POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Model-Based Abstraction Layer for Heterogeneous SDN Applications

Original
A Model-Based Abstraction Layer for Heterogeneous SDN Applications / Castellano, Gabriele; Cerrato, Ivano; Gharbaoui, Molka; Fichera, Silvia; Martini, Barbara; Risso, FULVIO GIOVANNI OTTAVIO; Castoldi, Piero. - In: INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS. - ISSN 1074-5351. - STAMPA. - (2019). [10.1002/dac.3989]

Availability:
This version is available at: 11583/2752481 since: 2019-09-17T20:10:58Z

Publisher:
Wiley

Published
DOI:10.1002/dac.3989

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Wiley postprint/Author's Accepted Manuscript

This is the peer reviewed version of the above quoted article, which has been published in final form at http://dx.doi.org/10.1002/dac.3989.This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

(Article begins on next page)

26 April 2024

Received 0000-00-00; | Revised 0000-00-00; | Accepted 0000-00-00

DOI: xxx/xxxx

ARTICLE TYPE

A Model-Based Abstraction Layer for Heterogeneous SDN Applications

Gabriele Castellano[1] | Ivano Cerrato[1] | Molka Gharbaoui[3] | Silvia Fichera[2] | Barbara Martini[3] | Fulvio Risso[1] | Piero Castoldi[2,3]

[1]Dept. of Computer and Control Engineering, Politecnico di Torino, Torino, Italy
[2]Scuola Superiore Sant'Anna, Pisa, Italy
[3]National Inter-University Consortium for Telecommunications (CNIT), Pisa, Italy

Correspondence
Fulvio Risso, Corso Duca degli Abruzzi, 24, 10129 Torino TO, Italy.
Email: fulvio.risso@polito.it

Present Address
—

Summary

Modern controllers for Software-Defined Networks (SDN) enable the execution of arbitrary SDN applications (e.g., NAT, traffic monitors) that may be exploited by an overarching set of services (e.g., application-layer orchestrators) to build even richer services. To this purpose, the above overarching services require a mechanism that allows reading the run-time state and writing the configuration of arbitrary SDN applications, possibly through an uniform API. Unfortunately, most SDN applications are not designed/implemented by taking into account the possibility to be used as part of higher level service workflows (e.g., a complex Intrusion Prevention System that leverages multiple elementary services as individual components), hence they may not provide an adequate interface that would allow overarching services to exploit their features. This paper addresses this problem by proposing an approach to represent the run-time state of arbitrary applications, where data are exported according to high-level model-based structures. Furthermore, the mapping from the high-level data model to the actual data representation within the SDN application is enabled by a suite of algorithms that are generic enough to operate independently of the actual source code of the application, thus avoiding undesired and invasive modifications to existing applications. The paper also presents a software framework and a prototype implementing the proposed approach, characterizes the resulting performance, and discusses pros and cons of the proposed approach.

KEYWORDS:
SDN, Network Services, Network Orchestration, Data Modeling, YANG

1 | INTRODUCTION

Modern controllers for Software-Defined Networking (SDN), such as ONOS[1] and Open Daylight[2], offer high-level programming abstractions that allow to develop software programs (i.e., *SDN applications*, or *SDNApps* in this paper) to perform arbitrary network-related tasks going beyond forwarding, routing or topology discovery operations, such as application-based traffic monitoring, firewall, NAT, deep packet inspection. This paradigm enables the creation of programmable network environments able to address the emerging communication requirements (e.g., QoS, security) of modern application services (e.g., Smart City, Cloud Robotics) operating on top of a distributed and dynamical environment of networked smart systems (e.g., sensors, smart terminals and buildings) to provide high-value and cooperative services to users.

These smart environments are featured by highly-changing and heterogeneous contexts of users, resources, and networks that require punctual and reactive network and service management operations to adapt systems to context changes and to assure and preserve proper levels of user experience[3]. In this respect, the use of multiple SDNApps in the scope of composite workflows (even executed in different SDN controllers) can lead to the delivery of richer added-value applications enabled by control and coordination tasks performed by higher-level *service control applications* (shortened as *ServiceApps* in this paper). In particular, ServiceApps may be in charge of collecting context information (e.g., user location, device capabilities, service availability) and, accordingly, taking coordinated actions (e.g., issuing of configuration commands) in order to prevent/recover from service corruptions or degradations. For instance, in the area of security and authorization, a ServiceApp could be an access control system extended with the capability of detecting anomalies while users make use of resources under control (i.e., usage control systems)[4]. In case of network environments, such an usage control system (acting as a ServiceApp) may focus on tracking the usage of bandwidth capacity. Thus, it may be interested in the retrieval of data throughput on per-user flow basis by leveraging an SDNApp (e.g., traffic monitoring analytics tool). If a misuse is detected according to a set of policies (e.g., violation of SLA in terms of data throughput), the usage control system can promptly take countermeasures by leveraging other SDNApps (e.g., traffic shapers) to keep resources usage within the agreed terms (e.g., traffic profile reconfigurations to shrink usable data throughput).

In more general terms, in the envisioned scenarios a ServiceApp may need to actively interact with diverse (set of) SDNApps both to collect their current run-time state and to modify previously established configurations[1]. Unfortunately, SDNApps may not consider the possibility to be used as part of an higher level service workflow. First, SDNApps generally offer only a partial view of their state, such as a NAT that exports its global configuration but hides the run-time information available in the NAT table. Furthermore, SDNApps may keep data (e.g., configuration and run-time state) in completely arbitrary structures, as well as expose such information through arbitrary APIs, exploiting diverse protocols such as NETCONF[5], RESTCONF[6] and more. This introduces lack of flexibility in ServiceApps, which must then be designed to work with specific SDNApps implementations. For instance, to interact with two different firewalls, a ServiceApp may need to use two different APIs, which are defined by those applications; the support for a third firewall might then require an extension of the ServiceApp since, very likely, such a firewall will use a third API different from the other two. Ultimately, this heterogeneity in SDN application implementations may prevent ServiceApps from being effectively able to take run-time decisions based on the current context of underlying network layer, or at best, makes their implementation more difficult as they may need to handle different southbound interfaces, with the consequent additional complexity and possible portability issues when required to select a different implementation of the same network function.

To cope with the aforementioned challenges, this paper presents an abstraction layer enabling ServiceApps to dynamically inspect the run-time state and/or change the current configuration of any SDNApp based on implementation-agnostic data-models, i.e., model-based abstraction layer, also enabling ServiceApps to promptly react to any event occurring in the controlled infrastructure, exploiting even services (e.g., network control) that were not originally engineered to export their data to an external consumer (e.g., application service orchestrators). Indeed, the proposed methodology can be easily applied, under few assumptions, to all existing SDNApps, introducing almost no changes in the original code of the application nor appreciable overhead, hence bringing the benefits of cross-application state inspections to a large set of existing services.

More specifically, the contribution of this paper is twofold.

First, we propose an overarching software architecture that enables novel ServiceApps to leverage information (including the run-time state) exported by multiple infrastructure-specific applications (SDNApps), with the aim of creating complex workflows spanning across multiple infrastructure domains (e.g., possibly managed with different SDN controllers), and independent from the actual implementation of any given network application. The proposed architecture leverages an abstraction layer based on a YANG[7] data model associated with each SDNApp, which describes the configuration and run-time state of such an application in an implementation-independent way, and a set of application-agnostic high-level APIs that are used by an external module (e.g., the ServiceApp) to access such data. The latter will support multiple communication patterns that enable either direct communication between the involved parties (e.g., through HTTP REST[8] operations), or a publish/subscribe paradigm (e.g., through a message broker) for more detached operations such as solicitations/advertisements.

Second, we define a set of application-agnostic *mapping algorithms* (and the companion prototype implementation) that map incoming requests for a specific data, that are specified according to the high-level YANG-based service abstraction, into a read/write operation of the actual run-time internal variable(s) of the SDNApp. These mapping algorithms are general enough to be independent from the specific SDNApp logic (and source code). Moreover, they can be put into operation without requiring

[1] In the reminder of the paper, the expression “access the run-time state”, referred to an SDNApp, will be used to indicate both these operations.

the application developer to create specific code to provide access to the selected data. In fact, to enable the ServiceApp to access any internal variable of arbitrary SDNApps, developers can simply include our application-agnostic mapping module in the SDNApp code as an external linked library.

Finally, the paper presents numerical results validating the approach and featuring the performance of the proposed software architecture and mapping algorithms in terms of execution time of both *read* and *write* operations as well as in terms of notification latency. Moreover, the paper assesses the performance of our approach under specific use cases and evaluates its overhead with respect to a direct access to the applications variables. Results show that the execution of the proposed software framework introduces a relatively low overhead and provides insights on how to optimize notification performance. Results also confirm that the overall latency introduced by our approach is strictly related to the nature of the operations executed in every adopted use case. Finally, the paper presents a discussion about the advantages of the proposed approach against the requirements SDNApps have to fulfill to be part of the proposed software framework.

This paper is structured as follows. Section 2 analyzes related works. Section 3 presents some use cases that highlight possible deployments where the solution presented in this paper can be adopted. Section 4 presents the overall software architecture. Section 5 describes the algorithms that transparently map the SDNApps variables in a YANG-based structure. Validation, including both mapping algorithms and the complete architecture, is carried out in Section 6. Finally, Section 7 discusses the main lessons learned while prototyping and validating this approach, and Section 8 concludes the paper.

2 | RELATED WORKS

This section presents the state of the art in main related research areas and highlights the contribution of this paper in comparison with the reported works.

A number of recent research works address the management of SDN application state [9,10,11]. In particular, OpenState [9] and P4 [10] propose modern data planes that allow instantiating stateful flow rules whose output may change based on previous processed traffic. In [11], instead, authors address the problem of consistency between control plane and a stateful data plane. However, these works focus on the limitation of standard SDN data planes (usually OpenFlow) rather than considering the problem of exposing the application state to an external and generic service, that has not been pre-defined in advance, in order to enable dynamically-established service workflows.

Generic modeling of SDN applications has been introduced in the Open Daylight controller [2], starting from version 2.0 (*Helium*), with a technology called Model-Driven Service Adaptation Layer (MD-SAL), which is available also in other proprietary software such as Cisco Network Services Orchestrator [12]. MD-SAL is an extensible middleware component that provides messaging and data storage functionality based on data and interface models defined by application developers (i.e. user-defined models), which rely on the YANG language. In this paper we also present a similar abstraction layer with data models based on YANG. However, the approach we propose is applied at a different level, i.e., on top of SDN controller and applications. Moreover, the proposed abstraction layer is agnostic with respect to the employed network controller (e.g., ONOS, ODL) and the logic of involved applications, also not requiring any ad-hoc modification of their code. These features overcome the major problem of the OpenDaylight Helium which required already deployed applications to be substantially rewritten with respect to their northbound interface. Furthermore, the software architecture we present in this paper is based on the publish-subscribe messaging pattern and, in particular, the proposed approach can be applied to any existing SDNApp by just patching it with few statements.

The problem to handle the run-time state of an application is also tackled in other fields different from SDN. In fact, some target-specific usage of the run-time application state can be seen in works such as [13,14,15], which investigate the problem of automatic Virtual Network Function (VNF) scaling and/or creating (either hot or cold) standby copies of a given network service instance. More specifically, with respect to the problem of handling the state, Pico [14] proposes an approach to move and/or duplicate portions of state among replicas; state is managed in a flow-centric way through FreeFlow [13], which models the state of a VNF as a flow table (where each line is usually identified by the TCP/IP five-tuple) and requires a per-VNF agent able to *get*, *put* and *migrate* flows. OpenNF [15] defines a northbound interface oriented to the use case of moving state between multiple instances. Although in this work authors mention the problem of VNFs, that do not provide APIs to access their run-time state, their focus is on possible race conditions when migrating (pieces of) state among different instances of the same VNF. All the works listed above address the problem of moving the state on new VNFs instances that are *analogous* to the original one, omitting the case in which the state has to be shared among applications that feature different purposes. Moreover, these works are oriented to the particular case of a flow-based state (that is typical in middle-boxes), i.e., the state can always be modeled

as a table where a row (or a set of) concerns a precise traffic flow. Instead, our paper addresses a more general problem, as we provide an approach to model any state of the application, not only flow-based states. Furthermore, the above approaches are limited to the applications they had in mind and for the purposes of their services (scaling in/out, hot/cold standby replicas), without considering the necessity to offer state handling capabilities to generic SDNApps and for arbitrary ServiceApps. Finally, we focus on SDN applications that, being software bundles running *within* the environment an SDN controller (e.g., exploiting the OSGi technology), they may have implementation constraints that prevent from trivially exposing the desired configuration interface through ad-hoc agents.

In the area of service orchestration, also including network state monitoring and control, a number of research works are devoted to create integrated workflows across application and network layers. In [16], a network-aware service composition and delivery solution is presented, which leverages network function virtualization mechanisms and programmable traffic steering capabilities from SDN to generate service chains according to user requirements and dynamic context. More specifically, authors focus on the architectural aspects in line with Service-Oriented Architecture to mainly address multi-domain requirements while leveraging intent-based SDN abstractions to steer the traffic along service chains. In this paper a different target is addressed aiming at the generalized use of SDN application state from application layers to build composite workflows in agnostic way. Another related research work has been carried out within the EU H2020 SELFNET project [17] focusing on an autonomic network management framework based on the Self-Organized Networks (SON) paradigm to achieve self-organizing capabilities in managing network infrastructures, thereby significantly reducing operational costs and improving user experience. However, this project pursues this goal with a different approach compared to what we propose in this paper. In fact, it proposes the deployment of a set of sensors with the aim of dynamically identifying possible issues that are then solved by a set of actuators. While the above architecture fits nicely with the final goal of keeping QoS under control, it may not be suitable for a broader scope, such as to enable generic services and applications to communicate and share arbitrary information as this work does. Indeed, this paper presents a first, partial implementation of a possible 5G Operating System (5GOS) is presented in [18], which extends to a distributed network infrastructure the concept of "everything as a service" presented in [19]. In this direction, this paper provides a detailed proposal of such architecture and a first set of answers to the problems that emerged from its implementation, such as the necessity to enable existing applications to be part of the new service-oriented architecture.

Ultimately, the analysis of the state of the art reveals a lack of solutions to enable SDNApps to effectively share their internal run-time state and to communicate with upper-layer functionalities (i.e., network services or application-layer orchestrators) and, thus, to be part of composite and dynamically-established workflows. In this direction, this work proposes a software architecture with an abstraction layer to enable the real-time exposition and modification of SDNApps internal variables, where data are exported according to high-level model-based structures. The mapping from the high-level data model to the actual data representation within the SDNApps is enabled by a suite of algorithms that are generic and independent from the specific source code of the SDNApp, thus avoiding undesired and invasive modifications to existing applications.

3 | USE CASES

The capability to collect the run-time state of a running SDNApps and modify it transparently at run-time (either by issuing the proper configuration commands or by crafting its state, if needed) brings important advantages in several use cases. Some possible examples are provided below.

SDN-based Intrusion Prevention System (IPS). In this first use case (Figure 1), an IPS (acting as an ServiceApp) coordinates a workflow involving the current status of an SDN-based Intrusion Detection System (IDS) and/or additional information collected by other SDNApps (e.g., network monitors) to feed an SDN-based firewall with the proper policies to block incoming attacks or suspicious traffic patterns. For instance, the IPS may need to observe the run-time state of the IDS (e.g., the internal variables that keep the list of policy violations) and of a network monitor to become aware of any anomalous state change and thus to detect anomalous traffic patterns (e.g., spikes in non-working hours). As soon as one among a set of particular changes occurs (i.e., a threat is detected by the IDS or an anomalous traffic is observed through the network monitor), the IPS service triggers the creation of the required firewall rules that are then injected in the target SDNApp to protect the network infrastructure.

Migration Service. Another common use case is the migration of a running SDNApp, e.g., a NAT, to a new location, for instance, to follow user movements [20]. In this case, a Migration Service (i.e., ServiceApp) may need to acquire the run-time state of the existing SDNApp instance to properly bootstrap the new instance with the current/correct state, which in case of a NAT, consists in the network address translation table. This technique avoids to copy the entire memory of the SDNApp such as in the traditional Virtual Machine migration approach [21] and it exploits the idea of moving only semantically rich information as

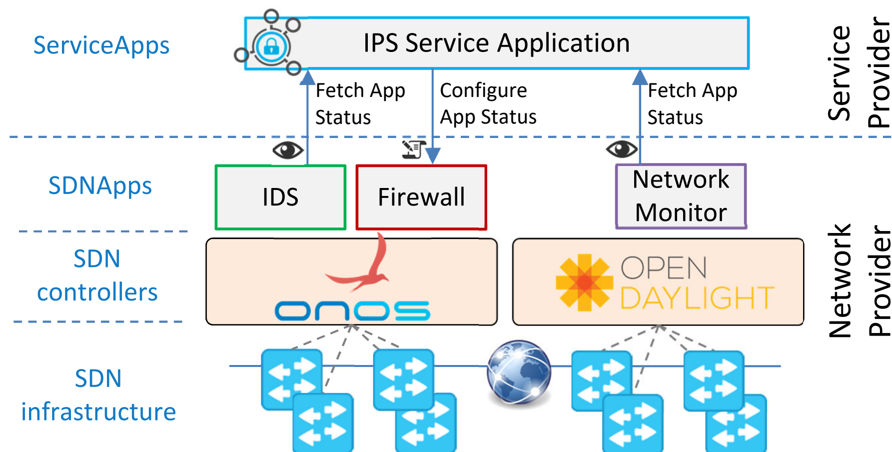

FIGURE 1 Use case example: an Intrusion Prevention System service (*ServiceApp*) requiring the access to the run-time state of multiple SDN applications (*SDNApps*).

proposed in[15]. In this respect, an application-agnostic Migration Service, which can operate on whatever application internal data structure is used while copying only semantically relevant information, is possible only thanks to the capability of inspecting dynamically the run-time state of the SDNApp under consideration.

Service Orchestrators. This use case refers to the deployment and lifecycle management operations carried out by a Network Service Orchestrator (e.g., ETSI NFV MANO[22]) to take care of the coordinated set-up and management of a set of VNFs that jointly realize a Network Service, e.g., Virtual Evolved Packet Core. As for network service deployment the Network Service Orchestrator leverages infrastructure managers (e.g., WAN Infrastructure Managers acting as SDNApp) to perform resource configurations (e.g., virtual links set-up connecting VNFs). During service lifecycle, the Network Service Orchestrator may leverage data analytics tools to derive given performance indicators to optimize Network Service management operations or to prevent SLA violations. Indeed, data analytics tools run applications (i.e., SDNApps) that collect monitoring data from the underlying cloud (e.g., VNFs and virtual links) or network resources (e.g., OpenFlow switches and links) and then aggregate them to derive consolidated performance data indicators (e.g., user data flow throughput)[23]. These data may be used by components of a Network Service Orchestrator acting as ServiceApp (e.g., NFV Orchestrator or the SLA Manager in the ETSI MANO) to maintain an up-to-date view of resource usage and/or performance offered by the underlying infrastructure to promptly react in case of service degradations due to concurrent usage of resources from many different Network Services or due to any adverse event (e.g., service outages, network congestions, improper usage of VNFs). In this context, application-agnostic data models may foster composite and comprehensive service lifecycle workflows where both cloud and network resource status information are considered to offer high-quality services and highly effective resource utilization.

4 | ARCHITECTURE

This section presents the software architecture we designed to enable ServiceApps to read and/or modify the run-time state of SDNApps, while being agnostic to the implementation-specific data structures they use internally. The overall architecture is depicted in Figure 2 . According to the figure, ServiceApps are envisioned to run on top of SDNApps and/or custom SDN controllers while relying on the following main components: *(i)* a YANG-based *data model* associated with each SDNApp and describing SDNApps run-time state through high-level and application-independent structures (Section 4.1); *(ii)* a *communication infrastructure* enabling interactions between the ServiceApp and the SDNApps where exchanged information and the communication API depend on the YANG data models associated with SDNApps (Section 4.2); and *(iii)* a *To Yang (ToY) Agent* running in each SDNApp and enabling ServiceApps to access and refer their SDNApps run-time state according to the specified implementation-independent YANG-based data model (Section 4.3).The remainder of this section details the above components.

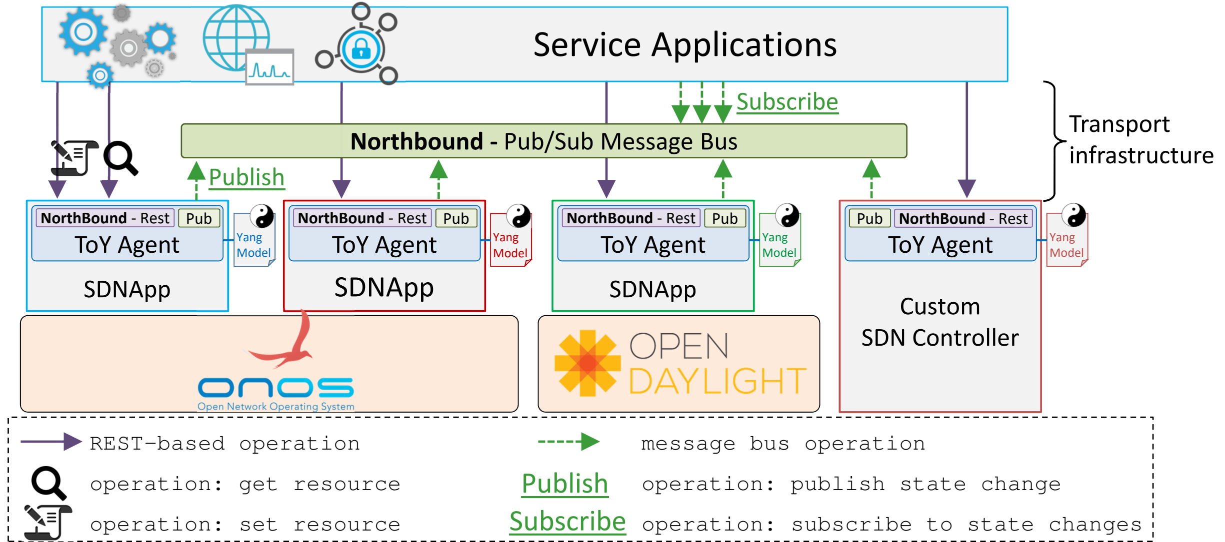

FIGURE 2 Overall software architecture. ServiceApps (on top) relies on a northbound interface (composed by the Pub/Sub message bus and on the Rest-based channel) to perform `get`, `set` and `subscribe` operations on the run-time state of SDNApps. The latter are accessible through this northbound thanks to the YANG-based mapping performed by ToY Agents.

4.1 | A data model for SDNApps

To avoid ServiceApps to be aware of each SDNApp internal data structures and proprietary APIs to access these data, we define an high level *model-based API*. ServiceApps can rely on this model to have uniform access to run-time state of SDNApps, regardless of their implementation. This interface provides data through common syntax and modular structures defined by formal models written using the YANG modeling language[7]. Through the data model abstraction, an SDNApp developer can arbitrarily decide the portion of the internal state that has to be exposed to external services, e.g. omitting critical data such as authentication details. Each SDNApp is associated to its own data model, which is used to describe: *(i)* which data is exposed by the SDNApp, *(ii)* how to access such data with the desired granularity and *(iii)* how exposed data is structured. The way we represent these information within the data model is defined below. An example of data model, associated with a NAT application, is shown on the left of Figure 3 .

Each node (e.g., module, container, leaf) of the data model represents a *resource*, which is uniquely identified through an identifier called *YANG path*. The YANG path associated with each resource is generated from the data model using rules derived from RESTCONF[6] and shown in Table 1 . Particularly, rows #1 to #7 of the table represent the basic rules, which can be combined as shown in row #8 to specify any element of the data model. According to the table, the YANG path is an URI built as an ordered list of *YANG labels*, namely of the names of all the YANG elements that enclose that resource, from the most external to the resource itself, separated by the character ''/''. For instance, in the data model provided in Figure 3 , the container `private` is enclosed in the container `interfaces`, which is in turn enclosed in the module `nat`; therefore, the resource ''`private`'' is identified by the YANG path `/nat/interface/private`. YANG paths may enclose predefined portions (e.g., `/nat/nat-session`) and parametric portions (e.g., `/{uuid}`). The latter are needed since `lists` in the data model produce multiple resources with the same structure. Each of these resources is identified by a YANG path that includes a *parametric label* characterized by braces (Table 1 , rows #5 and #7); this label corresponds to the value of the `key` leaf in the list node description, as it is unique. In other words, to identify a precise element of the list, the parametric label must be replaced with the value of the `key` leaf in that list item. As an example, referring again to the data model of Figure 3 , each element of the `nat-session` is identified by a YANG path in the form `/nat/nat-session/{uuid}`; thus, to identify a specific element of the `nat-session`, the parametric label `{uuid}` must be replaced by the instance value of the `uuid` field of the desired element, e.g., `0x26` as in Table 1 .

Each element is associated with a `config` statement that indicates whether a resource is writable (`config true`) or read-only (`config false`). For instance, the resource `name` in Figure 3 , which in this case represents the name of a network interface, cannot be modified by an external entity (i.e., by a ServiceApp). Moreover, inspired by recent works in IETF[24], we associate a resource with a new `advertise` statement (defined as a YANG extension) indicating in which situations the SDNApp advertises the current resource value to the outside world: `onchange`, advertised any time the value changes; `onthreshold`, advertised

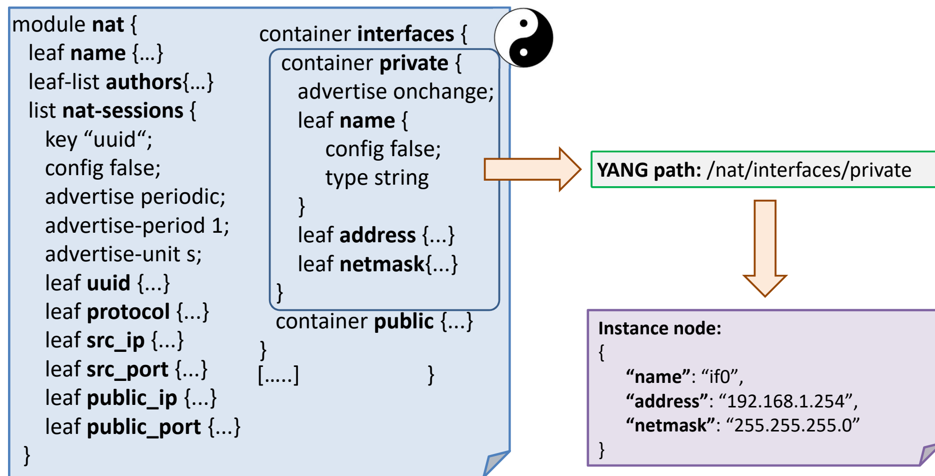

FIGURE 3 On the left, a YANG data model for the NAT application. On the right, an example of YANG path (the one of the *"private interface"* container highlighted in the model) and an instance node showing a possible run-time value.

TABLE 1 Deriving the YANG path from the YANG data-model.

#	YANG element	YANG path	Example from Figure 3
1	Entire data-model	/module-name	/nat
2	Container	[...]/container-name	[...]/interfaces
3	Leaf(/Anydata)	[...]/leaf-name	[...]/name
4	Entire leaf-list	[...]/leaf-list name	[...]/authors
5	Element in a leaf-list	[...]/leaf-list/{value}	[...]/authors/Castellano
6	Entire list	[...]/list-name	[...]/nat-session
7	Element in a list	[...]/list-name/{key-field}	[...]/nat-session/0xa26
8	Generic element	/element1/.../elementN	/nat/interfaces
			/nat/interfaces/private
			/nat/nat-session
			/nat/nat-session/0x26
			/nat/nat-session/0x26/dst_ip

just when the value exceeds a specific threshold; `periodic`, advertised with a certain frequency, regardless of whether it has changed or not (this is the case of the `nat-session` resource shown in Figure 3); `ondemand`, which is never advertised, hence the value must be explicitly requested by the caller. Additionally, each resource can be accessed `ondemand` (which represents the default setting) regardless the value of `advertise`.

High level ServiceApps use YANG paths to subscribe/get access to SDNApps resources. As an example, if a ServiceApp wants to modify the `address` and the `netmask` of the private interface of the NAT, it may use the YANG path associated with the resource ''`private`'' (shown in the right of Figure 3), thus selecting the specific resource (i.e., the private interface) and provides to the SDNApp the new data as a JSON. The JSON structure is derived from the data model, as shown on the bottom right of Figure 3 . In the remainder of this paper, a JSON formatted according to (a piece of) the YANG data model is called *instance node*. It may contain either the new values to be assigned, in the SDNApp, to the resource identified by the YANG path (i.e., a new configuration, as in the example above), or the current value of such a resource in the application itself (e.g., the run-time state exported by the SDNApp to the ServiceApp).

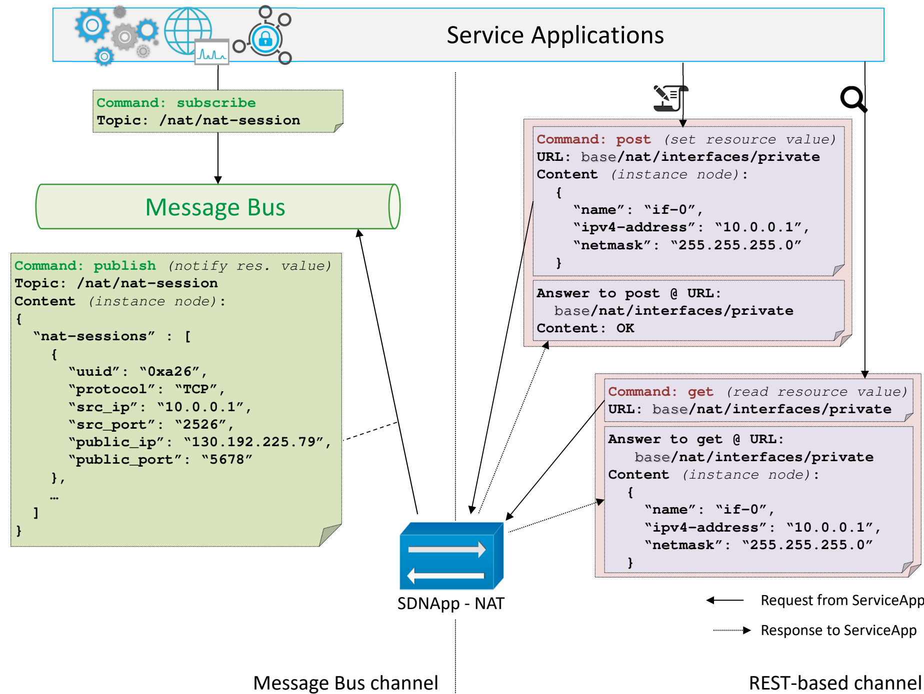

FIGURE 4 Example of messages sent on the message bus (in the left) and on the REST-based channel (in the right), defined according to the data model of Figure 3 .

4.2 | Communication infrastructure

To enable SDNApps to receive configurations from ServiceApps and to expose their own run-time state in accordance with the data model, we define a logical *communication infrastructure* that offers two types of connections: a direct (REST-based) communication channel and a shared message bus based on the publisher/subscriber (pub/sub) paradigm.

Direct channel. This channel, which relies on a set of REST APIs, enables a point-to-point connection between a client and a server. Particularly, according to Figure 2 , ServiceApps act as clients, while the ToY Agents running in the SDNApps play the role of servers. The direct channel is used by a ServiceApp to send `get()` and `set()` commands to the ToY Agent in the proper SDNApp, to explicitly retrieve or add/change resources described in the data model associated with the SDNApp itself.

Shared message bus. This channel is instead shared between entities that can `publish()` messages that are associated with *topics*, as well as they can `subscribe()` to topics; the message bus then brings a message to all the subscribers for the topic associated with the message itself. In our proposal, ServiceApps act as subscribers, while the ToY Agents are the publishers. Particularly, they publish notifications on updates of resources marked as `onchange`, `onthreshold` or `periodically` in the data model (e.g., notifications may enclose the new value, the fact the resource has been deleted, and more); each message is published on a topic corresponding to the YANG path associated with the resource related to the message itself, as shown in the example in the left of Figure 4 . Since subscriptions are managed in a hierarchical way, when a ServiceApp subscribes on a given resource, it will also receive updates about nested resources (e.g., a subscription to the resource identified by the YANG path `/nat/interfaces/private` of Figure 3 will also receive updates for any change in resource `/nat/interfaces/private/address`).

As shown in Figure 4 , in both channels the URI to be used by a ServiceApp to access a specific resource is dynamically derived from the YANG path associated with the resource of interest, becoming either a *URL* in the direct channel, or *topic* in the shared message bus. Moreover, the data provided (in case of `set()` command) or retrieved (in case of `get()` or `subscribe()` command) by the ServiceApp is organized according to the YANG data model itself.

4.3 | To Yang Agent

The *To Yang (ToY) Agent* is the core component of the proposed architecture. It is imported by each SDNApp as an external module, in order to *(i)* handle the mapping of (high level) resources described in the YANG data model on (low level) run-time application variables and *(ii)* manage the interaction between the SDNApp and the communication infrastructure. Using the

terminology introduced in Section 4.1, the ToY Agent takes care of setting the configuration and fetching the run-time state of the SDNApp, providing to the ServiceApps an interface based on YANG paths and YANG-modeled instance nodes (as shown in Figure 4). We designed the ToY Agent to be agnostic with regard to the different SDNApps, to operate without introducing invasive modifications on them.

The remainder of this section focuses on how this agnostic module works and is structured as follows. First we provide a description of structures and mechanisms used by the ToY Agent to map run-time application state into YANG-modeled data; then we present the architecture of the ToY Agent, hence detailing all the components that implement the above mechanisms. Finally, we describe the steps required to extend an existing SDNApp with our ToY Agent, also providing an overview of the application bootstrap workflow.

4.3.1 | Path Mapping Table and Mapping Mechanism

While the YANG data model (together with all the derived YANG resources) provides a logical, implementation-independent view of the application, each SDNApp has its own internal data structure that may be only loosely related to the logical view. Figure 5 shows a possible internal structure of a NAT application that can be modeled through the YANG in Figure 3 . By comparing the two structures, we can easily notice, for instance, that some common fields have a different name (e.g., the `nat-session` list becomes `sessionList`) or a different "location" (e.g., the leaf `public_ip` is no longer a value inside the session list, but is stored in the `publicAddress` variable). In general, there may be arbitrary differences.

Then, since the ToY Agent communicates with ServiceApps through the high-level YANG abstraction (i.e., YANG paths and instance nodes), it needs a mechanism to understand, each time, which variables in the SDNApp code must be accessed to accomplish a request or publish data updates. For example, if a ServiceApp performs a `get()` request regarding the YANG path `/nat/nat-session/0x26/public_ip`, the ToY Agent needs to know that this value is stored in a certain variable, e.g., in the `publicAddress` attribute of `natData` object in Figure 5 . Instead, if the request refers to the YANG path `/nat/nat-session/0x26`, it needs to know where all needed nested values (`protocol`, `src_ip`, `src_port`, etc. in Figure 3) are stored within the application code.

This information is different from case to case, hence it cannot be embedded in the ToY Agent itself, since we want this module to be agnostic with respect to both the SDNApp implementation and the data model. For this reason we defined a data structure called *Path Mapping Table (PMT)*, that describes each association between a YANG resource and the corresponding variable in the application source code. The ToY Agent, once imported in a particular SDNApp, accesses this structure as an external information and exploits it to learn, at run-time, how variables of that SDNApp are structured and how they are associated to the high-level YANG resources.

Particularly, as shown in Table 2 , the PMT maps the YANG paths derived from the data model to the proper *object path* within the application *object tree*. We define *object tree* the structure of all the variables (i.e., all objects and their attributes) in the application source code. For instance, the box in the center of Figure 5 reports the object tree corresponding to the source code of the box on the left. It is worth noting that, unlike the data model, that can be associated with all the SDNApps implementing the same functionality (e.g., NAT), the object tree is specific for a given implementation, since it depends on the source code of the application itself. The first element of the object tree is called *root object*, and corresponds to the most external object/variable that we have access to (e.g., the object `mainApp` in Figure 5). Each element of the object tree (i.e., an application variable) is identified by an *object path* that begins with the root (character "/"), followed by an ordered list of attributes built as already described in Section 4.1 in case of YANG path. The right side of Figure 5 shows the object paths corresponding to the object tree depicted in the central box of the figure.

Since an SDNApp may feature a source code structure that is very different from the high-level view presented in the YANG model, this originated the necessity of supporting a complex syntax for the PMT records. In the following we present some of the mapping cases we encountered in our work, using them to intuitively describe how the PMT and its records are formatted.

The most common mapping case is for application variables that are associated to YANG resources having simply a different name (e.g., `sessionList` is mapped to the `nat-session` resource). An other case is when application variables do not have any mapping to a resource in the data model. An example is given by the variable `mac` in the object tree show in Figure 5 , which does not have any correspondence with a resource in the data model of Figure 3 (in fact, no entry involving such a variable can be found in the PMT of Table 2). In some cases, resources that are *inside* the same parent element in the data model may correspond to variables located in different objects in the object tree, or vice versa. For example, the resources `nat-session` and `interfaces` (rows #2 and #6 in Table 2) are children of the `nat` node in the data model of Figure 3 , while their corresponding objects, namely `sessionList` and `ifs`, are attributes of two different objects in the object tree of Figure 5 . All the previous cases leads to the necessity of a record for any possible YANG path within the PMT.

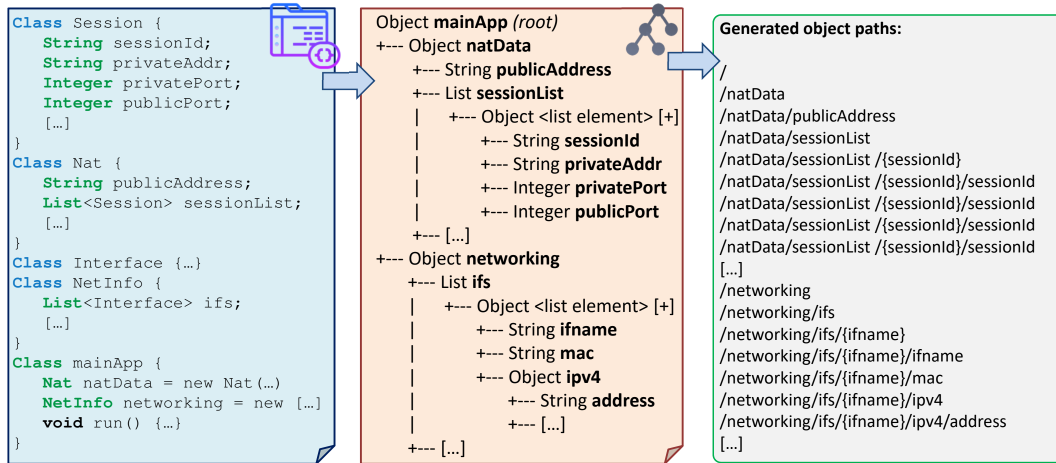

FIGURE 5 From run-time variables (in the left) to object paths (in the right).

A frequent mapping case is when items of a YANG list have a direct correspondence with items of a collection object (e.g., an array, a list, a set or a map). An instance of this mapping is realized with rows #3, #4 and #5 of Table 2 . To distinguish resources in `/nat/nat-session`, the parametric label `{uuid}` is used in the YANG path, since the `uuid` leaf is the `key` for the items of the `nat-session` list according with the YANG model of Figure 3 . To map each resource in this list with a specific object of `sessionList` in the object tree, the attribute `sessionId` is used as parametric label of the object path, since its value corresponds to the YANG `key` field (although the application source code does not indicate in any way that such a variable is a key in the list).

Finally, we encountered some occasional cases where a *non-list* resource in the data model is mapped to a specific item of a collection (e.g., a list) in the object tree (or vice versa). This is the case of the `private` resource in the data model of Figure 3 , which corresponds to a specific element of the list `ifs` in the object tree of Figure 5 ; to describe this mapping we need rows #7 and #8 of Table 2 . Particularly, in row #8 the parametric label in the object path is replaced with the specific value "`if0`", the one of the object corresponding to the resource identified by the YANG path `/nat/interfaces/private`. Moreover, row #7 maps a dummy resource (called `noresource` in Table 2) within YANG data model, into a generic element (identified by the parametric label) of the collection in the object tree. This way, the ToY Agent knows that the parametric label of `ifs` is `ifname` and, e.g., `if0` is the instance value of the corresponding variable `ifname` in the application code.

To make the ToY Agent agnostic with respect to the data model and independent from the particular SDNApp internal variables structure, the PMT must be coupled with a set of *Mapping Algorithms* that exploit this data structure and enable both read and write access to the configuration and run-time state starting from its high level representation. Particularly, given a PMT, a YANG data model and the associated SDNApp, the mapping algorithms both retrieve and set the SDNApp variables, converting YANG-modeled structures into application objects (and vice versa). As we do not want to introduce overhead on the SDNApps code by implementing an ad-hoc mapping for each of them, those algorithms are application-agnostic and just rely on the PMT and on the data model. Our mapping algorithms, described in detail in Section 5, provide the following elementary operations, which are implemented by the ToY Agent: *(i)* resolve the association between a YANG path and an object path and vice versa, through a lookup on the PMT; *(ii)* fetch an object starting from its object path; *(iii)* get the value of a given object and write it into the corresponding instance node (e.g., JSON), according to the YANG data model and the PMT (the YANG data model is used to derive the structure of the instance node, while the PMT shows how to map application variables into YANG resources); *(iv)* set the value of application objects starting from the corresponding instance node (e.g., JSON), according to the PMT.

4.3.2 | ToY Agent Architecture

Figure 6 details the architecture of the ToY Agent. According to the figure, such a component is linked to the original SDNApp and must be configured with both the data model and the PMT associated with the SDNApp itself. The ToY Agent architecture has been designed to work on top of any SDNApp implementation; it is described below through a top-down approach.

The northbound interface of the ToY Agent consists of two YANG-based modules that connect to the communication infrastructure, namely the **YANG Based Publisher** and the **YANG based REST API**, which implement respectively the shared message bus publisher and the HTTP REST server described in Section 4.2.

TABLE 2 Excerpt of PMT, referred to the data model of Figure 3 and to the object tree shown in Figure 5 .

#	YANG path	Object path
1	/nat	/
2	/nat/nat-sessions	/natData/sessionList
3	/nat/nat-sessions/{uuid}	/natData/sessionList/{sessionId}
4	/nat/nat-sessions/{uuid}/src_ip	/natData/sessionList/{sessionId}/srcAddr
5	/nat/nat-sessions/{uuid}/public_ip	/natData/publicAddress
6	/nat/interfaces	/networking/ifs
7	/nat/interfaces/{noresource}	/networking/ifs/{ifname}
8	/nat/interfaces/private	/networking/ifs/if0
9	/nat/interfaces/private/address	/networking/ifs/if0/ipv4/address

The **On-Demand Access Handler** manages `get()` and `set()` commands coming from the ServiceApps through the REST API, relying on the `get_node` and `set_node` core primitives provided by the Mapping Library. Instead, the **Object Listener** implements the monitoring procedure needed to recognize and notify updates on the SDNApp state. It performs the following operations: *(i)* identify, through the data model, the YANG path of resources that should be exported (and when to export them); *(ii)* rely on the Mapping Library to convert these YANG paths into objects (`map_path` and $\texttt{fetch}_o$ primitives); *(iii)* build the corresponding instance node (using `get_node` primitive) and export it through the YANG Based Publisher on the message bus whenever the condition specified in the YANG data model for that resource (`onchange`, `onthreshold` or `periodically`) is satisfied. Nodes that must be exported `onchange/onthreshold` are inspected periodically through a polling mechanism. Performance is analyzed and discussed in Section 6.

The **Mapping Library** is the core module of the ToY Agent, as it represents the bridge between the high level YANG interface and the low level object-based structure of application variables. It can access to both the PMT and the YANG data model, and relies on the Reflective Library to implement the mapping procedures introduced in 4.3.1, thus providing the following primitives: `map_path`, converts a YANG path to an object path through the PMT; $\texttt{fetch}_o$, returns an object starting from its object path; `get_node`, returns the instance node (e.g., a JSON structure) of the requested YANG resource, reading values from the application variables; `set_node`, modifies the application variables based on the input YANG resource instance. In Section 5 we provide a detailed description of four algorithms that implement these procedures.

The **Reflective Library** is the component that directly interacts with the original application and is in charge of actually accessing to application variables to perform the classical CRUD (Create, Read, Update, Delete) operations. Particularly, it allows to examine and modify application variables at run-time through the use of *reflection*[25] (a.k.a. *introspection*), a feature that is required since the ToY Agent knows attribute names just during its execution, e.g., when a particular request is performed, deriving them from the object path retrieved in the PMT[2]. This approach enables the Mapping Library of the ToY Agent to operate without having an a-priori knowledge of the SDNApp internal structure. Through the run-time introspection of application objects, the Reflective Library provides an interface built with the following primitives (also shown in Figure 6): `get_attribute()`, `get_by_attribute()`, `set_attribute()` and `append_new()`. These are the only procedures that have direct access to the original SDNApp execution environment (Figure 6). For example, the ToY Agent may receive a `get()` request for the resource `/nat/nat-sessions/0x26/public_ip`; then, through the PMT it learns that it has to access to the object identified by the object path `/natData/publicAddress` (Table 2 , row #5). At last, the primitive `get_attribute` is used to actually access this object. This function accepts an object reference (in this case, object `natData`) and the name of an attribute of that object (in this case, the string `‘‘publicAddress’’`), and returns the object reference to that attribute. In Section 5 all reflective functions are exhaustively defined.

It is worth noting that, since the ToY Agent directly accesses to application variables, race conditions may derive from possible situations in which the ToY Agent sets a variable value on which the SDNApp itself is working on it (or even read a temporarily inconsistent piece of state). To avoid this problem, we assume that SDNApps are thread safe, i.e., that critical regions are properly protected so that the ToY Agent is prevented from modifying variables during critical operations or read inconsistent data.

[2] We validated our proposal through Java-based SDNApps, then we exploited the *JAVA Reflection API* to access and manipulate run-time objects.

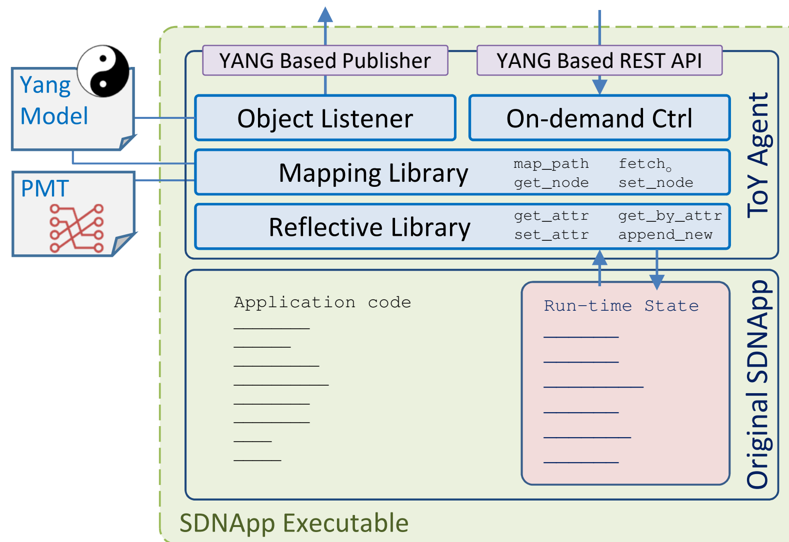

FIGURE 6 Architecture of the ToY Agent.

```
// Import the ''ToY Agent'' module into this application
import sdnapp.ToYAgent;

// This is the code that has to be added in the ''bootstrap method'' of the SDNApp
// Instantiate the ToYAgent, providing it a reference to the root object, the PMT and the data model
public ToYAgent tya = new ToYAgent(this, pmt, yangModel);
// Start the ToYAgent and let's forget about it
tya.start();
```

FIGURE 7 Steps required to include the ToY Agent in an existing application.

4.3.3 | Embedding Mechanism and Operating Workflow

Mapping mechanism used by the ToY Agent, as well as its architecture itself, have been designed to keep the ToY Agent independent from the particular SDNApp (from both its implementation and its data model). This way, we reduce the overhead of the SDNApp developer, which must not write an ad hoc ToY Agent for each new application, thus allowing the inclusion of any SDNApp in our configuration framework almost without introducing any changes in the application code.

Figure 7 shows the steps that an SDNApp developer is required to carry out to make its software compatible with our framework. Basically, he/she is just expected to: *(i)* include the ToY Agent module in the SDNApp; *(ii)* initialize the ToY Agent at the application bootstrapping, by providing it a reference to the root object (the most external object that can be accessed from the outside), the Path Mapping Table and the YANG data model; *(iii)* start the ToY Agent[3].

At the bootstrapping, the ToY Agent performs the following preliminary operations (which are hidden to the SDNApp developer): *(i)* notifies the SDNApp identifier and the YANG data model through the message bus, so that external ServiceApps are enabled to exploit the new application; *(ii)* analyzes the data model to identify the way in which the value of each resource defined in the model itself must be exported (i.e., `onchange`, `onthreshold`, `periodically` or `ondemand`); *(iii)* reads the PMT to understand how to map YANG resources into application objects; *(iv)* starts an *Object Listener* (Figure 6) that monitors and exports objects associated with resources to be notified `onchange`, `onthreshold`, whenever the conditions defined in the data model are satisfied, and `periodic`, whenever the appropriate timer expires; *(v)* enables the REST-based communication channel to allow any ServiceApp to explicitly access the SDNApp run-time state.

[3]We based our prototype on SDNApps Java-based OSGi bundles running on top of ONOS; then, we implemented the ToY Agent as a library included in the *Main* class, instantiated during the SDNApp activation and initialized with the reference to that class instance as root object.

5 | MAPPING ALGORITHMS

In this section, we detail the algorithms used to map the object-oriented internal state of an SDNApp into the high level YANG-modeled state, and vice versa. We identified four elementary operations (i.e., those introduced in Section 4.3.1), implemented by as many Mapping Algorithms: `map_path()`, $\texttt{fetch}_o\texttt{()}$, `get_node()` and `set_node()`. These algorithms are independent from both the SDNApp and the YANG model, thus enabling implementation-independent access to run-time variables of arbitrary SDNApps.

The operations implemented by the algorithms presented in this section constitute the Mapping Library, the core module of the ToY Agent presented in Section 4.3.2.

In the first part of this section we present the notation that will be used in the rest of the section to give a complete and formal description of our mapping algorithms.

5.1 | Structures and Function Notations

Table 3 presents formal notation for data structures that will be considered during the description of the mapping algorithms. For the sake of cleariness, besides its formal description, every notation is accompaigned with an example that illustrates it within the NAT use case shown in Figure 3 . Additionally, mapping algorithms rely on the functions below.

Label Functions. Given a resource $\boldsymbol{\upsilon} \in Y$, the function $\texttt{label}_y$ returns the label $\lambda_y \in L_y$ of that resource, i.e., the name of the corresponding YANG node in the model. Given an object $\boldsymbol{o} \in O$, the function $\texttt{label}_o$ returns the label $\lambda_o \in L_o$, i.e., the name of that variable/attribute in the run-time application environment.

Parametric Label Functions. Since, as described in Section 4.1, resources in a list are identified by a YANG path that ends with a parametric label (namely, the name of the `key` field in the model), we use function $\texttt{param_label}_y : L_y^n \rightarrow L_y$ to know the parametric label of a list resource. Analogously, we use function $\texttt{param_label}_o : L_o^n \rightarrow L_o$ for collection objects.

For instance, in Figure 3 , $\texttt{param_label}_y$(“`/nat/nat-sessions`”) = “`hash`”. Note that this function can be easily implemented through an inspection on the PMT.

Instance Node Helpers. Given a key-value instance node $\boldsymbol{v} \in \mathcal{N}_{kv}$ and a label $\lambda_y \in L_y$, we use *(i)* the function $\texttt{get_element}(\boldsymbol{v}, \lambda_y)$ to get the node $\boldsymbol{v}_i \in \boldsymbol{v}$, and *(ii)* the function $\texttt{put_element}(\boldsymbol{v}, \lambda_y, \boldsymbol{x})$ to set $\boldsymbol{v}_i$ to an input value $\boldsymbol{x} \in \mathcal{N}$, where $\boldsymbol{v}_i$ is the node nested in $\boldsymbol{v}$ with key $\boldsymbol{\lambda_y}$ (in symbols $\texttt{key}(\boldsymbol{v}_i) = \boldsymbol{\lambda_y}$). Given a list instance node $\boldsymbol{v} \in \mathcal{N}_l$, we use *(iii)* function $\texttt{append_element}(\boldsymbol{v}, \boldsymbol{v}_i)$ to add the instance node $\boldsymbol{v}_i \in \mathcal{N}$ to the list $\boldsymbol{v}$.
For example, lets $\boldsymbol{v}$ be the instance node in Figure 3 ; then, $\texttt{get_element}(\boldsymbol{v}, \text{“}\texttt{name}\text{”}) = \text{“}\texttt{if0}\text{”}$.

Reflective Library Functions. As described in Section 4.3.2, Mapping Library relies on the Reflective Library. It provides some functions that, given an object $\boldsymbol{o} \in O$, allow to read or modify the value of any $\boldsymbol{o}_i \in \boldsymbol{o}$ (i.e., any of its child objects). On ordinary objects, i.e., $\boldsymbol{o} \in O_o$, $\boldsymbol{o}_i$ is an attribute of $\boldsymbol{o}$; thus, the library provides *(i)* function $\texttt{get_attribute}(\boldsymbol{o}, \lambda_{o_i})$, that returns attribute named $\lambda_{o_i} \in L_o$ of object $\boldsymbol{o} \in O_o$, and *(ii)* function $\texttt{set_attribute}(\boldsymbol{o}, \lambda_{o_i}, \boldsymbol{x})$, that sets attribute named $\lambda_{o_i} \in L_o$ of object $\boldsymbol{o} \in O_o$ to value $\boldsymbol{x} \in O$. On collection objects, i.e., $\boldsymbol{o} \in O_l$, $\boldsymbol{o}_i$ is an item of the collection $\boldsymbol{o}$; thus, the library provides *(iii)* function $\texttt{get_by_attribute}(\boldsymbol{o}, \lambda_{o_i}, x)$, that returns the first item of collection object $\boldsymbol{o} \in O_l$ whose attribute named $\lambda_{o_i} \in L_o$ is equal to value $\boldsymbol{x} \in O$, and *(iv)* function $\texttt{append_new}(\boldsymbol{o}, \lambda_{o_i}, x)$, that appends to collection object $\boldsymbol{o} \in O_l$ a new item, whose attribute named $\lambda_{o_i} \in L_o$ is initialized to value $\boldsymbol{x} \in O$.
For example, let $\boldsymbol{o}$ be the object `natData` of Figure 5 ; then a call to $\texttt{get_attribute}(\boldsymbol{o}, \text{“}\texttt{public_Address}\text{”})$ returns the value stored in `natData.publicAddress`.

Fetch Resource Function. Given a YANG path $\boldsymbol{\rho}_y \in L_o^n$, to retrieve the corresponding resource $\boldsymbol{\upsilon} \in Y$ we use the function $\texttt{fetch}_y : L_o^n \rightarrow Y$.
Note that $\texttt{fetch}_y$ can be easily implemented through a simple lookup in the YANG model.

5.2 | Algorithms Description

In this section, we describe the algorithms that perform the bidirectional map between run-time application variables and resources defined by the YANG data model. They constitute the **Mapping Library** module of the ToY Agent (Section 4.3.2). Since mapping algorithms exclusively rely on the content of the PMT and the YANG model, they enable the ToY Agent to be agnostic regarding the particular SDNApp implementation. The description of all mapping algorithms is provided below.
Map Path Algorithm. This algorithm establishes a link between YANG resources and application objects, since it converts a YANG path $\boldsymbol{\rho}_y \in L_y^n$ to the corresponding object path $\boldsymbol{\rho}_o \in L_o^m$, according with information stored into the Path Mapping

TABLE 3 Data structures formal notation used in the description of the algorithms.

Notation	Description	Example
$Y = Y_{co} \cup Y_{li} \cup Y_{le} \cup Y_{ll}$	Set of all YANG nodes in the data model, i.e., modules and containers (Y_{co}), lists (Y_{li}), leafs (Y_{le}) and leaf-lists (Y_{ll})	`nat` and `interfaces` (Figure 3) belong to Y_{co}, while `nat-session` belongs to Y_{li}
$\boldsymbol{\upsilon} \in Y$	YANG resource, i.e., each node of a YANG model	In the YANG of Figure 3 , each element in bold represents a resource
$\boldsymbol{\upsilon}_i \in \boldsymbol{\upsilon}$	Resource $\boldsymbol{\upsilon}_i$ is a node directly nested in resource $\boldsymbol{\upsilon}$ (child resource)	Container `private` is a child resource of container `interfaces` (Figure 3)
$\boldsymbol{\upsilon}_{root} \in Y$	The root resource in Y	The `nat` module in Figure 3
L_y	Set of all YANG labels, i.e., the name of all YANG nodes in Y	For the YANG in Figure 3 , L_y={“`nat`”, “`name`”, “`authors`”, ...}
$\boldsymbol{\rho}_y \in L_y^n$	YANG path of a given resource $\boldsymbol{\upsilon} \in Y$	The YANG path of the resource highlighted in Figure 3 is `/nat/interfaces/private`
$O = O_e \cup O_l \cup O_o$	Set of all application variables, i.e., “elementary objects” (O_e), e.g., integers, strings, booleans, “collection objects” (O_l), e.g., list, maps, “ordinary objects” (O_o) e.g., instances of custom classes	In Figure 5 , `networking` belongs to O_o, `ifs` belongs to O_l and `ifname` to O_e
$\boldsymbol{o} \in O$	Application object, i.e., each variable/attribute in the SDNApp execution environment	Every Java object in Figure 5
$\boldsymbol{o}_i \in \boldsymbol{o}$	$\boldsymbol{o}_i$ is an attribute of $\boldsymbol{o}$ (child object)	In Figure 5 `publicAddress` is a child of the object `natData`
$\boldsymbol{o}_{root} \in O$	Root object in O	The root object of the object tree in Figure 5 is `mainApp`
L_o	Set of all object labels, i.e., the name of all run-time variables of an SDNApp	For the example in Figure 5 , L_o = {`natData`, `publicAddress`, `sessionList`, ...}
$\boldsymbol{\rho}_o \in L_o^m$	Object path of a given object $\boldsymbol{o} \in O$	The object path of the list “ifs” in Figure 5 is `/networking/ifs`
$\mathcal{N} = \mathcal{N}_e \cup \mathcal{N}_l \cup \mathcal{N}_{kv}$	Domain of all the possible instances (e.g., JSON) of resources described by a data model. They can be elementary nodes $\mathcal{N}_e$ (i.e., nodes without nested nodes), list nodes $\mathcal{N}_l$ and key-value nodes $\mathcal{N}_{kv}$	The instance of a container resource is a key-value node, while the instance of leaf resource is an elementary node
$\boldsymbol{\nu} \in \mathcal{N}$	Instance node of a resource $\boldsymbol{\upsilon} \in Y$, i.e., a piece of data that is structured as defined by the portion of YANG model associated with the resource $\boldsymbol{\upsilon}$	An example of (key-value) instance node is the JSON data in Figure 3
$\boldsymbol{\nu}_i \in \boldsymbol{\nu}$	Node $\boldsymbol{\nu}_i$ is directly nested in node $\boldsymbol{\nu}$ (child node)	In Figure 3 , node “`name`” is directly nested in the example instance node

Table. This path mapping is implemented by the function `map_path` : $L_o^m \rightarrow L_y^n$, whose algorithm is omitted since it is a trivial key-value lookup in the PMT, that retrieves the proper object path based on the input YANG path.

Fetch Object Algorithm. Algorithm 1 shows the $\texttt{fetch}_o$ procedure, which, given an object path $\boldsymbol{\rho}_o$, retrieves the reference to the corresponding object. To do this, it performs an iterative access to run-time variables in the object tree, fetching, at each iteration, the object associated with the next label of the path $\boldsymbol{\rho}_o$ given as input (Algorithm 1, line 2). For each label, the next object is fetched through the function `get_attribute`, in case the previous object is not a collection (line 7), while

Algorithm 1 `fetch`$_o$ for object path $\boldsymbol{\rho}_o$

Input: $\rho_o \in L_o^n$
Output: $o \in O$

1: $\boldsymbol{o} = \boldsymbol{o}_{root}$
2: **for all** $\lambda_i \in \boldsymbol{\rho}_o$ **do**
3:     **if** $o \in O_l$ **then**
4:         $\lambda_i^p = \texttt{param_label}_o(\texttt{sub_path}(\boldsymbol{\rho}_o, i))$
5:         $\boldsymbol{o} = \texttt{get_by_attribute}(\boldsymbol{o}, \lambda_i^p, \lambda_i)$
6:     **else**
7:         $\boldsymbol{o} = \texttt{get_attribute}(\boldsymbol{o}, \lambda_i)$
8:     **end if**
9: **end for**
10: **return** $\boldsymbol{o}$

`get_by_attribute` is used otherwise, passing the parametric label as attribute (lines `4-5`). Both these functions are provided by the Reflective Library (as described in Section 5.1).[4]

Example: Referring to the object tree in Figure 5 , lets consider object path $\boldsymbol{\rho}_o$ = `/networking/ifs/if0` as an input to the `fetch`$_o$ procedure. The algorithm iterates over the three labels of $\boldsymbol{\rho}_o$ (i.e., ''`networking`'', ''`ifs`'' and ''`if0`'') to access, each time, the next nested object until the reference to the last one (''`if0`'') is finally fetched. At the first iteration, the attribute `networking` of the root object is accessed (Algorithm 1, line `7`), then, at the second iteration attribute `ifs` of the object `networking` is accessed. Since `ifs` is a list, during the third and last iteration the procedure performs a lookup on the PMT to discover its parametric label λ_i^p, i.e., `ifname` (Table 2 , row 7); the parametric label is then used to fetch the correct object inside the list `ifs`, namely, the one whose attribute `ifname` is equal to ''`if0`'' (Algorithm 1, lines `4-5`). Since ''`if0`'' is the last label of the object path, finally the `fetch`$_o$ procedure returns a reference to this object (Algorithm 1, line `10`).

The two main operations needed to map the SDNApp run-time variables (that represent configuration and run-time state) into an high-level YANG-modeled structure (and vice versa) are described by the recursive algorithms `get_node` and `set_node`.

Get Node Algorithm. Each time the ToY Agent receives, from a ServiceApp, a `get()` command related to a given YANG path $\boldsymbol{\rho}_y$, it uses procedure `get_node` to create and return the corresponding instance node (i.e., JSON data). It is *(i)* structured according to the piece of data model associated with the resource $\boldsymbol{\upsilon} = \texttt{fetch}_y(\boldsymbol{\rho}_y)$ and *(ii)* filled with current values of application variables. To fill the instance node with proper values, a series of recursive calls to `get_node` are performed (for readability, recursive calls are marked with underscores in Algorithm 2). At each recursion step, the algorithm relies on `map_path` and `fetch`$_o$ procedures (line `1`) to fetch the object $\boldsymbol{o}$ that, according with the PMT, is mapped on the resource identified by the input YANG path $\boldsymbol{\rho}_y$. The algorithm starts building the node instance of the resource identified by the input YANG path $\boldsymbol{\rho}_y$. Then, at each recursion step, this path is extended in order to build and fill all inner nodes (Algorithm 2, lines `6` and `13`); these inner nodes are then added to the parent node (lines `7` and `14`)[5]. Each recursion branch stops when a leaf resource is reached; the corresponding (elementary) instance node is filled with the value of the object fetched in that recursion (lines `16-17`).

Example: We now provide an example of the `get_node` procedure, using as reference the YANG in Figure 3 and supposing that a ServiceApp needs to retrieve the resource identified by the YANG path $\boldsymbol{\rho}_y$ = `/nat/interfaces/private`. In Algorithm 2, line `1`, the `get_node` procedure retrieves the object path $\boldsymbol{\rho}_o$ = `/networking/ifs/if0` through a lookup into the PMT (Table 2 , row #8). Then, in the same line, it gives this object path as input to the `fetch`$_o$ procedure, in order to get the reference to the corresponding object $\boldsymbol{o}$ (as described in the previous example). After the object $\boldsymbol{o}$ has been fetched, the same is done for the corresponding resource $\boldsymbol{\upsilon}$ (line `2`) through the YANG model, which is needed to know how the instance node must be structured. Since in our example the resource `/nat/interfaces/private` is a container, a key-value instance node is initialized (Algorithm 2, line `4`). Then, for each child resource, the algorithm is repeated recursively and the resulting child-node is appended using its label as key (lines `6-7`). For instance, let's consider the child resource with label ''`address`''; the path $\boldsymbol{\rho}_y$ is extended with this label (i.e., `/nat/interfaces/private/address`) and `get_node` is called again. At this point, the object path obtained through the lockup into the PMT is `/networking/ifs/if0/ipv4/address`. Thus, `fetch`$_o$ accesses to the inner objects, i.e., first `ipv4`, then `address`, thus returning the last one, whose value is the IP address of the private interface. Since

[4] Algorithm 1 uses an helper function `sub_path` to truncate a path to the position given as input (position is considered from the root for positive inputs, from the last label for negative ones). For instance, `sub_path`(''`/natData/sessionList/0x26`'', −1) = ''`/natData/sessionList`''.

[5] Algorithm 2 uses an helper function `add_path` to extend the YANG path with the label given as input.

Algorithm 2 `get_node` from a YANG path $\boldsymbol{\rho}_y$

Input: $\boldsymbol{\rho}_y \in L_y^n$
Output: $\boldsymbol{v} \in \mathcal{N}$

1: $\boldsymbol{o} = \texttt{fetch}_o(\texttt{map_path}(\boldsymbol{\rho}_y))$
2: $\boldsymbol{\upsilon} = \texttt{fetch}_y(\boldsymbol{\rho}_y)$
3: **if** $\boldsymbol{\upsilon} \in Y_{co}$ **then**
4: $\quad \boldsymbol{v} = \texttt{dict_node}()$
5: $\quad$ **for all** $\boldsymbol{\upsilon}_i \in \boldsymbol{\upsilon}$ **do**
6: $\quad\quad \boldsymbol{v}_i = \underline{\texttt{get_node}}(\texttt{add_path}(\boldsymbol{\rho}_y, \texttt{label}_y(\boldsymbol{\upsilon}_i)))$
7: $\quad\quad \texttt{put_element}(\boldsymbol{v}, \texttt{label}_y(\boldsymbol{\upsilon}_i), \boldsymbol{v}_i)$
8: $\quad$ **end for**
9: **else if** $\boldsymbol{\upsilon} \in Y_{li} \cup Y_{ll}$ **then**
10: $\quad \boldsymbol{v} = \texttt{list_node}()$
11: $\quad$ **for all** $\boldsymbol{o}_i \in \boldsymbol{o}$ **do**
12: $\quad\quad \lambda_i^p = \texttt{param_label}_o(\texttt{map_path}(\boldsymbol{\rho}_y))$
13: $\quad\quad \boldsymbol{v}_i = \underline{\texttt{get_node}}(\texttt{add_path}(\texttt{path}_y(\boldsymbol{\rho}_y, \lambda_i^p)))$
14: $\quad\quad \texttt{append_element}(\boldsymbol{v}, \boldsymbol{v}_i)$
15: $\quad$ **end for**
16: **else**
17: $\quad \boldsymbol{v} = \boldsymbol{o}$
18: **end if**
19: **return** $\boldsymbol{v}$

Algorithm 3 `set_node` $\boldsymbol{v}$ in object associated with $\boldsymbol{\rho}_y$

Input: $\boldsymbol{\rho}_y \in L_y^n$, $\boldsymbol{v} \in \mathcal{N}$

1: $\boldsymbol{\rho}_o = \texttt{map_path}(\boldsymbol{\rho}_y)$
2: $\boldsymbol{o} = \texttt{fetch}_o(\boldsymbol{\rho}_o)$
3: **if** $\boldsymbol{v} \in \mathcal{N}_{kv}$ **then**
4: $\quad$ **for all** $\boldsymbol{v}_i \in \boldsymbol{v}$ **do**
5: $\quad\quad \underline{\texttt{set_node}}(\texttt{add_path}(\boldsymbol{\rho}_y, \texttt{key}(\boldsymbol{v}_i), \boldsymbol{v}_i)$
6: $\quad$ **end for**
7: **else if** $\boldsymbol{v} \in \mathcal{N}_l$ **then**
8: $\quad$ **for all** $\boldsymbol{v}_i \in \boldsymbol{v}$ **do**
9: $\quad\quad \boldsymbol{id}_i = \texttt{get_element}(\boldsymbol{v}_i, \texttt{param_label}_y(\boldsymbol{\rho}_y))$
10: $\quad\quad \texttt{append_new}(\boldsymbol{o}, \texttt{param_label}_o(\boldsymbol{\rho}_o), \boldsymbol{id}_i)$
11: $\quad\quad \underline{\texttt{set_node}}(\boldsymbol{v}_i, \texttt{add_path}(\boldsymbol{\rho}_y, \boldsymbol{id}_i))$
12: $\quad$ **end for**
13: **else if** $\boldsymbol{v} \in \mathcal{N}_e$ **then**
14: $\quad \boldsymbol{o}_{parent} = \texttt{fetch}_o(\texttt{sub_path}(\boldsymbol{\rho}_o, -1))$
15: $\quad \texttt{set_attribute}(\boldsymbol{o}_{parent}, \texttt{label}_o(\boldsymbol{o}), \texttt{value}(\boldsymbol{v}))$
16: **end if**

the resource $\boldsymbol{\upsilon}$ (i.e., `address`) of the current recursion is a leaf, this value is directly used to fill the node $\boldsymbol{v}$ (line 17). The just filled instance node is then returned to the caller (line 19) so that it can be appended to the parent node (lines 7 or 14) The same is done for other children resources of `/nat/interfaces/private`, namely `name` and `netmask`. At the end of all the recursion steps, the just built JSON node will be equal to the one shown in Figure 3 ; at last, it can be returned by the ToY Agent to the ServiceApp as response to the `get()` command.

Set Node Algorithm. The `set_node` procedure (Algorithm 3) is used by the ToY Agent to modify the configuration of an SDNApp each time the ToY Agent receives a `set()` command from a ServiceApp. It takes a YANG path $\boldsymbol{\rho}_y$ and an instance node $\boldsymbol{v}$ (e.g., JSON) as input, and configures the SDNApp modifying its run-time variable according to the data in the instance

node. To configure all run-time variables with value stored in the instance node, a series of recursive calls to `set_node` are performed (for readability, recursive calls are marked with underscores in Algorithm 3). Similarly to `get_node`, the algorithm starts by fetching the object corresponding to the input YANG path ρ_y; this is configured through functions `append_new` or `set_attribute` (lines `10`, `14`), provided by the Reflective Library. At each recursion step, the base YANG path is extended with the label of each child resource, in order to fetch and configure all inner objects. If a new object must be created as element of a list (lines `8-11`), first its unique ID (attribute corresponding to the parametric label) is initialized according to the value of the instance node (the attribute initialization is implemented by the function `append_new` of the Reflective Library); then, it is in turn configured through a recursive call to `set_node` (line `11`). Also in this case, each recursion branch stops when a leaf node in the YANG data model is reached; its instance value is used to configure the corresponding object (lines `14-15`).[6]

Example: Lets suppose that instance node $\boldsymbol{v}$ of Figure 3 is used to configure an application having the object tree in Figure 5 . The YANG path ρ_y = `/nat/interfaces/private` is given as input to `set_node`, together with the instance node $\boldsymbol{v}$. Algorithm 3 lines `1-2` fetch object $\boldsymbol{o}$ and resource $\boldsymbol{\upsilon}$. At this point, since the instance node is a key-value structure (condition in line `3` is verified), the YANG path is extended once for each inner node (lines `4-5`), in order to continue the recursion. For instance, for the inner node with key "`address`", the input YANG path `/nat/interfaces/private` is extended as `/nat/interfaces/private/address`; this is used, together with the inner node value "`192.168.1.154`", to call again the `set_node` procedure (line `5`). With this new input, due to the mapping into the PMT (Table 2 , row #9), object $\boldsymbol{o}$ returned by $\texttt{fetch}_o$ in line `2` is the string attribute `privateAddress` (Figure 5). Since $\boldsymbol{v}$ is now an elementary node, object $\boldsymbol{o}$ is set to its value "`192.168.1.154`" (Algorithm 3, line `15`). In the same way, `set_node` is called recursively also for other child resources of `/nat/interfaces/private`, namely `name` and `netmask`.

The performance of the algorithms presented above are analyzed and discussed in the next section.

6 | VALIDATION

To validate the approach proposed in this paper, we set up a testbed consisting of a Mininet virtual network, controlled by an instance of the ONOS SDN controller (Junco version 1.9.0)[7]. Our evaluation focuses on two major sets of results. We first measure the overhead introduced by the ToY Agent for individual *read* and *write* operations, together with update notifications; performance obtained with an ad-hoc manipulated SDNApp are used as comparison reference. Then we analyze the applicability of our approach over each use case presented in Section 3.

6.1 | ToY Agent Evaluation

To evaluate the delay introduced by the ToY Agent, we used an existing application called *"SSSA-analytic-tool"*, which has been validated in previous works of authors to serve network orchestration functions within SDN domains[26]. Since this application has been developed independently from this work, it does not natively provide any interface to allow an external service (e.g., network service orchestration) to access its state. Indeed, the above-mentioned network orchestration functions could only be carried out by referring to implementation-specific data formats and API.

To enlarge the scope of applicability of the *SSSA-analytic-tool*, this SDNApp has been extended by integrating it in the software architecture presented in this paper. Thus, we simply included the generic ToY Agent module (the source code of our prototype is available at[27]) and initialized it on bootstrap by merely invoking its "`start`" method. Additionally, to evaluate the ToY Agent overhead against traditional ad-hoc approaches, we conducted our tests also on a second version of the *SSSA-analytic-tool*, that has been modified by means of a REST interface that enables direct access to run-time data. In this case, the JSON representation is strictly dependent on the application itself (it follows the same structure of the SDNApp variables) and there is no higher level common abstraction.

Measurements have been taken considering the variation of the `depth` parameter, i.e., the maximum length among the YANG node branches starting from the node that has to be set/retrieved by the operation to perform. For instance, all leafs have depth `0`, the instance node of the *private interface* in Figure 3 has depth `1` (since it contains only elements with depth 0), and so on. The maximum depth of an instance node of the YANG in Figure 3 is `3` (we then say that the depth of the YANG model itself is `4`). For validation purpose, a YANG model with depth 9 has been used ($\approx$ 1000 resources), together with a PMT to associate YANG

[6] Note that, to modify the value of a leaf object, Algorithm 3 needs to fetch the reference to the parent object.

[7] In our tests we deployed ONOS, Mininet, and all modules of our architecture on a Linux debian 4.14.13-1-amd64, on top of a physical machine with 16 GB of RAM running an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz.

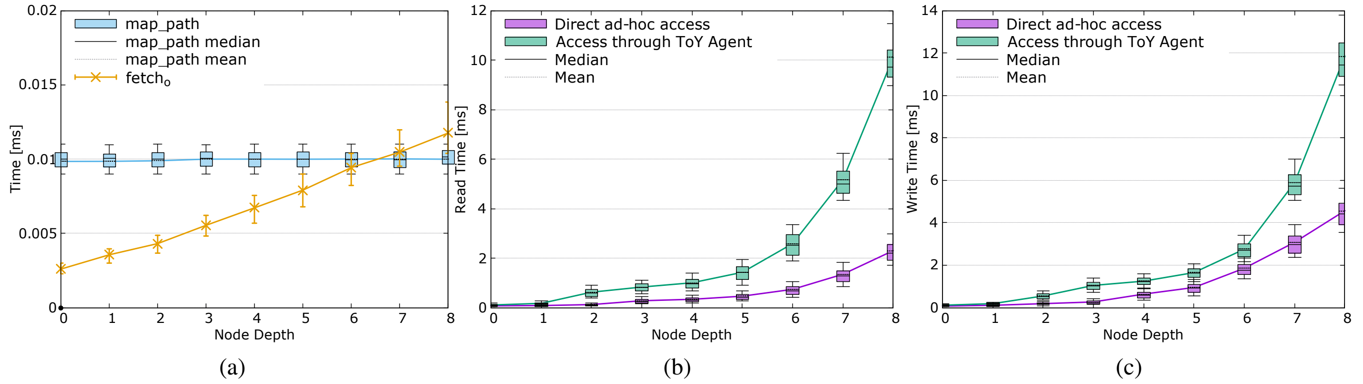

FIGURE 8 (a) Partial times needed to *(i)* map YANG paths into Object paths and *(ii)* fetch an object reference starting from its path, for different depth levels. (bc) `get_node` (b) and `set_node` (c) total times for different depth levels; values are compared with direct ad-hoc access. Upper and lower quartils are plotted along with errorbars.

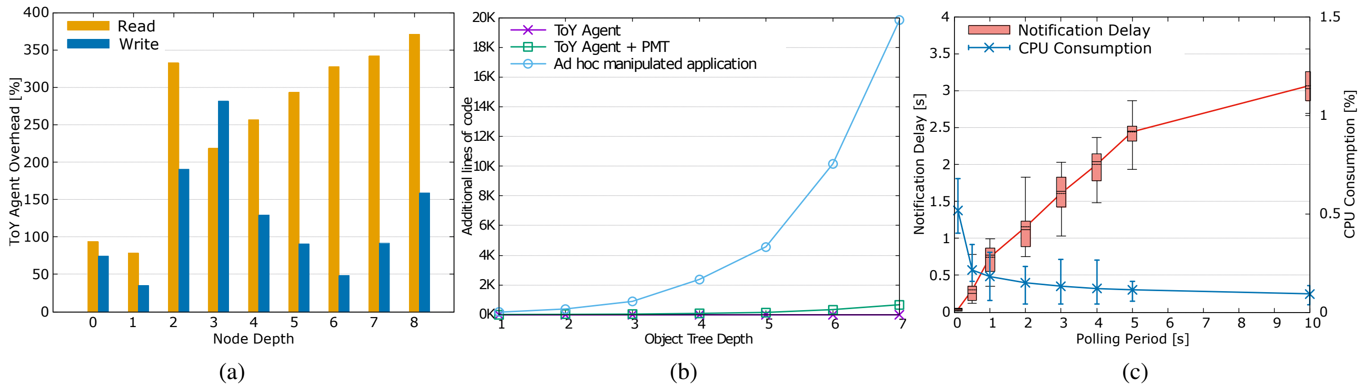

FIGURE 9 (a) Time overhead introduced by the ToY Agent for read and write operations, at different depth levels. (b) Additional lines of code needed to enable read and write access to SDNApp variables for different depth levels. (c) ToY Agent CPU consumption and notification delay for different polling periods (upper and lower quartils are plotted along with errorbars).

resources with the SDNApp variables.[8] Higher levels of depth have not been considered since rarely widespread data-models exceed a depth of 6.

Access Time Evaluation. We performed some `get()`/`set()` operations on the variation of the depth parameter. Partial times of each algorithm described in Section 5.2 have been measured to test the performance of each of them. Results taken over thousand samples for each depth level are shown in Figures 8 (a-c).

In particular, Figure 8 a shows the partial times required by the `map_path` and the $\texttt{fetch}_o$ procedures. The partial time taken to convert a YANG path into the corresponding object is almost constant (about ten microseconds) with the depth growing, since this operation corresponds to a lookup on the PMT. The $\texttt{fetch}_o$ procedure time has the same order of magnitude, but its value slightly increases with the increase of the depth (with a slow linear growth).

The graph in Figure 8 b depicts the entire time needed by the ToY Agent to perform read (`get_node`) operations. As the graph shows, even if the curve features an exponential trend (as expected from the use of recursive algorithms), for depth levels below 6 it still grows almost linearly. This is a significant result since widespread data-models usually feature data model with far smaller depths. Additionally, values are compared with those of the ad-hoc manipulated application. For depth levels below 7, despite the ToY Agent overhead, the total time still sticks below a couple of milliseconds, that is reasonably low for read operations of a whole SDNApp state. Similar considerations can be done for write (`set_node`) operations (Figure 8 c), which in general require slightly higher times.

[8]The YANG model and the PMT used are not shown here for space reasons. A big data model has been prepared ad-hoc to measure performance to a very high (and uncommon) level of depth. For the same purpose, the *SSSA-analytic-tool* application itself has been extended with a significant amount of additional variables.

Figure 9 a depicts the proportionate delay overhead introduced by the ToY Agent normalized on the baseline direct access to SDNApp variables. The graph shows that, especially for write operations, there is no direct connection between the proportionate overhead and the depth level of the accessed node, i.e, increasing the depth level does not necessarily leads to an higher percentage of overhead. This is explained by noting that, despite being noticeable faster, also the ad-hoc approach features exponential trends with the node depth (Figures 8 b-c). Indeed, this is the expected trend for object parsing/serialization. The variations in Figure 9 a are instead the symptoms that the algorithm is more susceptible to the particular nature of the data that each request has to manipulate (e.g., differences between the YANG and the internal representation, data structures used to implement object collections, etc.).

Finally, please note that this overhead does not refer to any particular application, since it is based on measurement taken using a large data model that features arbitrary data structures. The correlation between the overhead and the specific use case, along with considerations about the involved data structures, will be analyzed in Section 6.2.

Programming Overhead Evaluation. The overhead in terms of additional line of codes that the two approaches introduce to the original SDNApp is shown in Figure 9 b. In Section 4.3.3 we have already shown that 3 lines of code are enough to import and run our ToY Agent into an existing SDNApp. On the other hand, the "ad-hoc" application has been extended with a minimal Jersey REST server that is managed by ONOS. This required some few extra classes with a considerable amount of additional code, over than some modification in the original SDNApp source code. Ad-hoc modifications are detailed below.

For the REST interface we needed two small classes plus the implementation of a controller object, whose length depends on the depth level of the run-time state, as it needs to implement two different methods for every single object and variable that have to be exported. In our implementation, we needed 9 lines of code on the REST controller for each minimal `get()` operation and 15 for each `set()`. Of course this grows exponentially with the depth level (for our setup, we implemented just the methods needed to measure the performance, i.e. two for each depth level). Since ONOS manages the REST service through an inversion of control paradigm, no additional code has been required to run it. However, getter/setter methods have been added to the attributes of the root class, to allow the REST controller to access them (a total of 32 lines for 4 attributes). Most important, all existing classes have been extended with two custom methods to perform custom JSON parsing and serialization ($\approx$ 25 lines per class, depending on the number of attributes to be exposed). Totals for each depth level are shown in Figure 9 b. Since the code overhead in the ToY Agent is extremely low and constant (3 lines of code), the graph also takes into account the lines of the PMT, despite being this completely external and not an application invasive modification. The figure clearly assesses that, from the application maintainer point of view, the programming overhead to enable read and write access through our approach can be considered negligible compared with an ad-hoc manipulation of the SDNApp.

Notification Delay Evaluation. As presented in Section 4.2, ServiceApps may subscribe to desired resources to be notified when something on them changes. Since variables are monitored with a polling mechanism, we measured the notification delay together with the CPU consumption of the system, on the variation of the polling frequency[9]. Figure 9 c summarizes the results taken over hundred samples. The graph shows how, obviously, with a growing polling period, the time needed to receive the notification increases, while the CPU consumption decreases (CPU consumption values are shown in the vertical axis on the right side). Measured values show that, with a polling period of $\approx 0.8s$ or greater, CPU consumption remains almost constant (with a slow decrease from $\approx 0.2\%$ to $\approx 0.1\%$), while the notification latency starts to grow (with an overall logarithmic trend, but the curve slope is very high until a polling period of $5s$). A good compromise, therefore, is a polling period of $0.8s$, that provides low CPU consumption ($\approx 0.2\%$) with a reasonable average notification latency of $\approx 0.5s$.

6.2 | Use Cases Evaluation

In this subsection, we compare the performance of the ToY Agent against the ad-hoc application under some specific scenarios, to discuss the delay overhead introduced by this component for different use cases. For this purpose, in Figure 10 , we describe the workflows of the use cases presented in Section 3 which detail the list of operations that the ServiceApps execute on the SDNApps (i.e., *read/write* operations).

IPS workflow. In Figure 10 a we assume that the ServiceApp (IPS) periodically performs *read* operations toward the IDS and the Network Monitor SDNApps to collect, analyze and aggregate the network status. Once an intrusion is detected (e.g., a threat or an anomalous traffic), the IPS performs a *write* operation on the SDNApp (i.e., firewall in this case) to install a new rule to protect the network infrastructure.

[9]Measurements still refer to our validation YANG model with depth 9 and $\approx$ 100 resources.

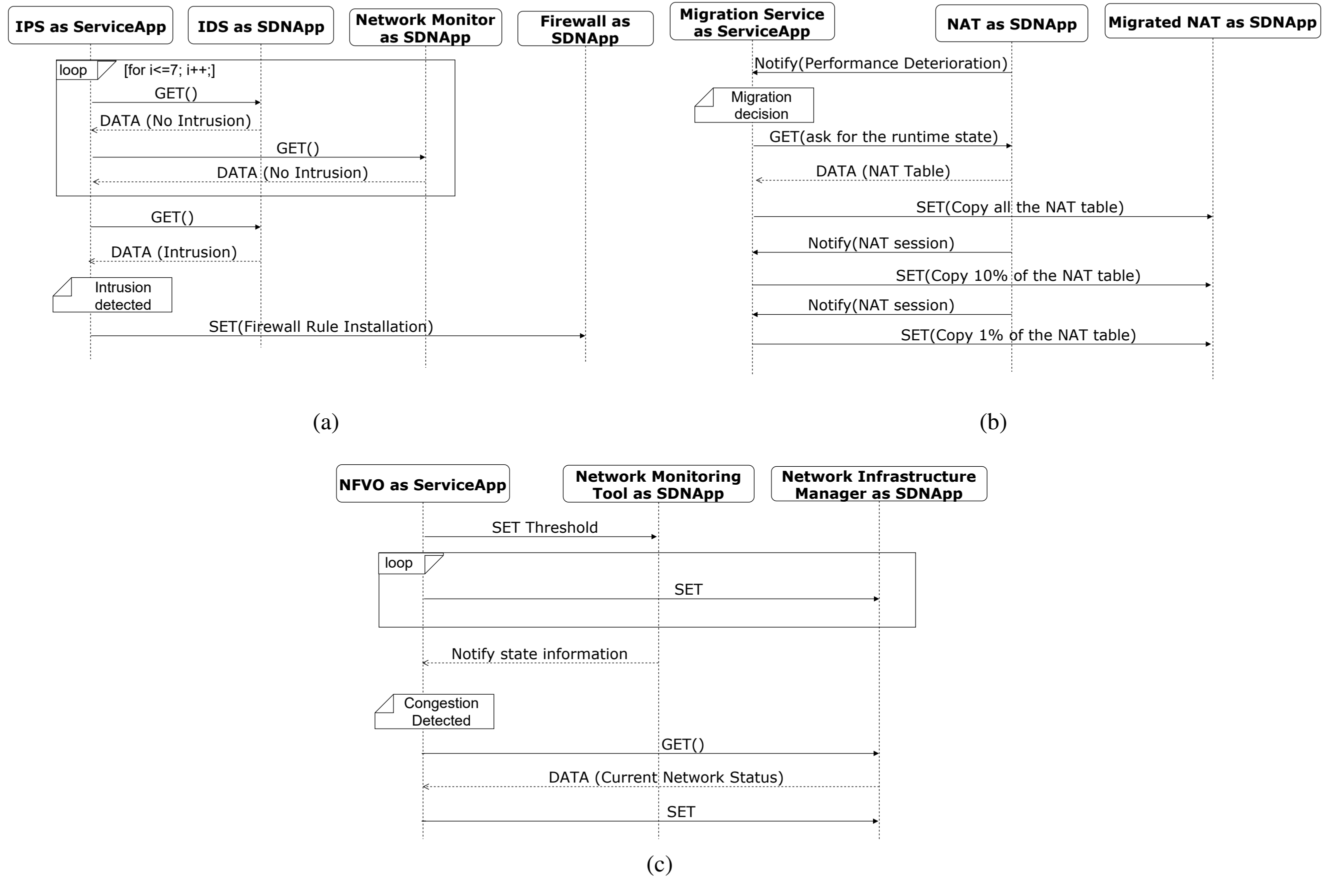

FIGURE 10 Use case workflows for (a) IPS (b) Service Migration and (c) Service Orchestration scenarios.

Migration workflow. The workflow related to this scenario is depicted in Figure 10 b. The ServiceApp (i.e., migration service) waits for threshold based notifications from a monitored SDNApp (i.e., a NAT) which reports that the quality of service is deteriorating and that a migration is necessary. At that point, the ServiceApp first performs a *read* operation to acquire the current run-time state of the existing instance (the state can be modeled by the YANG model in Figure 3). Then, it copies the whole state to a new SDNApp instance (a *write* operation is performed). We then assume that, if in the meanwhile, the status of some SDNApp parameters changes (e.g., new entries on the NAT table), other smaller *write* operations will be executed, where those parameters are updated.

Orchestration workflow. Finally, Figure 10 c depicts the workflow of the service orchestrator use case. In this scenario, the NFVO executes some *write* operations to setup virtual links connecting VNFs of a Network Service. The orchestrator may set some QoS threshold values according to negotiated SLA for a given Network Service (e.g., maximum switches throughput, maximum links bandwidth for virtual links). When a SLA is violated (e.g., due to a congestion in the network), the NFVO receives a notification from the Network Monitoring Tool. At that point, the NFVO first gets the current status of the network by executing a *read* operation and then, through a *write* operation, properly configures the flow entries of the virtual link that is suffering from the congestion, thus satisfying the SLA.

From the above descriptions, we come to the conclusion that each use case has different characteristics in terms of interactions between the ServiceApps and SDNApps, and in terms of exchanged data. In the following, we numerically characterize the 3 use cases and we report, for each of them, the overall amount of time needed to perform all the listed operations. For what concerns the IPS scenario, as stated in[28], a tuned IDS produces an average of 3000 alerts per day, among which only 13.2% report an intrusion. Based on these data, we can assume that, for each 7 *read* operations of below SDNApps each performed every 30 seconds, 6 of them do not report an intrusion, thus tens of bytes are exchanged. Once an intrusion is detected, a slightly bigger *read* that retrieves all the related information is performed, followed by the small *write* operation that sets the firewall rule. These data may be modeled with depth 1 instance nodes. On the contrary, in the migration service scenario, a threshold based notification is immediately followed by the decision of migrating the SDNApp, to avoid performance deterioration. At

TABLE 4 Delay introduced by the ToY Agent for three different use cases compared with direct ah-hoc access (average times taken over thousand samples are shown).

Use Case	Direct Access [ms]	ToY Agent [ms]
IPS	0.8611	1.5944
Migration	1516.41	2298.81
Orchestration	0.8131	1.1299

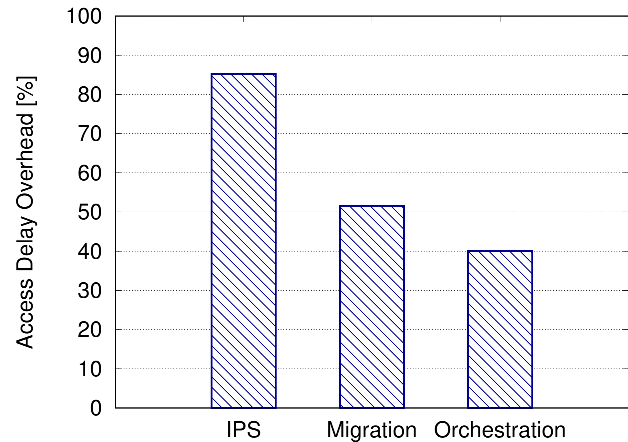

FIGURE 11 Overall percentage overhead introduced by the ToY Agent for three different use cases.

this point the ServiceApp and the SDNApp exchange huge amount of data compared to the previous scenario, to migrate the application state (the whole NAT table in a big network may consist of around 10K of entries). The *write* operation which sets this amount of data modeled with a 3-depth instance node requires hundreds of milliseconds, if the incremental overhead of all the table entries (depth 1) is also considered. The subsequent *write* operations regard few entries of the table and then require less time (tens of milliseconds). Finally, the service orchestration scenario mainly requires *write* operations that exchange only few bytes (value of the threshold, flow rules parameters) and which require few time (less than one millisecond). The number of *write* operations necessary to install the Network Service depends on the particular scenario. In this numerical evaluation, we have considered the scenario described in [29], where 5 virtual links are installed.

According to these assumptions, we evaluated, for each use case, the overall time necessary to execute all the operations described above using both *(i)* the ToY Agent and *(ii)* ad-hoc manipulated applications with direct access to run-time state. Results are reported in Table 4 . Moreover, to better evaluate the overhead that the TOY Agent introduces in each use case, we normalized the overall delay to the baseline value needed with direct access to application variables (Figure 11). For all the use cases, the ToY Agent introduces a proportionate overhead below the 100%. However, a particular high overhead has been registered for the IPS scenario compared to the others, as it is the only one where the overhead largely exceeds a value of 50%. It appears that the high proportion of *read* operations that characterizes this use case constitutes a penalty for using the ToY Agent. In fact, as shown in Section 6.1 (Figure 9 a), mapping algorithms seem to suffer, on average, more for *read* operations than for *write* ones, when compared to direct object access.

7 | DISCUSSION

This section points out the advantages of the approach we proposed in this paper, as well as it discusses the requirements that must be satisfied by the SDNApps for being able to exploit our run-time state inspection algorithms and the companion software framework.

Experimental validation demonstrates that our approach achieves its objectives while guaranteeing low implementation and execution overhead. More specifically, developers of SDNApps need to perform very few modifications to their source code to enable automatic state inspection (i.e., only 3 lines of codes were added to the `SSSA-analytic-tool` application, as shown in Figure 7). Furthermore, as shown in Section 6, the execution time of the above algorithms is low (i.e., units of milliseconds for SET/GET operations). Of course, even if the overall overhead measured on reference use cases workflows is reasonable, the percentage overhead of single read/write operations may be high in some cases. The main limitation is that the current prototype adopts an interpreter-based approach to retrieve data from the application. Therefore, a performance improvement may be achieved by integrating well-known dynamic code generation approaches, through which, once the data model is known, the ad-hoc code needed to parse data can be generated.

It is worth noticing that the proposed approach is rather general, not being limited to SDNApps. In fact, the creation of an abstract data model for each application and the capability to provide automatic access to its internal data may be useful in other contexts beyond SDN applications and controllers, and may be an interesting direction for our future research activities.

Our work pointed out several requirements for current SDNApps to be able to exploit the capabilities of our state inspection library, to enable the seamless adaptation of existing applications to the new service orchestration architecture. First, it requires SDNApps to be thread safe, e.g., to avoid that the ToY Agent modifies the value of a particular variable while the application is executing a critical section. Second, SDNApps must be written in a language (e.g., Java, Python) that supports reflection to access variables or a similar mechanism, such as Java Reflective APIs[30]. Nevertheless, this does not represent an important limitation for SDNApps, since most important SDN controllers are developed in Java and exploit the OSGi framework, hence support only applications developed as Java bundles.

Finally, some limitations of this approach can be envisioned as well. First, the State Agent must use polling cycles to promptly discover any modification to a run-time variable of the SDNApp that, according to the YANG data model, has to be exported in the `onchange` mode (i.e., as soon as it value change), and consequently publish the new value on the message bus. According to our tests, this may have a negligible impact in terms of CPU cycles in case the update frequency is kept on the order of tens of milliseconds (which looks reasonable for most of the applications), but may worsen in case higher frequencies are requested[10]. An alternative solution consists in avoid the polling by modifying the application and defining a callback invoked when the variable value changes. However, this goes against one of the objectives of our work, which aims at limiting the impact on the source code of SDNApp to simplify the porting of existing applications to our framework.

Second, the language that has to be used to create the *object path* in the PMT is rather complex, hence writing the required statements may be far from trivial. This is due to the possible very different design of the YANG data model compared to the internal structure of the application, and the necessity to obtain the object path purely from the incoming request, which contains only references to the data model. However, in our experience the effort required to design a YANG-native interface and implement the required interactions with the northbound API (e.g., in case the SDNApp has to be developed from scratch) is much higher than the one guaranteed by our approach. This was verified by implementing a YANG-based northbound interface compatible to our software architecture in some SDNApps written in Python[31], and leveraging an helper library to facilitate the writing of the interface code; the application was obviously more optimized (e.g., when handling `onchange` variables), but the development process was more time-consuming (on the order of one man/week for an application such as a NAT), which confirmed the advantages of the approach proposed in this paper, when possible.

Third, our approach is not available as long as precompiled applications are concerned[11], which may be rather common when VNFs are deployed in a cloud-based datacenter as virtual machines, such as many network applications that are inherited from the Linux world (`iptables` for NAT and firewalling, `bind` for DNS, etc.). However, our experience shows that also the above applications can be ported to the proposed software architecture by using a mixture of bash scripting, monitoring of in-kernel structures, and writing the equivalent of a wrapper ToY agent that implements the required northbound interface. For instance, in[32] an `iptables`-based VM was integrated within the overarching software architecture presented in this paper. In this respect, it is worth noticing that the mapping algorithms presented in Section 5 have been used also in this case to provide the state inspection capabilities to traditional applications as well. This suggests that the state inspection algorithms presented in this paper may have a general validity, hence are not subject to the requirements that refers to the SDNApps presented earlier in this Section.

8 | CONCLUSION

This paper presents an architecture that allows to makes SDN application internal state available to external Services, granting both read and write access to run-time variables, and also the possibility to receive notifications with state updates any time something of interest changes.

To model the arbitrary, possibly complex, run-time state of SDNApps into high level structures, the YANG modeling language has been used. A set of algorithms maps the low level data structure of the SDNApps state into the high level YANG one, independently from the structure itself and relying just on information kept into the SDNApp specific YANG model and Path Mapping Table. This allows to include arbitrary SDNApps into the proposed architecture without introducing neither changes

[10]The polling frequency is one of the parameters of our library, which the caller SDNApp can set upon loading the ToY Agent.

[11]For the sake of precision, although the approach presented in this paper may be more general, this paper focuses on SDNApps that operate in an SDN controller, which should exclude all the applications that are executed as native software in VMs.

on the code of the application itself nor computing overload. Performance validation showed that the delay introduced by the proposed approach and mapping algorithms scale well even for unusually big YANG models with high depths. By analyzing the applicability of this approach on different use cases, it appears that that the introduced access overhead is reasonable for most scenarios. Moreover, numerical data show that the delay overhead is widely compensated by the few programming requirements, in terms of additional lines of code, compared with an ad-hoc manipulation of each application.

Although our validation has been carried out in the SDN context, this work can be easily extended and can accommodate other kind of applications as well. This may allow high level services to exploit, with a modular approach, any kind of Network Service across technologically heterogeneous infrastructures (e.g., SDN-based networks, data centers, modern CPEs). Such an extension is left as a future work.

ACKNOWLEDGEMENT

The authors would like to thank Tierra Telematics and Telecom Italia Mobile, in particular Nicola Smaldone and Antonio Manzalini, for their support in making this work happening. We would also thank Lara Cesaretti, who contributed to the first version of the prototype presented in this paper.

References

1. ONOS - Open Network Operating System http://onosproject.org/.
2. Open Daylight https://wiki.opendaylight.org/.
3. Paganelli F., Ulema M., Martini B.. Context-aware service composition and delivery in NGSONs over SDN. *IEEE COmmunication Magazine.* 2014;8:97–105.
4. Carniani E., D'Arenzo D., Lazouski A., Martinelli F., Mori P.. Usage Control on Cloud Systems. *Future Gener. Comput. Syst..* 2016;63(C):37–55.
5. Network Configuration Protocol (NETCONF) https://tools.ietf.org/html/rfc6241.
6. Bierman A., Bjorklund M., Watsen K.. *RESTCONF Protocol.* RFC 8040: RFC Editor; 2017.
7. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF) https://tools.ietf.org/html/rfc6020.
8. Richardson L., Ruby S.. *RESTful Web Services.* Sebastopol, CA, USA, OReilly, 2007.
9. Bianchi G., Bonola M., Capone A., Cascone C.. OpenState: programming platform-independent stateful openflow applications inside the switch. *Computer Communication Review.* 2014;44:44-51.
10. Bosshart Pat, Daly Dan, Gibb Glen, et al. P4: programming protocol-independent packet processors. *Computer Communication Review.* 2014;44:87-95.
11. Han Wonkyu, Hu Hongxin, Zhao Ziming, et al. State-aware Network Access Management for Software-Defined Networks. *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies.* 2016;:1–11.
12. Cisco Network Services Orchestrator (NSO) Solutions https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions.html.
13. Rajagopalan S., Williams D., Jamjoom H., Warfield A.. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.. *NSDI.* 2013;13:227–240.
14. Rajagopalan S., Williams D., Jamjoom H.. Pico Replication: A high availability framework for middleboxes. *Proceedings of the 4th annual Symposium on Cloud Computing.* 2013;:1.

15. Gember-Jacobson Aaron, Viswanathan Raajay, Prakash Chaithan, et al. OpenNF: Enabling Innovation in Network Function Control. *SIGCOMM Comput. Commun. Rev.*. 2014;44(4):163–174.

16. Martini Barbara, Paganelli Federica. A service-oriented approach for dynamic chaining of virtual network functions over multi-provider software-defined networks. *Future Internet.* 2016;8(2):24.

17. SELFNET https://selfnet-5g.eu/.

18. Manzalini Antonio, López Diego R., Lønsethagen Håkon, et al. A unifying operating platform for 5G end-to-end and multi-layer orchestration. In: Netsoft 2017:1–5; 2017.

19. Peterson Larry, Baker Scott, De Leenheer Marc, et al. XoS: An Extensible Cloud Operating System. In: BigSystem '15:23–30ACM; 2015; New York, NY, USA.

20. Taleb Tarik, Ksentini Adlen, Frangoudis Pantelis A. Follow-me cloud: When cloud services follow mobile users. *IEEE Transactions on Cloud Computing, Special Issue on Mobile Cloud, Vol.PP, N°99, 2016, ISSN: 2168-7161.* 2016;.

21. Barham Paul, Dragovic Boris, Fraser Keir, et al. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*. 2003;37(5):164–177.

22. ETSI GSNFVMAN. ETSI MANO NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf.

23. Gharbaoui M., Fichera S., Castoldi P., Martini B.. Network orchestrator for QoS-enabled service function chaining in reliable NFV/SDN infrastructure. *Network Softwarization (NetSoft), 2017 IEEE Conference on.*. 2017;.

24. Clemm Alexander, Voit Eric, Prieto Alberto, et al. *Subscribing to YANG datastore push updates.* Internet-Draft draft-ietf-netconf-yang-push-07: IETF Secretariat; 2017.

25. Malenfant Jacques, Jacques Marco, Demers Franois Nicolas. A tutorial on behavioral reflection and its implementation. In: :1–20; 1996.

26. Mohammed A.A., Gharbaoui M., Martini B., Paganelli F., Castoldi P.. SDN controller for network-aware adaptive orchestration in dynamic service chaining. *Network Softwarization (NetSoft), 2016 IEEE Conference on.*. 2016;.

27. SDN Application ToY Agent repository https://github.com/netgroup-polito/sdn-app-toy-agent.

28. Tjhai G., Papadaki M., Furnell S., Clarke N.. Investigating the problem of IDS false alarms: An experimental study using Snort. *Proceedings of The Ifip Tc 11 23rd International Information Security Conference. SEC 2008. IFIP âĂŞ The International Federation for Information Processing.* 2008;278.

29. Gharbaoui M., Contoli C., Davoli G., et al. Experimenting latency-aware and reliable service chaining in Next Generation Internet testbed facility. *Proceedings of IEEE NFV-SDN.* 2018;.

30. Forman Ira R., Forman Nate. *Java Reflection in Action.* Manning Publications; 2004.

31. FROG Configurable VNF repository https://github.com/netgroup-polito/frog4-configurable-vnf.

32. Bonafiglia Roberto, Castellano Gabriele, Cerrato Ivano, Risso Fulvio. End-to-end service orchestration across SDN and cloud computing domains. In: Netsoft 2017:1–6; 2017.