Policy Framework Prototype for ONOS

Canellas Cruz, Ferran; Kentis, Angelos Mimidis; Bonjorn, Nestor; Soler, José

Published in:
Proceedings of IEEE NetSoft 2019

Link to article, DOI:
10.1109/NETSOFT.2019.8806691

Publication date:
2019

Document Version
Peer reviewed version

Policy Framework Prototype for ONOS

Ferran Canellas†, Angelos Mimidis†, Nestor Bonjorn*, José Soler†
DTU Fotonik
Technical University of Denmark
Kgs. Lyngby, Denmark
†{fccr, agmimi, joss}@fotonik.dtu.dk, *nestorbonjorn@gmail.com

***Abstract*— The Software Defined Networking (SDN) and Network Function Virtualization (NFV) paradigms can provide increased levels of flexibility, automation and programmability to fields related to Cloud and 5G technologies. One field that can benefit greatly from SDN/NFV is that of policy-based network management. Building upon previous work, this paper presents a network policy framework prototype for the ONOS SDN controller. This paper presents an architecture, where the policy manager and the different policy types are disaggregated as different applications (communicating through dedicated REST APIs). This allows for the ad-hoc installation and removal of new policy types to the policy framework, without causing system-wide disruptions. Details of the underlying framework design are provided, together with concrete insight on the network policy prototype implementation. The prototype is then validated through three Proof of Concept (PoC) policy types, a firewall, a NAT and a connectivity policy.**

***Keywords*— SDN, NFV, Policies, Policy Enforcement**

I. INTRODUCTION

Policy-based network management is a highly desirable trait by network operators, as it facilitates control over their deployments. However networks tend to be very diverse in terms of the involved protocols, Application Programming Interfaces (APIs) and supported features. This diversity can act as a barrier to scalable and flexible network-wide management solutions, as policies ideally need to be defined in a generic and not technology specific way. However recent network operating systems (e.g. ONOS, OpenDaylight), which are based on the Software Defined Networking (SDN) paradigm, are designed in a way that provides a network abstraction layer to administrators and application developers. By utilizing this layer, and based on the generalization of SDN-based network equipment, it is now possible to design and implement technology-agnostic policy-based network management frameworks.

This paper provides a redesign of the framework originally described in [1] and proposes enhancements to the policies' lifecycle. Moreover, this paper presents a prototype of the policy framework, for the ONOS SDN controller [2], which demonstrates and validates its capabilities. This prototype disaggregates the technology-agnostic logic of the policies' lifecycle from the technology-specific logic necessary to enforce the policies in the underlying network. The policy framework prototype consists of an ONOS application called Policy Manager, which is responsible for managing and storing the state of the policy instances, and a set of ONOS applications that implement the logic of the different policy types. The Policy Manager offers an external REST API that allows Create-Read-Update-Delete (CRUD) operations for policy instances. In order to perform these operations, the Policy Manager communicates with the policy type applications throw internal REST APIs.

The rest of this paper is organized as follows. Section II provides necessary background information and an overview of related work. Section III describes the design of the policy framework together with its implementation. Section IV presents a list of Proof of Concept (PoC) policy-types supported by the framework, as well as how they are validated and enforced in the network. Finally, Section V provides conclusions and considerations for future work.

II. RELATED WORK

A. Related products

In this subsection, we introduce two open source projects, which offer an execution and management context for policy enforcement. The policy framework depicted in [3] is designed for the ONAP platform. This policy framework is the "decision making" component of the platform and is the source of truth for policy decisions. The objective of the framework is to be deployment agnostic by managing policy execution and enforcement regardless of how they are deployed. Besides, the framework offers a model-driven approach in order to be flexible enough to support rapid development workflows (e.g. DevOps Model). Although the framework already supports key features such as placement or control loops, the platform's latest release still shows that the policy framework is a work-in-progress project. Given the different context (platform vs network), and the open APIs, the two proposals are not exclusive to each other. This means that with the proper extensions, the presented framework could be operated through the ONAP policy framework. The ONOS Intent Framework [4] allows applications to define the desired control behavior in form of intents rather than directly programming the network fabric. Therefore, the Intent Framework focuses on *which* actions have to be enforced in the network instead of defining *how* these actions are actually enforced. However, the Intent Framework lacks a robust conflict/context resolution mechanism and is limited to connectivity requests. Our prototype uses the ONOS Intent Framework when applicable, but also provides a whole framework on top that guarantees the correct behavior of any kind of network policy, including conflict and context validations.

B. Related work

The authors in [5] present a policy management approach that decouples the policy resolution from the policy enforcement. They target some of the drawbacks introduced by middlebox deployment. Authors claim to tackle the drawbacks identified from middlebox deployment and described throughout the paper (e.g. latency) since the traffic traverses a shorter path (bypassing the firewall) once the traffic is flagged as valid and legitimate. In our design, we focus on the SDN/NFV interworking as opposed to the focus

978-1-5386-9376-6/19/$31.00 ©2019 IEEE

on middlebox deployment of the authors. In contrast to the framework described in [5], the framework depicted herein relies on intents to define connectivity between endpoints. Moreover, the design of the framework described in this paper is decoupled from the forwarding engine, and thus, another path computation service could also be used. The authors in [6] presented a PoC of an intent-based northbound interface based on a REST API able to control the underlying network infrastructure through technology-agnostic policies. Moreover, this PoC is able to operate in multiple SDN domains, interconnected by non-SDN domains. Finally, the authors experimentally evaluate their PoC based on scalability tests. In contrast with [8], this paper presents a disaggregated architecture where policy types can be added and removed ad-hoc and a policy lifecycle that ensures that active policies are not conflicting. One of the core challenges with regards to network policy enforcement is to ensure that no two policies in the network conflict with each other at any time. Having a conflict-free set of applied policies is crucial, as it ensures that the applied network state is the desired one. As stated later in this paper at the moment the proposed framework performs conflict identification and resolution at a high level (network policy level), between policies of the same type. However an approach considered for future work would be to identify and resolve conflicts at flow rule level. Dissecting policies into their flow rules will allow for cross-policy type conflict identification and resolution. To that end, the work described in [9] can act as a possible inspiration. There, the authors classify conflicts in several categories such as redundancy, shadowing, generalization, etc.

III. Design and Implementation

The scope of this section is twofold. It introduces the design guidelines of the policy framework, including but not limited to how the policies are modelled and how they are managed within the policy framework. Then this section introduces the architecture and implementation details of the framework in relation to the ONOS SDN controller.

A. Design

In this subsection, the redesigned framework is presented, with respect [1]. The initial policy framework was designed to support platform-wide and technology-agnostic policies. To this end, a policy model and a policy lifecycle were defined. The former allows defining policies in a generic way regardless of the underlying technologies, while the latter defines the possible states of a policy, through their lifecycle, as well as the logic to move from one state to another. These two concepts are summarized hereafter.

1) Policy Model

The policy model used by the policy framework is loosely based on the Policy Core Information Model (PCIM) [7], [8]. In this model, each policy is characterized as a *Policy Rule*, which is associated to a specific *Policy Type*, and is comprised by a set of *Policy Conditions* and *Policy Actions*, each of them having a pair of a *Policy Variable* and a *Policy Value*. Moreover, the policy framework allows multiple *Policy Rules* to be added at the same time as a *Policy Group*. In this case, the *Policy Rules* are processed in priority order, based on the priority field of each *Policy Rule*. The policy model is depicted in Fig. 1.

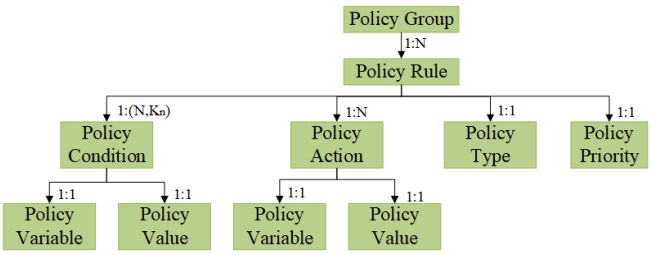

Fig. 1. The policy model used by the policy framework.

The set of *Policy Conditions* of a *Policy Rule* are represented as a 2-dimensional list, as illustrated in Fig. 1. This is treated by the framework in two ways: **Conjunctive Normal Form (CNF)**: *Policy Conditions* are defined as the intersection of unions. For instance, $(C_1^1 \cup C_2^1 \cup C_3^1) \cap (C_1^2 \cup C_2^2)$, where C_k^n is a *Policy Condition*. **Disjunctive Normal Form (DNF)**: *Policy Conditions* are defined as the union of intersections. For instance, $(C_1^1 \cap C_2^1) \cup (C_1^2 \cap C_2^2 \cap C_3^2) \cup C_1^3$.

A single *Policy Condition* evaluates to *true* when the *Policy Variable* and the *Policy Value* match a particular traffic. If the whole CNF/DNF expression of *Policy Conditions* evaluates to *true*, the corresponding *Policy Actions* are applied. Given that any set of conditions expressed in CNF can be converted to DNF and vice versa, the presented prototype automatically converts CNF conditions to DNF because DNF clauses can be directly mapped to flow rules. In his way, each clause corresponds to a flow rule, where each condition maps to a parameter of the traffic to match. A limitation of the prototype is that it does not support negated *Conditions,* either in CNF nor in DNF format.

2) Policy Lifecycle

The Policy Lifecycle manages the state of the network policies through three validation steps: formal, context and conflict validation. ***Formal Validation*** analyzes whether the policy structure is valid. The analysis is specific to the policy type; however, a structured way follows:

1. Each of the *Policy Conditions* and *Policy Actions* must be defined by a supported *Policy Variable*.
2. The *Policy Value* associated with a *Policy Variable* must be consistent. For example, if a *Policy Condition* has "src_ip" as a *Policy Variable*, its *Policy Value* must have the format of an IP address.
3. The dependency requirements among *Policy Variables* must be met. In some cases, several *Policy Variables* need to be defined at the same time, or the other way around, they cannot be defined at the same time. For example, to allow the communication between two hosts, the policy must define the pair of addresses belonging to these hosts.
4. The specific requirements of the particular policy type must be met. Each policy type can define its custom *Formal Validation* requirements.

Context validation analyzes if the resources involved in the policy are available. This analysis also depends on the type of policy. However, there are 3 distinct scenarios:

1. A resource must exist in the network: For example, if a policy defines that two hosts must be able to communicate, the addresses provided by the policy must belong to hosts attached to the network.
2. A resource must not exist in the network: For example, if a policy defines that a particular address has to be translated into another one, the new address must not be in use by another host.
3. Context validation is not needed: For example, a firewall policy does not need the resources to exist by the time the policy is pushed.

Formal and context validations only consider a particular policy instance. ***Conflict Validation*** is needed to ensure that this policy instance is not in conflict with already enforced policy instances and; therefore, the behavior of the network is always the expected. In contrast to the first two validations, *Conflict Validation* is generic for all policy types. *Conflict Validation* comprises of two steps, *Conflict Identification* and *Conflict Resolution*. *Conflict Identification* is responsible for checking if a new policy instance is in conflict with other existing policy instances. If this is true then the role of *Conflict Resolution* is to decide which policy instance is to prevail. With regards to *Conflict Identification,* two policy instances are considered conflicting when: (1) They are of the **same type**. Enabling cross-policy type conflict identification is a future work. (2) They have **dependent conditions**. A sufficient condition to check if two sets of conditions are independent is to check if they share at least one *Policy Variable* that has a different *Policy Value*. Thus, two sets of DNF conditions are dependent if any pair of DNF clauses is not independent. (3) They define **different actions**.

If a conflict has been identified, then *Policy Resolution* is called by the framework. In order to resolve the conflict, the framework utilizes the priorities assigned to each of the conflicting policy instances. Given a conflict, one of the following scenarios can take place.

1. The new policy instance has either equal or lower priority compared to the existing policy instances. In this case the existing policy instances remain in the network and the new one is not enforced.
2. The new policy instance is of higher priority. Here, the existing policy instances are removed from the network and the new policy instance is enforced.

Depending on the resolution of these three validations, a policy can be in any of the following six states:

- **New**: when a policy is pushed to the framework and has a valid JSON structure it is added in the New state.
- **Formally validated**: the policy definition is correct but it has not been yet context validated.
- **Context validated**: the policy has been formally validated and the involved resources are available.
- **Enforced**: the Policy Actions are applied in the network for the traffic that matches the Policy Conditions.
- **Pending**: a formally validated policy, where either the context or the conflict validations failed. Whenever network conditions change or policies are added/removed from the framework, all policies in pending state will be re-evaluated (starting with the *Context Validation*)
- **Removed**: a policy that is not formally valid or that has been manually removed, is in the Removed state.

The logic that runs the different validation steps and handles the resulting state of the Policy Lifecycle is depicted in Figure 2. When a policy instance is pushed, the framework starts treating it in the *New* state. Then, the policy goes through a formal validation process. If it does not pass this validation, the policy is moved to the *Removed* state (1), but if the structure of the policy is valid, it is moved to the *Formally Validated* state (2). Then, the policy goes through a context validation process. If any of the resources of the policy is not available, the policy is moved to the *Pending State* (4). If it passes this validation, the policy is moved to the *Context Validated* state (3). Then, conflicting policies are analyzed. In case there is any conflict, the policy is also moved to the *Pending* state (6), whereas if it is conflict validated it is moved to the *Enforced* state (5). Only in this case, the policy is enforced in the underlying system. Moreover, an administrator can request to deactivate an *Enforced* policy (9) (i.e., move it to the *Pending* state), as well as to remove a *Pending* (8) or *Enforced* (10) policy (i.e., move it to the *Removed* state), at any time. If an *Enforced* policy is either removed or deactivated, it is also un-deployed from the system. Finally, a policy in *Pending* state tries to change its state (7) in two different scenarios, depending on how they reached the *Pending* state:

- **Validation failure** (4) or (6): these policies are attempted to be enforced every time there is a change in the framework (e.g. policy is removed).
- **Manual deactivation** (9): these policies can only be enforced if they are manually reactivated.

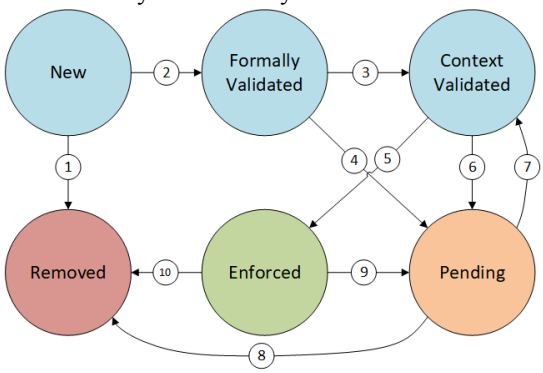

Fig. 2. Policy lifecycle within the policy framework.

B. Implementation

In order to implement and validate the presented policy framework, a prototype has been developed. This prototype consists of the following blocks:

- **Policy Manager (PM)**: An ONOS application that stores the policies and handles their lifecycle. Moreover, it also acts as the point of contact for network administrators through a REST API.
- **Policy Types (PTs)**: Each policy type is implemented as an individual ONOS application, which implements the formal and context validations as well as the enforcement and removal of the corresponding flow rules from the network.

This disaggregation between the PM and the PTs offers two advantages over a monolithic approach (everything as a single ONOS application):

1. When a PT has to be added, updated or removed, the PM does not have to be undeployed, then updated and

redeployed. In this way, each PT is independent from each other and; therefore, there is no downtime for the policy framework as a whole.

2. Developers are not required to have in-depth knowledge about the internals of the PM. A PT can be developed, maintained and installed as an individual component and it only has to implement the aforementioned functionalities (policy model and lifecycle).

Figure 3 illustrates this design in detail. Both the PM and the PTs expose a REST API, which have different purposes. The former is a North-bound interface, meant for the interaction between the administrator of the system and the PM. This API allows CRUD operations for policy instances. The latter is an internal interface that is used by the PM when the functions implemented in a particular PT have to be called. With this approach, all generic actions/requests (e.g. request for a policy status) can be addressed to the PM, hence simplifying and generalizing the implementation of new policy types.

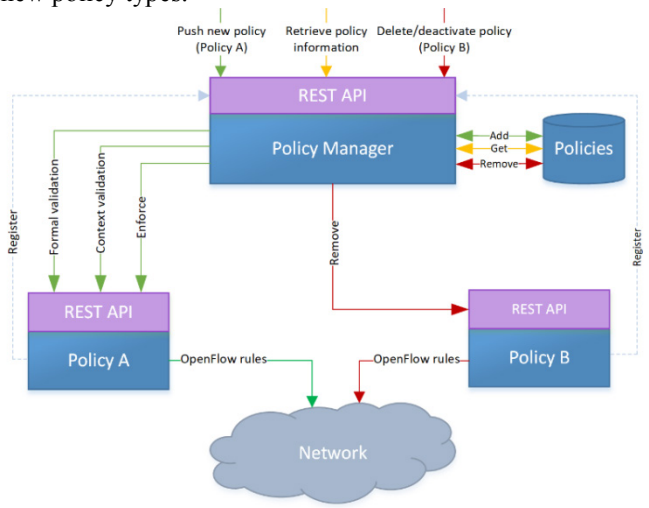

Fig. 3. Network policy framework prototype architecture

The GET requests are used to retrieve information about the policies installed in the framework anddo not involve communication with the PTs. The POST request is used to push new policies to the framework in JSON format. Since this requires the validation and enforcement of the policies, this command will trigger POST requests to the REST API of the corresponding PT. Two PUT requests are available. One is used to register a PT in the PM and the other one is used to modify the priority of a policy. This process implies that this policy has to be removed from the network and pushed again with the new priority. For this reason, this endpoint also involves the validation and enforcement of a policy and; thus, the communication with the corresponding PT. Finally, DELETE requests are used to remove or deactivate a policy from the system, or to remove a PT. All of these endpoints return error/success messages. If an error occurs, not only the type of validation error is returned, but also specific details about the error so that it can be easily fixed. Figure 4 depicts the internal operations of the PF, with respect to the interactions between the different entities. First, the installation of the PM is shown in red. Since it is a native ONOS application, the commands for installation and activation are sent to the ONOS core. A similar process is drawn in green for the installation of a PT. However, this process also includes the registration process of the PT in the PM so it can be used in the future. The process in purple shows the policy instantiation process. A policy instance is pushed by the network administrator, it is validated, and then a set of OF rules are installed in the network. Finally, the id assigned to the policy instance is returned to the network administrator for administration purposes. The process of policy instance removal is shown in blue. The network administrator requests to delete a policy instance with a specific id, and then the remove endpoint of the PT is used to purge the corresponding OF rules from the network. The process in orange corresponds to the PT removal, where the network administrator communicates with ONOS core to uninstall the PT. This triggers the deregistration of that PT from the PM, which eventually removes all the policy instances and flow rules corresponding to that PT from the PM and the underlying network. Finally, the yellow process corresponds to the PM removal, which uninstalls the PM and removes all the flow rules, associated with the PF, from the network devices.

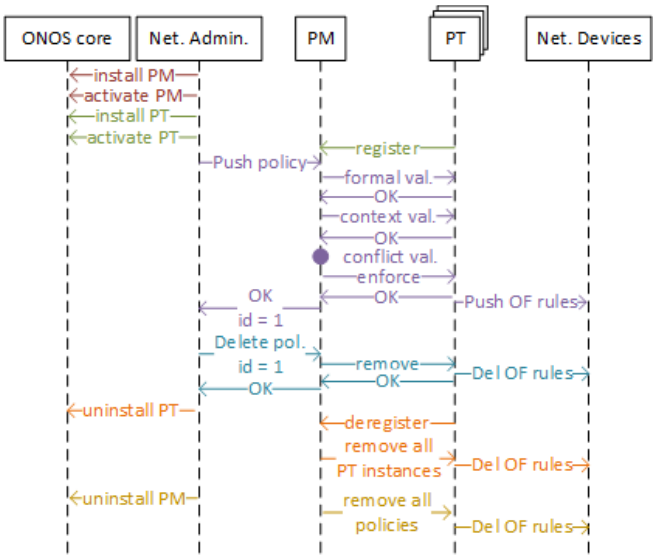

Fig. 4: Internal Operation of the NPF

IV. Policy Types

In order to implement a PT, an ONOS app with a REST API that contains the endpoints for formal validation, context validation, enforcement and removal of policies, has to be created. Each of these endpoints receives HTTP POST requests with the policy in JSON format in the body and return a JSON containing a success/error message. On the one hand, the validation endpoints perform a high-level validation of the policy following the procedure explained in the Policy Lifecycle section of this paper. On the other hand, for the enforcement and removal endpoints, the OpenFlow semantics is used to characterize the flow rules to be pushed in the network switches, although other southbound protocols could be used. When a PT is installed and activated in ONOS, a REST request is issued to the */policytype/register* endpoint. In this way, the PM is notified that a new PT has been added and can call its corresponding endpoints. Similarly, when a PT is uninstalled or deactivated, a REST request is issued to the */policytype/deregister*. This will remove all the flow rules of this policy type from the PM and will prevent new policies

of that type to be enforced. As a PoC, three different types of policies are currently implemented: a firewall policy, a network address translation (NAT) policy and a connectivity policy. These PTs are simple in nature since the scope of this paper is not to create novel policy types, but to use them as a validation for the PF. This section describes the each policy type, how they get validated, enforced and removed.

A. Firewall policy

This policy type implements a simple packet-filtering firewall, which allows a network administrator to block traffic that matches a set of specified parameters. On the one hand, the supported parameters by this policy are the source MAC address, destination MAC address, source IP address, destination IP address, source port, destination port, transport protocol and device. On the other hand, this policy type just allows one action: to block the traffic. This policy implements its four REST endpoints in the following way: (1) **Formal validation**: checks that the provided conditions and actions are supported by the policy type, and their values have the appropriate format. (2) **Context validation**: it is always valid. Firewalls do not need the resources (hosts) to be available when defining a rule. (3) **Enforcement**: based on the Policy Conditions and the Policy Action, the matching criteria and the corresponding treatment instructions are defined, composing a set of flow rules. Each of these flow rules has an id based on the policy id that generated them. (4) **Removal**: all the flow rules with an id based on the given policy id are removed.

B. NAT policy

This policy type implements a simple network address translation, which allows a network administrator to specify the address a host will have when communicating with hosts in another network. To this end, it supports one condition, the IP of a host, and one action, the translated IP. This policy implements its four REST endpoints in the following way: (1) **Formal validation**: checks that the provided conditions and actions are supported, and their values have the appropriate format. (2) **Context validation**: the provided host IP must exist, and the provided translated IP must not be being used by other hosts. (3) **Enforcement**: A reactive packet processor handles the ARP messages for the translated IP address, keeps the translation tables updated and installs the appropriate flow rules. (4) **Removal**: the host is removed from the translation tables.

C. Connectivity policy

This policy allows controlling the connectivity between a pair of hosts. For this reason, it supports two conditions, the source and destination IP or MAC addresses, and one action, to connect both hosts. This policy implements its four REST endpoints in the following way: (1) ***Formal validation***: a part from valid Policy Conditions and Policy Actions, a pair of source and destination hosts must be provided. (2) ***Context validation***: both the source and destination hosts must exist in the network. (3) ***Enforcement***: based on the provided P*olicy Conditions, a host-to-host intent is created, which is responsible for obtaining the path between the* hosts and installing the flow rules*. Moreover, if the path between the two hosts breaks, the intent is recompiled and a new path is provided.* (4) **Removal**: the intent is withdrawn from the network.

V. CONCLUSIONS AND FUTURE WORK

This paper presented a policy framework prototype that enables the management of SDN network policies for the ONOS SDN controller. The original design of this policy framework presented in [1] has been extended in this paper by enhancing the lifecycle that manages the policies state and disaggregating the core of the framework from the specific policies. In this paper, the presented prototype has been implemented as a set of ONOS applications: a Policy Manager, responsible for managing the state of the policies and to make sure they are not conflicting, and a set of Policy Types, which are independent of the Policy Manager and describe how a policy has to be validated and enforced in the network. Three network policies (firewall, NAT and connectivity) have already been implemented and validated. The next steps of this work are to develop a web-based dashboard that allows the administrator to monitor the state or parameters of existing policies, as well as to manage the addition and deletion of policies into the framework. Moreover, new policy types (e.g., bandwidth and latency policies) can be implemented to further validate the policy framework. Finally, cross-policy-type conflict resolution could also be considered for future implementation. A possible approach would be to perform conflict validation in the level of OF rules, instead of the policy level. Doing so removes any context attached to the specific policy types, allowing treating them as objects of the same type.

ACKNOWLEDGMENT

This work has been performed in the framework of the NGPaaS project, funded by the European Commission under the Horizon 2020 and 5G-PPP Phase2 programmes, under Grant Agreement No. 761 557 (http://ngpaas.eu).

REFERENCES

[1] A. Mimidis, E. Ollora, and J. Soler. "Policy Framework for the Next Generation Platform as a Service." 27th European Conference on Networks and Communications. IEEE, 2018.
[2] "ONOS." [Online]. Available: http://onosproject.org/.
[3] Wiki.onap.org. (2018). The ONAP Policy Framework - Developer Wiki - Confluence. [online] Available at: https://wiki.onap.org/display/DW/The+ONAP+Policy+Framework [Accessed Aug. 2018].
[4] Intent Framework - ONOS - Wiki. [online] Available at: https://wiki.onosproject.org/display/ONOS/Intent+Framework [Accessed Aug. 2018].
[5] Y. Ben-Itzhak, K. Barabash, R. Cohen, A. Levin and E. Raichstein, "EnforSDN: Network policies enforcement with SDN" 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), Ottawa, ON, 2015, pp. 80-88.
[6] Callegati, Franco, et al. "Performance of intent-based virtualized network infrastructure management." Communications (ICC), 2017 IEEE International Conference on. IEEE, 2017.
[7] B. Moore, E. Ellesson, and A. Westerinen, "Policy Core Information Model -- Version 1 Specification,", RFC 3060, 2001.
[8] Moore, E. "Policy Core Information Model-Extensions. IETF Request for comments (RFC 3460)(January 2003)."
[9] Pisharody, Sandeep, et al. "Brew: A security policy analysis framework for distributed SDN-based cloud environments." IEEE Transactions on Dependable and Secure Computing (2017).