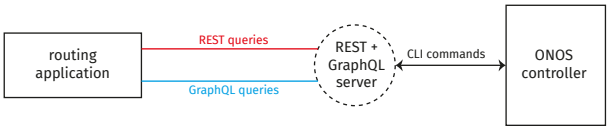


Fig. 2: Interaction between our application and the controller

C. Use Case: Intent-based SDN routing application

To test our approach's feasibility, we developed a proactive intent-based routing application (see Fig. 3). Note that our approach is generic and not only specific to intent-based applications. Given intents expressed by users in ONOS's Intent Framework, this installs, in the corresponding switches, the OpenFlow rules needed to enable host communication. Unlike in reactive routing, these OpenFlow rules do not expire.

a) Path computation: Once the user provides a pair of hosts to communicate, the application computes the shortest-path needed to connect them. This is computed using Dijsktra's algorithm, as implemented in the Networkx Python library. The path is specified as a list containing the IDs and ports of hosts, as well as that of the switches through which the packets have to transit. It thus suffices to process this list and install, in each switch, the needed routing rules.

b) Intent installation: Next, we install the same intents to connect the input host pairs, using the REST and GraphQL APIs. The REST approach uses $2 \times n$ GET and POST requests, where n is the size of the considered path. GraphQL uses a single query, with the list of intents as its argument.

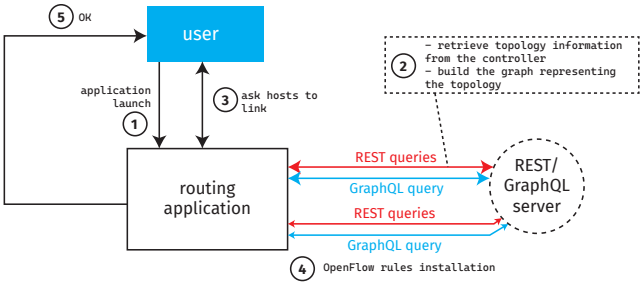


Fig. 3: Application overview

IV. Performance Evaluation

A. Experimental setup

a) Test environment: Our experiments are performed on a VirtualBox virtual machine running an Ubuntu 18.04 LTS server with two 3.8GHz 64-bits Intel Pentium CPUs, 5 GB of memory and a 250 GB solid-state drive. We used ONOS 2.6 [7] and Mininet 2.3.0 to start an emulated ONOS network on a single development machine; including a logically-centralized ONOS cluster, a modeled control network, and a data network.

b) Network topologies: We used Mininet to generate synthetic network topologies (torus and tree) and real-world network topologies from "The Internet Topology Zoo" [10] (Renater and China Telecom) (cf. Table 4). For real-world topologies, we used a Python converter [11] that parses the network topologies in the GraphML format from The Internet Topology Zoo, and then creates the Mininet topologies.

Topologies \ Components	Switches	Hosts	Links
Torus 3 x 3	9	9	36
Torus 4 x 4	16	16	64
Tree (depth 4, fanout 2)	15	16	28
Renater	43	43	112
China Telecom	42	40	132

Fig. 4: Characteristics of the topologies used in the analysis.

c) Performance metrics: To compare ONOS's REST API with the GraphQL-based API we designed for ONOS's Northbound interface, we use the intent-based routing application that we developed on ONOS. For each network topology, and for a given number N of host pairs to be linked, we propose to consider and measure the following performance metrics :

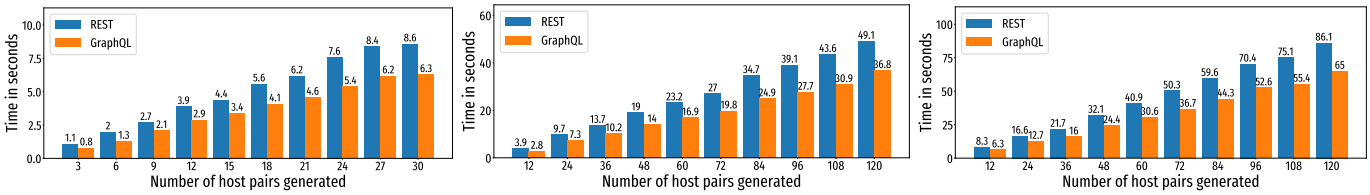
- The number of REST/GraphQL requests required to connect the N couples of hosts,
- The time needed to execute these requests (the intent installation delay: the sum of the delay to reach the GraphQL server and that to install the intents on switches)
- The deduced throughput for both types of API.

For the synthetic topologies, we vary their type. Then, for a fixed type, we vary their corresponding size.

B. Results

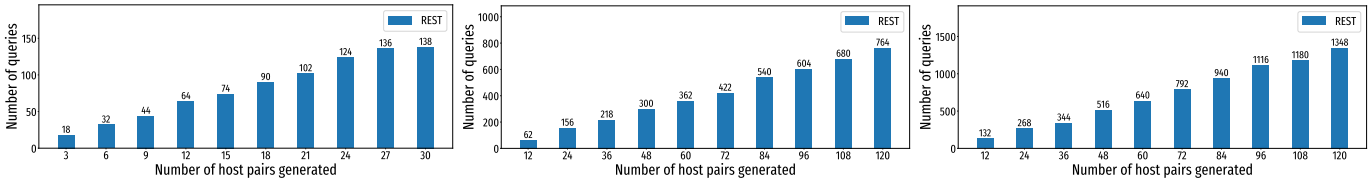
We first compare the RESTful and GraphQL APIs, in terms of the time required to generate and install OpenFlow rules, for a given topology, and a given number of host pairs.

The results, for the synthetic and real-world topologies we considered, are summarized in Figures 5, 6, and 7. These confirm the superiority of the GraphQL-based approach, which leads to a time efficiency gain of approx. 27%, on each of these configurations. The reason is the relative number of GraphQL queries and, respectively, REST requests that need to be transmitted. Specifically, for n intents to be installed, $2 \times n$ REST requests are required, while only one GraphQL query suffices. This is due to the expressiveness of GraphQL, whose support for arbitrarily nested fields and relationships allows to retrieve comparatively more information than simple REST queries. While one could set up a routing path within the same GET/POST request, data structuring is more difficult with REST, since there is no standardized resolver on the server side, as in GraphQL. Even with modifying ONOS's REST API to send all intents at once, as in GraphQL, one has to deserialize on the server side to well-structure the received data. We avoid this bottleneck by leveraging GraphQL and encoding all needed information in a single query.



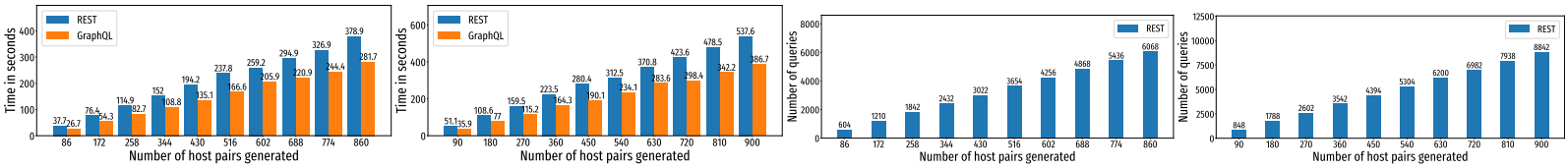
(a) Torus topology (3x3) (b) Torus topology (4x4) (c) Tree topology (depth 4, fanout 2)

Fig. 5: GraphQL vs. REST runtime performance comparison (generation time of intents rules)



(a) Torus topology (3x3) (b) Torus topology (4x4) (c) Tree topology (depth 4, fanout 2)

Fig. 6: Number of generated REST requests (number of needed queries to generate intent rules)



(a) China topology (b) Renater topology (c) China topology (d) Renater topology

Fig. 7: GraphQL vs. REST runtimes (a, b) and number of REST requests (c, d) for intent generation for real-world topologies

Figures 6 and 7 show that, for our intent-based routing application, the number of necessary intents ranges from less than 1.5 K for the synthetic topologies, to 6K and respectively 8K for the China Telecom and Renater real-world topologies. This leads to the generation of up to 7K REST requests for the synthetic topologies, as well as 33K and, respectively, 48K requests, for China Telecom and Renater. This result trend also holds when running our experiments under information loss.

One might think that the advantage would be more significant for GraphQL. Indeed, if we send only one request to the GraphQL server, it has to decrypt and execute it, thus calling the intent installation function for each element of the previously sent list, i.e., $2 \times n$ times. As the calls are local, we gain the network latency times we would have with REST.

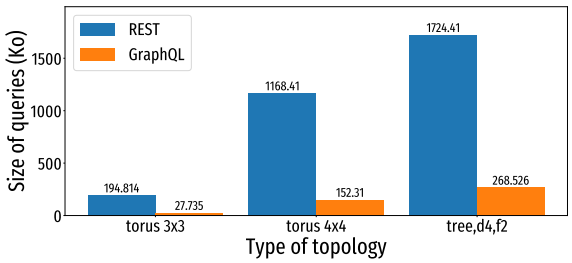


Fig. 8: REST/GraphQL query size relative to the topology type

As can observed in Figure 8, the size of the requests is much smaller with GraphQL. Indeed, as can be noticed from the synthetic topologies we considered, the size of the GraphQL queries is, on average, approx. 14% of that corresponding to the REST requests needed to install the same intents.

A similar explanation for this efficiency gain is that the GraphQL language allows object nesting. As such, a single POST query can contain the list of all the intents to be added. The succinctness and expressivity of GraphQL queries make it possible to transfer a large amount of information, without having to overload the network with a high number of queries. This has the added benefit that there is a much lower risk of losing or altering information during the process. Given that in the SDN setting the controller alone is responsible for the network control, these reliability advantages are critical.

V. Conclusion

In our experimental study, we found that our GraphQL Northbound API outperforms REST on both synthetic and real-world emulated networks. Specifically, on each of the considered topologies, our approach leads to a speed-up of approximately 27% in intent rule installation. Also, we have shown that relying on GraphQL leads to a dramatic overhead reduction, as only one query is needed to communicate with the external application we consider, while the REST approach can require up to 9K requests for the same purpose. These promising results show the feasibility of our proposed methodology and its scalability to realistic topologies.

Looking ahead, a main direction of future work consists in integrating our GraphQL API with a database representing a dynamic network topology, in the context of developing a web-scalable database-defined network architecture.

REFERENCES

[1] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[2] M. He, M.-Y. Huang, and W. Kellerer, "Optimizing the flexibility of sdn control plane," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9.

[3] V. Huang, G. Chen, P. Zhang, H. Li, C. Hu, T. Pan, and Q. Fu, "A Scalable Approach to SDN Control Plane Management: High Utilization Comes With Low Latency," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 682–695, 2020.

[4] D. Espinel Sarmiento, A. Lebre, L. Nussbaum, and A. Chari, "Decentralized sdn control plane for a distributed cloud-edge infrastructure: A survey," *IEEE Communications Surveys Tutorials*, vol. 23, no. 1, pp. 256–281, 2021.

[5] "GraphQL," https://graphql.org/.

[6] G. Brito and M. T. Valente, "REST vs graphql : A controlled experiment." arXiv, 2020.

[7] "The ONOS project," https://onosproject.org/.

[8] Z. Latif, K. Sharif, F. Li, M. M. Karim, S. Biswas, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *Journal of Network and Computer Applications*, vol. 156, p. 102563, 2020.

[9] M. Seabra, M. F. C. Nazário, and G. Pinto, "REST or GraphQL?: A performance comparative study," in *SBCARS*. ACM, 2019, pp. 123–132.

[10] "The Internet Topology Zoo," http://www.topology-zoo.org/dataset.html.

[11] "The GraphML-Topo-to-Mininet-Network-Generator converter," https://github.com/sjas/assessing-mininet/blob/master/parser/GraphML-Topo-to-Mininet-Network-Generator.py.