Rules Placement Problem in OpenFlow Networks: a Survey

Xuan Nam Nguyen, Damien Saucez, Chadi Barakat, Thierry Turletti

Rules Placement Problem in OpenFlow Networks: a Survey

Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, Thierry Turletti
INRIA Sophia Antipolis, France

***Abstract*—Software-Defined Networking (SDN) abstracts low-level network functionalities to simplify network management and reduce costs. The OpenFlow protocol implements the SDN concept by abstracting network communications as flows to be processed by network elements. In OpenFlow, the high-level policies are translated into network primitives called rules that are distributed over the network. While the abstraction offered by OpenFlow allows to potentially implement any policy, it raises the new question of how to define the rules and where to place them in the network while respecting all technical and administrative requirements. In this paper, we propose a comprehensive study of the so-called *OpenFlow rules placement problem* with a survey of the various proposals intending to solve it. Our study is multi-fold. First, we define the problem and its challenges. Second, we overview the large number of solutions proposed, with a clear distinction between solutions focusing on memory management and those proposing to reduce signaling traffic to ensure scalability. Finally, we discuss potential research directions around the OpenFlow rules placement problem.**

***Index Terms*—Software-Defined Networking, OpenFlow, rules placement, survey**

I. Introduction

Computer networks today consist of many heterogeneous devices (e.g., switches, routers, middleboxes) from different vendors, with a variety of sophisticated and distributed protocols running on them. Network operators are responsible for configuring policies to respond to a wide range of network events and applications. Normally operators have to manually transform these high level policies into low level vendor specific instructions, while adjusting them according to changes in network state. As a result, network management and performance tuning are often complicated, error-prone and time-consuming. The main reason is the tight coupling of network devices with the proprietary software controlling them, thus making it difficult for operators to innovate and specify high-level policies [1].

Software-Defined Networking (SDN) advocates the separation between forwarding devices and the software controlling them in order to break the dependency on a particular equipment constructor and to simplify network management. In particular, OpenFlow implements a part of the SDN concept through a simple but powerful protocol that abstracts network communications in the form of flows to be processed by intermediate network equipments with a minimum set of primitives [1].

OpenFlow offers many new perspectives to network operators and opens a plethora of research questions such as how to design network programming languages, obtain robust systems with centralized management, control traffic at the packet level, perform network virtualization, or even co-exist with traditional network protocols [2], [3], [4], [5], [6], [7], [8]. For all these questions, finding how to allocate rules such that high-level policies are satisfied while respecting all the constraints imposed by the network is essential. The challenge being that while potentially many rules are required for traffic management purpose [9], in practice, only a limited amount of resources, and in particular memory [10], is available on OpenFlow switches. In this paper, we survey the fundamental problem when OpenFlow is used in production networks, that we refer to it as the *OpenFlow rules placement problem*. We focus on OpenFlow as it is the most popular southbound SDN interface that has been deployed in production networks [11].

The contributions of this paper include:

- A generalization of the OpenFlow rules placement problem and an identification of its main challenges involved.
- A presentation, classification, and comparison of existing solutions proposed to address the OpenFlow rules placement problem.
- A discussion of potential directions around the OpenFlow rules placement problem.

The paper is organized as follows. In Sec. II, we provide a background on SDN and OpenFlow. In Sec. III, we formalize the OpenFlow rules placement problem, discuss the challenges and illustrate them through different use cases. We continue with existing ideas that address the two main challenges of the OpenFlow rules placement problem: memory limitation in Sec. IV, and signaling overhead in Sec. V. We finally discuss potential research directions in Sec. VI and conclude in Sec. VII.

II. Background

Software-Defined Networking (SDN) is a new networking paradigm in which the control plane (i.e., how to handle traffic) is separated from the data plane (i.e., how to forward packets according to decisions from the control plane). OpenFlow [1], depicted in Fig. 1, is the most popular implementation of the southbound interface of SDN [5], [6], [7], [11], [12].

In OpenFlow, forwarding devices are called OpenFlow switches and all forwarding decisions are flow-based instead of destination-based like in traditional routers. An OpenFlow switch consists of flow tables, each containing a prioritized list of rules that determine how packets have to be processed

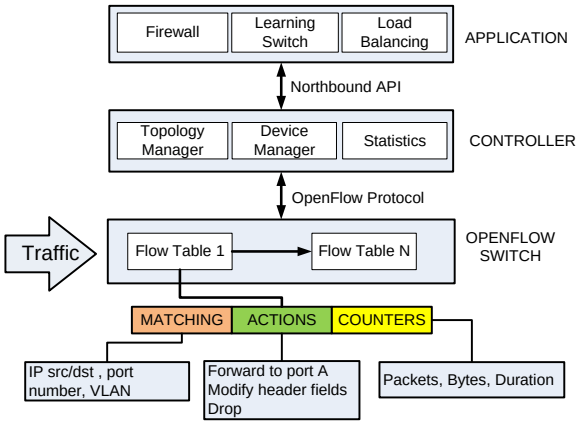

Fig. 1. OpenFlow Architecture. Network control functions are separated from the forwarding devices (i.e., OpenFlow switches) and moved to an entity called the controller. The controller manages the network using the OpenFlow protocol and exposes a northbound Application Program Interface (API) for the operator to write applications for the automation of management tasks, such as load balancing.

by the switch.[1] A rule consists of three main components: a *matching pattern*, an *actions field* and a *counter*. The matching pattern forms a predicate whose value is computed on-the-fly by the switch based on packet meta-information (e.g., source IP address). All packets making true the matching pattern predicate are said to belong to the same *flow*. The actions specified in the actions field of the rule are applied to every packet of the corresponding flow. The most common actions are: *forwarding*, *dropping*, or *rewriting* the packets. Finally, the counter is used to count the number of packets that has been processed (i.e., that made the predicate hold true) along with the lifetime of this rule. As a packet may match multiple matching patterns, each rule is associated with a priority and only the rule with the highest priority that matches the packet is considered to take actions on it. The prioritization of rules permits constructing default actions that can be applied on packets only if no other rule can be used. Examples of default actions are dropping packets, forwarding packets to a default interface, or even to the controller. For efficiency and flexibility reasons, the latest versions of OpenFlow [13] support pipeline processing where a packet might be processed by several rules from different flow tables.

The control plane of OpenFlow is implemented with a *controller*. In general, the controller is logically centralized, but this does not prevent it to be physically distributed for scalability and fault tolerance reasons. The controller maintains a global view of the network with its state at any instant and updates flow tables in switches using the OpenFlow protocol. Many OpenFlow controller platforms are available to handle connections with OpenFlow switches and to provide basic functions like routing, flow table management, and topology discovery [14], [15], [16], [17], [18]. However, most of controller platforms still force operators to manage their network at the level of individual switches, by selecting and populating flow tables to satisfy network constraints [19].

[1]In this paper we follow the OpenFlow model terminology where a packet consists of any piece of information traveling through the network and a switch stands for any device processing such a packet.

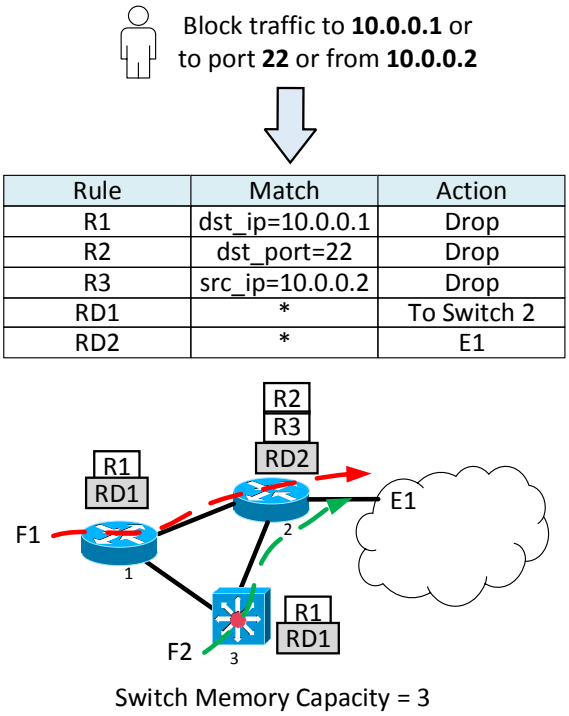

Fig. 2. An example of access control rules placement. The firewall policy is transformed into a list of rules $R1, R2, R3$ used to block matching packets and two default rules $RD1, RD2$ for forwarding non-matching packets towards endpoint E. Then, these rules are distributed on several switches to ensure that flows F1 and F2 pass through all rules R1, R2, R3 to satisfy the policy.

For the sake of simplicity, we summarize the terminology and the notations used in this survey in Table I and Table II.

III. OpenFlow Rules Placement Problem

A. Motivation

OpenFlow facilitates network management by raising the level of network abstraction ([2], [3], [4], [19], [20]). High level of abstraction hides the complexity of the network devices, and exposes a simple interface to the operators. As a result, the operator models the network as a single big switch, a blackbox, and no longer needs to care about low-level details, such as how to configure devices and manage resources. Therefore, network management becomes simpler and more robust, which reduces the risk of errors and provides a higher degree of freedom in management.

A layer between the operators and the networks is required to transform the high-level policies from operators into low-level rules to be installed on OpenFlow switches. Therefore, it is necessary to solve the OpenFlow rules placement problem that consists in selecting the most relevant rules to place on each switch so that both the high-level policies and the network constraints are satisfied. A rule placement solution defines which rules must be deployed in the network and where.

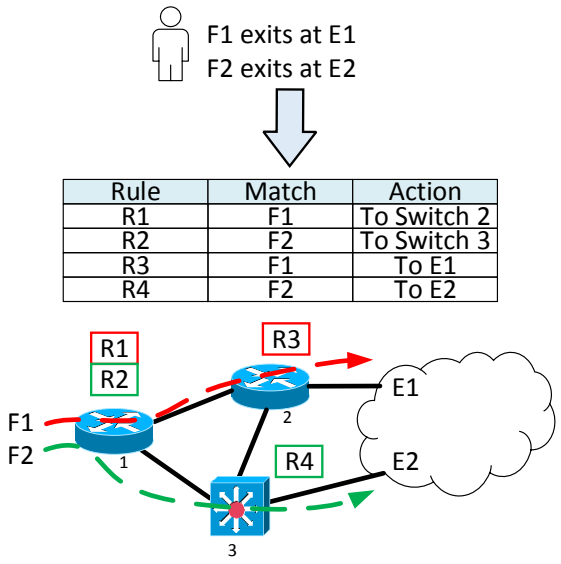

Fig. 3. An example of forwarding rules placement. Forwarding rules are installed on appropriate paths to make sure that the endpoint policy is satisfied. Rules R1, R3 (R2, R4) are installed on switches 1,2 (1, 3) to route F1 (F2) towards its endpoint E1 (E2).

Many applications can benefit from OpenFlow, for example, monitoring, accounting, traffic shaping, routing, access control and load balancing [1]. Each of these applications requires a dedicated rules placement solution. In multi-tenant clouds, each user has different policies (e.g., firewall policy), which also requires a dedicated rules placement.

In the following, we describe two representative scenarios, that motivate the need to solve OpenFlow rules placement problem:

a) Access Control: As a part of endpoint policy, the firewall policy is critical to the network security. Most of firewall policies can be defined as a list of prioritized access control rules, that specify which flows are authorized and where. OpenFlow is a potential candidate to implement the access control policy, due to its flexible matching patterns and actions.

Ideally, all access control rules should be placed on the ingress switches to limit unwanted network traffic. However, the switch memory constraints prohibit usually placing all rules in ingress switches. An alternative solution is to put all rules in the software switches that have large memory capacity and to direct all traffic to them. However, software switches are generally slower than hardware-accelerated solutions. Therefore, a solution is required to distribute rules over network such that the semantic of the original access control list is preserved and resource constraints are satisfied.

Fig. 2 shows an example of access control rules placement. The firewall policy must be applied on all flows originated from switch 1 and 3. A solution is to use rules $R1, R2, R3$ for blocking matching packets and use the default rules $RD1, RD2$ for forwarding non-matching packets towards endpoint E. Then, these rules are distributed on the switches, according to the memory capacity, to enforce firewall policy on all flows.

b) Traffic engineering: The role of the network is to deliver packets towards its destinations and to satisfy the operator's requirements (e.g., low latency, low loss rate). OpenFlow allows defining rules matching any type of packets and forwarding them on any paths.

Normally, the forwarding rules that match the packets should be placed on the shortest paths to forward them toward their endpoints. However due to memory limitation of the OpenFlow switch, all rules to be installed might not be fit into the shortest paths. Therefore, it is important to select the right paths and to make extensive usage of wildcard rules to satisfy requirements.

An example of forwarding rules placement is shown in Fig. 3. Forwarding rules are required to be installed on appropriate paths to make sure that the endpoint policy for $F1, F2$ is satisfied. To that aim, a solution is to install forwarding rules $R1, R3$ (resp. $R2, R4$) on switches $1, 2$ (resp. $1, 3$) to route $F1$ (resp. $F2$) towards its endpoint $E1$ (resp. $E2$).

B. Problem Formalization

In the following, we formalize the OpenFlow rules placement problem using the notations in Table II.

TABLE II
NOTATIONS USED IN THIS PAPER

Notation	Definition
V	set of OpenFlow nodes
E	set of links
O	set of endpoints (e.g., peering links)
F	set of flows (e.g., source – destination IP flows)
R	set of possible rules for selection
T	set of time values
$FT(v,t)$	flow tables of node v at time $t \in T$
C_i	memory capacity of node i (e.g., in total number of rules)
P	set of possible paths to the endpoints (e.g., shortest paths)
m	matching pattern field (e.g. $srcIP = 10.0.0.*$)
a	actions field (e.g. dropping the packets)
q	priority number (0 to 65535)
t_{idle}	idle timeout (s)
t_{hard}	hard timeout (s)
EP	endpoint policy, defines the endpoint(s) $o \in O$ of $f \in F$
RP	routing policy, defines the path(s) $p \in P$ of $f \in F$

The network is modeled as a directed graph $G = (V, E)$, where V is the set of nodes and each node $v \in V$ can store C_v rules, E is the set of links. O is the set of endpoints where the flow used to exit the network (e.g., peering links, gateways, firewalls). A flow can have many endpoints $o_f \in O$. P is the set of possible paths that flows can use to reach their endpoints $o_f \in O$. Each path $p \in P$, consists of a sequence of nodes $v \in V$. F, R are the set of flows and rules for selection, respectively.

The **output** of the problem is the content of the flow table of node $FT(v,t) = [r_1, r_2, ...] \subset R$, which defines the set of rules required to install node $v \in V$ at time $t \in T$. $T = [t_1, t_2, ...]$ is set of the time instants at which $FT(v,t)$ is computed and remains unchanged during the period $[t_i, t_{i+1}]$. Each rule r_j is defined as a tuple, which contains values for matching pattern m, actions a, priority number q and timeouts

t_{idle}, t_{hard}, selected by the solvers.[2] The flow table content of all nodes $FT(v,t), \forall v \in V$ at a time t is defined as a *rules placement* solution. Furthermore, $FT(v,t)$ changes over time t to adapt with network changes (e.g., topology changes, traffic fluctuation). In order to construct rules placement, the following **inputs** are considered:

- **Traffic flows** F, which stand for the network traffic. The definition of a flow, implemented with the matching pattern, depends on the granularity needed to implement the operator policies. For example, network traffic can be modeled as set of Source-Destination (SD) flows, each flow is a sequence of packets having the same source and destination IP address.
- **Policies**, which are defined by the operator, can be classified into two categories: (i) the *end-point policy* $EP : F \rightarrow O$ that defines where to ultimately deliver packets (e.g. the cheapest link) and (ii) the *routing policy* $RP : F \rightarrow P$ that indicates the paths that flows must follow before being delivered (e.g. the shortest path) [19]. The definition of these policies is often the result of the combination of objectives such as cost reduction, QoS requirements and energy efficiency [21], [20], [22], [19], [23].
- **Rule space** R, which defines the set of all possible rules for selection, depending on applications. For example, an access control application allows selecting rules that contain matching m for 5-tuples IP fields (source/destination IP address, source/destination port number, protocol number) while a load balancing application requires rules that contain matching m for source/destination IP address [24]. The combination of fields and values forms a large space for selection.
- **Resource constraints**, such as memory, bandwidth, CPU capacity of the controller and nodes. Rules placement solutions must satisfy these resource constraints. As an example, the total number of rules on a node should not exceed the memory capacity of the nodes: $|FT(v,t)| \leq C_v, \forall (v,t) \in V \times T$.

There might be a countless number of rules placement possibilities that satisfy the above inputs. Therefore, $FT(v,t)$ is usually selected based on additional requirements, such as in order to minimize the overall rule space consumption $\sum_{v \in V} |FT(v,t)|$. Note that in general, the OpenFlow rules placement problem is NP-hard [25], [20].

C. Challenges

Elaborating an efficient rules placement algorithm is challenging due to the following reasons.

1) Resource limitations: In most of production environments, a large number of rules is required to support policies whereas network resources (e.g., memory) are unfortunately limited. For example, up to 8 millions of rules are required in typical enterprise networks [26] and up to one billion for the management of tasks in the cloud [9]. According to Curtis et al. [27], a Top-of-Rack switch in data centers may need 78,000 rules to accommodate traffic.

While the number of rules needed can be very large, the memory capacity to store rules is rather small. Typically, OpenFlow flow tables are implemented using Ternary Content Addressable Memory (TCAM) on switches to ensure matching flexibility and high lookup performance. However, TCAM is board-space costly, is 400 times more expensive and consumes 100 times more power per Mbps than RAM-based storage [28]. Also, the size of each flow entry is 356 bits [13], which is much larger than the 60-bit entries used in conventional switches. As a consequence, today commercial off-the-shelf switches support only from 2k to 20k rules [10], which is several orders of magnitude smaller than the total number of rules needed to operate networks. Kobayashi et al. [12] confirm that the flow table size of commercial switches is an issue when deploying OpenFlow in production environments.

Recently, software switches built on commodity servers (e.g., Open vSwitch [29]) are becoming popular. Such switches have large flow table capacity and can process packets at high rate (e.g., 40 Gbps on a quad-core machine [28]). However, software switches are more limited in forwarding and lookup rate than commodity switches [30] for two main reasons. Firstly, software switches use general purpose CPU for forwarding, whereas commodity switches use Application-Specific Integrated Circuits (ASICs) designed for high speed throughput. Secondly, rules in software switches are stored in the computer Random Access Memory (RAM), which is cheaper and larger, while rules in commodity switches are stored in TCAM, which allows faster lookup but has limited size. For example, an 8-core PC supports forwarding capacities of 4.9 millions packets/s, while modern switches using TCAMs do forwarding at a rate up to 200 millions packets/s [31].

To accelerate switching operations in software switches, flow tables can be stored in CPU caches. Nevertheless, these caches are rather small, which brings the same problem than with ASICs.

In Sec. IV, we extensively survey the techniques proposed in the literature to cope with the memory limitation in the context of the OpenFlow rules placement problem.

2) Signaling overhead: Installing or updating rules for flows triggers the exchange of OpenFlow messages. Bad rules placement solutions might also cause frequent flow table misses that would require controller to act. While the number of messages per flow is of the order of magnitude of the network diameter, the overall number of messages to be exchanged may become large. For instance, in a data center with 100,000 new flows per second [32], at least 14 Gbps of overall control channel traffic is required [33]. Comparably, in dynamic environments, rules need to be updated frequently (e.g., routing rules may change every 1.5s to 5s [9] and forwarding rules can be updated hundreds times per second [34]).

In situations with large signaling load, the controller or switches might be overloaded, resulting in the drop of signaling messages and consequently in potential policy violations, blackhole, or forwarding loops. High signaling load also impacts the CAPEX as it implies investment in powerful

[2]We focus on important fields only. The complete list of fields of an OpenFlow rule can be found in the OpenFlow specifications [13].

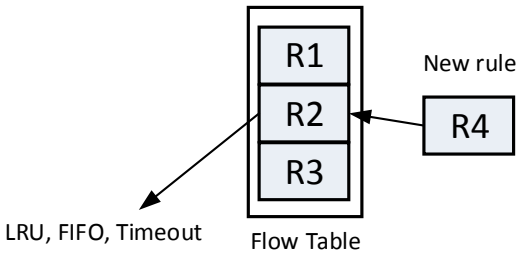

Fig. 4. An example of eviction. Rule $R2$ in the flow table is reactively evicted using replacement algorithms (e.g., LRU, FIFO) when the flow table is full and a new rule $R4$ needs to be inserted. $R2$ can also be proactively evicted using, for example, a timeout mechanism.

hardware to support the load.

We provide a comprehensive survey of OpenFlow rules placement problem solutions that deal with signaling overhead in OpenFlow in Sec. V.

IV. Efficient Memory Management

As explained in Sec. III-C, all required rules might not fit into the flow table of a switch because of memory limitations. In this section, we classify the different solutions proposed in the literature to manage the switch memory into three categories. In Sec. IV-A, we detail solutions relying on *eviction* techniques. The idea of eviction is to remove entries from a flow table before installing new entries. Afterwards, in Sec. IV-B, we describe the techniques relying on *compression*. In OpenFlow, compressing rules corresponds to building flow tables that are as compact as possible by leveraging redundancy of information between the different rules. Then, in Sec. IV-C, we explain the techniques following the *split and distribution* concept. In this case, switches constitute a global distributed system, where switches are inter-dependent instead of being independent from each other. Finally, we provide in Table III a classification of the related work and corresponding memory management techniques.

A. Eviction

Because of memory limitation, the flow table of a switch may be filled up quickly in presence of large number of flows. In this case, eviction mechanisms can be used to recover the memory occupied by inactive or less important rules to be able to insert new rules. Fig. 4 shows an example where the flow table is full and new rule $R4$ needs to be inserted. In this case, rule $R2$ in the flow table is evicted using replacement algorithms (e.g., LRU, FIFO). $R2$ can also be proactively removed using OpenFlow timeout mechanism.

The main challenge in using eviction is to identify high value rules to keep and to remove inactive or least used rules.

Existing eviction techniques have been proposed such as *Replacement algorithms* (Sec. IV-A1), *Flow state-based eviction*(Sec. IV-A2) and *Timeout mechanisms* (Sec. IV-A3).

1) Replacement algorithms: Well-known caching replacement algorithms such as Least Recent Used (LRU), First-In First-Out (FIFO) or Random replacement can be implemented directly in OpenFlow switches. Replacement algorithms are performed based on lifetime and importance of rules, and is enabled by setting the corresponding flags in OpenFlow switches configuration. As eviction is an optional feature, some OpenFlow switches may not support it [13]. If the corresponding flags are not set and when the flow table is full, the switch returns an error message when the controller tries to insert a rule.

Replacement algorithms can also be implemented by using the controller delete messages (OFPFC_DELETE). If the flag OFPFF_SEND_FLOW_REM is set when the rule is installed, on rule removal, the switch returns a message containing the removal reason (e.g., timeout) and the statistics (e.g., flow duration, number of packets) at the removal time to the controller [13]. From OpenFlow 1.4, the controller can get early warning about the current flow table occupation and react in advance to avoid flow table full [35]. The desired warning threshold is defined by the controller.

Vishnoi et al. [36] argue that replacement algorithms are not suitable for OpenFlow. First, implementing replacement algorithms on the switch side violates one of the OpenFlow principles, which is to delegate all intelligence to the controller. Second, implementing replacement algorithms at the controller side is unfeasible because of large signaling overhead (e.g., statistic collections and delete messages).

Among replacement algorithms, LRU outperforms others and improves flow table hit ratio, by keeping recently used rules in flow table, according to studies [37], [35]. However, the abundance of mice flow in data center traffic can cause elephant flows' rules to be evicted from the flow table [38]. Therefore, replacement algorithms need to be designed to favor frequent used rules.

2) Flow state-based Eviction: Flows vary in duration and size [32]. Therefore, flow state information can also be used to early evict rules before their actual expiration [37], [39], [40]. For example, based on observation of flow packet's flags (e.g., TCP FIN flag), the controller can decide to remove the rule used for that flow by sending delete messages. However, eviction algorithms relying on flow state can be expensive and not easy to implement, because of large signaling overhead [37].

3) Timeout mechanisms: Rules in flow tables can also be *proactively evicted* after a fixed amount of time (*hard_timeout*) t_{hard} or after some period of inactivity (*idle_timeout*) t_{idle} using the timeout mechanism in OpenFlow switches [13], if these values are set when the controller installs rules.

Previous controllers have assigned static idle timeout values ranging from $5s$ in NOX [14], to $10s$ and $60s$ in DevoFlow [27]. Zarek et al. [37] study different traces from different networks and observe that the optimal *idle_timeout* value is $5s$ for data centers, $9s$ for enterprise networks, and $11s$ for core networks.

Flows can vary widely in their duration [32], so setting the same timeout value for all rules may lead to inefficient memory usage for short lifetime flows. Therefore, adaptive timeout mechanisms [36], [41], [40], [42], [43] have been proposed. In

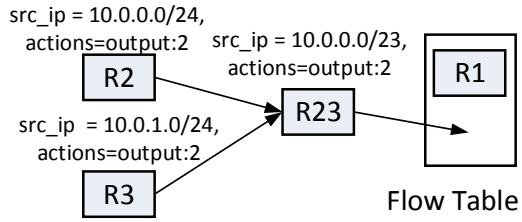

Fig. 5. An example of compression. $R2$ and $R3$ have the same actions field and are compressed into a single rule $R23$ that has matching pattern covering both matching patterns of $R2$ and $R3$. Thus, rule space consumption is reduced while the original semantic is preserved.

these studies, the timeout value is chosen and adjusted based on flow state, controller capacity, current memory utilization, or switch location in the network. These approaches lead to better memory utilization and do not require the controller to explicitly delete entries. However, obtaining an accurate knowledge about flow state is usually expensive as it requires a large signaling overhead to monitor and collect statistics at high frequency.

In the original scheme of OpenFlow, when a packet matches a rule, the idle timeout counter of the rule is reset but the gain is limited [41]. Therefore, Xie et al. [41] propose that switches should accumulate remaining survival time from the previous round to the current round, so that the rules with high matching probability will be kept in the flow table. Considering the observation that many flows never repeat themselves [32], a small idle timeout value in the range of 10ms – 100 ms is recommended for better efficiency, instead of using the current minimum timeout of 1s [13], [36]. These improvements require modifications in the implementation of OpenFlow.

All of the above studies advocate using idle timeout mechanism, since using hard timeout mechanism may cause rules removal during transmission of burst packets and leads to packet loss, increased latency, or degraded network performance [39].

B. Compression

Compression (or aggregation) is a technique that reduces the number of required rules while preserving the original semantics, by using wildcard rules. As a result, an original list of rules might be replaced by a smaller one that fits the flow table. As an example in Fig. 5, $R2$ and $R3$ have the same actions field and are compressed into a single rule $R23$ that has matching pattern covering both matching patterns of $R2$ and $R3$. Thus, rule space consumption is reduced while the original semantic is preserved.

Traditional routing table compression techniques for IP such as ORTC [44] cannot be directly applied to compress OpenFlow rules because of two reasons. First, OpenFlow switches decide which rule will be used based on rule priority number when there are several matching rules. Second, rules may contain multiple actions and multiple matching conditions, not restricted to IP.

The challenge when using compression is to maintain the original semantics, keep an appropriate view of flows and achieve the best tradeoff between compression ratio and computation time. The limitation of this approach is that today not all the OpenFlow matching fields support the wildcard values (e.g., transportation port numbers).

In the following, we discuss the compression techniques used for access control rules in Sec. IV-B1 and forwarding rules in Sec. IV-B2. Compression techniques may reduce flow visibility and also delay network update; therefore, we discuss its shortcoming and possible solutions in Sec. IV-B3.

1) Access control rules compression: Most of firewall policies can be defined as a list of prioritized access control rules. The matching pattern of an access control rule usually contains multiple header fields, while the action field is a binary decision field that indicates to drop or permit the packets matching that pattern. Normally, only rules with *drop* action are considered in the placement problem since rules with *permit* action are complementary [45]. Because the action field is limited to *drop* action, access control rules can be compressed by applying compression techniques on rules matching patterns, to reduce the number of rules required.

To that aim, rule matching patterns are represented in a bit array and organized in a multidimensional space [46], [19], where each dimension represents a header field (e.g., IP source address). Afterwards, heuristics such as Greedy are applied on this data structure to compute optimized wildcard rules. For example, two rule with matching $m_1 = 000$ and $m_2 = 010$ can be replaced by a wildcard rule with $m = 0 * 1$.

Matching patterns usually have dependency relationships. For example, packets matching $m_1 = 000$ also match $m_2 = 00*$, therefore m_2 depends on m_1. When rules with these matching patterns are placed, the conflict between them needs to be resolved. An approach for compression and resolving conflicts is to build a rule dependency graph [28], [45], where each node represents a rule and a directed edge represents the dependency between them. Analyzing this graph makes it possible to compute optimized wildcard rules and to extract the dependency constraints to fetch for their optimization placement model.

The network usually has a network-wide blacklisting policy shared by multiple users, for example, packets from a same IP address are dropped. Therefore, rules across different access control lists from different users can also be merged to further reduce the rule space required [45]. Also, traditional techniques exist to compress access control rules on a single switch [47], [48], [49].

2) Forwarding rules compression: In OpenFlow networks, forwarding rules can be installed to satisfy endpoint and routing policies. A naive approach is to place exact forwarding rules for each flow on the chosen path. However, this can lead to huge memory consumption in presence of large number of flows. Therefore, compression can be applied to reduce the number of rules to install.

Matching pattern of forwarding rules are usually simpler than access control rules, but they have a larger palette of actions (e.g., bounded by the number of switch ports) and they outnumber access control rules by far [50]. In addition,

forwarding rules compression has stricter time constraints than access control rules compression when it comes to satisfying fast rerouting in case of failure.

OpenFlow forwarding rules can be interpreted as logical expressions [50], for example, `('11*', 2)` represents for rules matching prefix `'11*'` and the action field is to forward to port 2. Normally, rules with same forwarding behavior are compressed into one wildcard rule. Also, it is important to resolve conflicts between rules, for example, by assigning higher priority for rule `('11*', 3)` to avoid wrong forwarding caused by rule `('1**', 2)`. To compress and to resolve conflicts, the Expresso heuristic [50] borrowed from logical minimization can be applied to obtain an equivalent but smaller sets, which is then transformed to corresponding rules. Forwarding rules can also be compressed based on source or destination IP address [51].

The routing policy plays an important role in applying compression techniques, as it decides the paths where forwarding rules are placed to direct flows towards their endpoints. Single path routing has been widely used because of its simplicity, however, it is insufficient to satisfy QoS requirement, such as throughput [22]. Hence the adoption of multipath routing. Normally, forwarding rules are duplicated on each path to route flows towards their destinations. By choosing appropriate flow paths such that they transit on the same set of switches, forwarding rules on these switches can be compressed [22]. For example, flow F uses path $P1 = (S1, S2, S3)$ and $P2 = (S3, S2, S4)$ that have Switch $S2$ in common. On the latter switch, two rules that forward F to $S3, S4$ can be compressed into one rule $match(F) \rightarrow select(S3, S4)$. Also, forwarding rules may contain the same source (e.g., ingress port), that can also be compressed [43].

Generally, OpenFlow switches have a default rule with the lowest priority that matches all flows. Forwarding rules can also be compressed with the default rule if they have the same actions (e.g., forwarding with the same interface) [20]. Also, forwarding paths for flows can be chosen such that they leverage the default rules as much as possible. In this way, flows can be delivered to their destinations with the minimum number of forwarding rules.

Even though the actions field of a rule may contain several actions (e.g., encapsulate, then forward), the number of combinations of actions is much less than the number of rules and can thus be represented with few bits (e.g., 6 bits) [52]. Several studies [52], [33] propose to encode the actions for all intermediate nodes in a list. This list is added to the header of each packet (e.g., using VLAN field [33]) by the ingress switch. Afterwards, each intermediate node identifies its actions in the list (e.g., using pop VLAN action) and executes them. Finally, the egress switch recovers the original packet and forwards it to the destination. This idea is similar to IP source routing [53]. This approach allows decreasing significantly the number of forwarding rules in the core nodes, but at the same time, it increases the packet size headers of all packets.

3) Shortcomings of the Compression approach: The compression approach reduces the number of rules required, but it makes flows less visible since the wildcard rule is not used for a single flow and consequently, the controller cannot control a flow (e.g., monitoring, rate limitation) without impacting other flows. In many applications, one rule is required for each flow to ensure flow visibility and controllability [54]. Moreover, finding a rule placement with high compression ratio may require high computation time [55].

First, in many scenarios, full control and visibility over all flows is not the right goal as only some important, significant flows need full control [27]. For example, load balancing requires handling long lived, high throughput flows. According to traffic analysis studies [32], only a few percentages of flows (called elephant flows), send a large number of bytes and the rest of flows, send a small number of bytes. Therefore, wildcard rules [27] or default rules [20] can be used to handle these flows locally on switches and dedicated rules are installed for elephant flows. In this manner, the number of rules required can be reduced, since according to the flow size distribution [32], the number of elephant flows is much smaller than the number of other kinds of flows.

Second, even if each flow requires full control, usually only one exact-matching rule for the flow in the network is needed [33], and on the rest of the flow path wildcard rules are used to handle it. In [33], a solution is proposed to install an exact forwarding rule for the flow at the first switch, which usually consists in a software switch with a high capacity flow table. At intermediate switches, forwarding rules that have the same output actions can be compressed into one rule [52], [33]. Other solutions [56], [34] leverage exact-matching tables (e.g., MAC forwarding tables), beside the wildcard matching flow tables in switches. More precisely, the network is divided into two domains: one where flows are controlled through wildcard rules and the other with exact-matching rules in these tables. The controller computes and defines the best tuning point (i.e., where the flow starts to use exact-matching rules) per flow basis.

Above solutions can reduce the number of forwarding rules while preserving exact matching rules for flow management (e.g., monitoring, rate limitation). However, the first hop switch is required to have high capacity and to perform intensive computations (e.g., packets header changes), which incurs performance penalty.

Compression also incurs computational overhead and slows down the network configuration update. Moreover, during the updating time, forwarding errors such as reachability failures and loops are susceptible to occur [55]. Therefore, to be efficient, compression algorithms must achieve a trade-off between compression ratio and update time. In general, most of compressed rules do not change during the update, so, the designed algorithm only needs to identify and re-compress the affected rules. An example of such an algorithm is iFFTA [55].

C. Split and Distribution

In general, a single switch does not have sufficient TCAM to store all rules. Therefore, the set of rules is usually split and distributed over the network in a way that satisfies policies. As shown in Fig. 6, the list of access control rules $R1, R2, R3, R4$ is split and distributed on the switches, according to the device

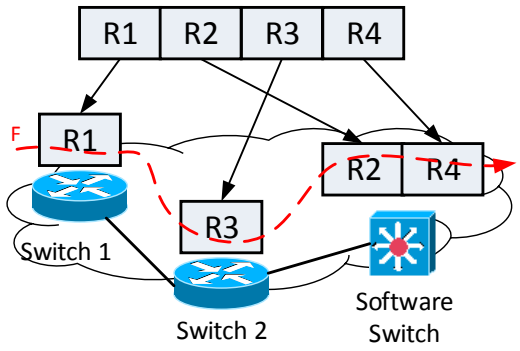

Fig. 6. An example of distribution approach. The list of access control rules $R1, R2, R3, R4$ is split and distributed, according to the device capacity and such that every flow in F passes through all the rules in the list.

capacity and such that every flow F passes through all the rules in the list.

The common approach to distribute rules is to formalize an optimization model that decides which rules are placed on which node, such that policy constraints, memory constraints, and rule dependency constraints are satisfied. The objective functions are flexible and depend on applications, such as to minimize the total number of rules needed [19], [46], [45], to minimize energy consumption [21], or to maximize traffic satisfaction [20]. Since the rules placement problem is NP-hard in most of the cases, these studies also propose heuristics to obtain near optimal rules placement solutions.

We first show in Sec. IV-C1 the different options to distribute rules over a network composed of multiple commodity OpenFlow switches built around TCAM-based flow tables. Finally, in Sec. IV-C2 we present how elementary network functions can be performed by software switches or additional network devices to reduce the controller overhead without impairing the management flexibility offered by OpenFlow.

1) Rules distribution among commodity switches:

a) Access control rules distribution: There are different solutions to split and distribute access control rules in OpenFlow networks [46], [19], [45], [28], [57].

The first challenge is how to split original access control rules into small, semantic equivalent subsets to fit in flow tables. The common approach is to represent access control rules as a directed dependency graph [46], [28], [45], which can be decomposed into subgraphs (e.g., using Cut Based Decomposition algorithm [46]), corresponding to subsets of rules that maintain original semantics. Other approaches propose splitting rules based on range [26] or using the Pivot Bit Decomposition algorithm [46].

The second challenge is how to distribute and assign these subsets of rules to switches. To that aim, linear programming models are formalized to assign subsets of rules to switches. Kanizo's model [46] distributes rules over all shortest paths from ingress to the egress node, such that each flow passes through all access control rules. However, as shown in [19], this approach is suboptimal for two reasons. First, only some paths require enforcing all access control rules. Second, their algorithm cannot use all available switches when the shortest path's length is small. In Kang's model [19], paths are derived from the routing policy and only the rules that affect packets traversing that path is installed. Zhang's model [45] captures the rules dependency and accounts for the compression across rules from different ingress points to further reduce the number of rules required.

b) Forwarding rules distribution: Different forwarding rules distribution algorithms have been proposed to implement forwarding plane for different objectives ([34], [33], [56], [22], [20], [21], [23]). The key challenge in forwarding rules distribution is how to select paths to install the forwarding rules that satisfy the policies and network constraints.

Path choice plays an important role in forwarding rules placement. Flows use rules on the paths to reach their endpoints and each path requires different rules. Some paths are more efficient than others; for example, the shortest hop paths are preferred because the minimum number of forwarding rules is needed [33], [56]. The path choice also depends on the traffic engineering goals (e.g., energy efficiency [21]). As shown in Fig. 7, to satisfy the endpoint policy (i.e. flow F exits at $E1$), rules can be placed on two different paths, one path needs two rules $R1, R2$ and the other needs three rules $R3, R1, R2$. In this case, the former path is preferred since less memory is consumed.

Proposed studies can be classified into two groups: one group enforcing routing policy (e.g., using shortest paths) [46], [19], [45], [56], [33], [23], [43] and another group that does not [22], [20], [9], [26], [28], [21]. In the first group, the path is an input to the problem, while it is an output in the second group. Strictly following the routing policy (e.g., using shortest paths) sometimes is necessary to obtain the required performance (e.g., throughput, latency). However, the paths specified by the routing policy may not always have enough capacity to place all the necessary rules [25]. Relaxing routing policy is suggested [25] to improve resource utilization while respecting the endpoint policy.

But relaxing routing policy may lead to numerous possible paths. Therefore, path heuristics are used to select the path the flow will use. For example, Nguyen et al. [20] choose the flow path such that it shares some segments with the default path formed by sequence of default rules on each node. In this manner, the flow can leverage default rules to reach the destination, thus reducing memory requirements.

However, a flow may not always be carried on a single path (e.g., because of bandwidth constraints). Paths can be chosen such that they satisfy the requirements while maximizing the number of nodes between them, so that all the forwarding rules required can be reduced thanks to compression [22]. In context of user mobility, paths can be predicted based on velocity and direction, and then forwarding rules can be installed on potential paths to avoid transmission interruption [23], [43], [58].

2) Rules distribution among commodity switches and additional resources: Studies presented in Sec. IV-C1 aim to distribute rules on commodity switches and cannot directly be applied to under-provisioned networks where memory budget, in particular with TCAMs, is limited [25], [28].

Rule	Match	Action
R1	F	To Switch 2
R2	F	To E1
R3	F	To Switch 3

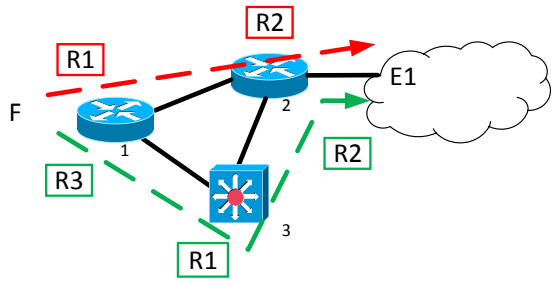

Fig. 7. An example of path choice. To satisfy the endpoint policy (e.g., flow F exits at $E1$), rules can be placed on two different paths, one path needs two rules $R1, R2$ and the other needs three rules $R3, R1, R2$. In this case, the former path is preferred since less memory is consumed.

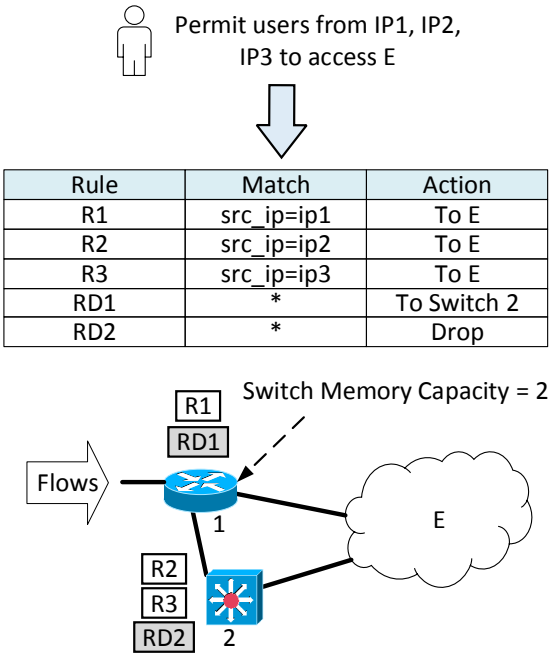

Fig. 8. An example of using additional devices to offload flow processing from commodity switches. Since Switch 1 does not have enough capacity to place all necessary rules $(R1, R2, R3)$, it uses the default rule $RD1$ to redirect flows to software switch 2 for further processing. Flow from $IP1$ passes through switch 1 to reach E, while flow from $IP2, IP3$ passes through switches 1,2 to reach E.

In practice, some flows are more sensitive to network conditions than others. For example, flows from delay-sensitive applications (e.g., VoIP) require lower latency than best effort traffic (e.g., Web browser). As a consequence, one can allow some flows to be processed on low performance paths and let room for critical flows on high-performance paths.

Recent studies suggest placing rules on additional, inexpensive devices without TCAM (e.g., software switches) to offload the memory burden for commodity switches [9], [20], [28]. The default rules on the commodity switches can be used to redirect flows that do not match any rule to these devices (e.g., the controller). These devices usually have large capacity (e.g., large flow tables implemented in RAMs), they are cheap to build (e.g., using Open vSwitch on a general purpose CPU [29]) but have limitations in forwarding and lookup performance, compared to commodity switches. An example is shown in Fig. 8; since Switch 1 does not have enough capacity to place all necessary rules $(R1, R2, R3)$, the default rule $RD1$ is used to redirect flows to software Switch 2 for further processing. Flows from $IP1$ pass through Switch 1 to reach E, while flows from $IP2, IP3$ pass through Switches $1, 2$ to reach E.

With the support from additional devices, resources are split into two kinds: *fast* (e.g., TCAM matching) and *default* (e.g., software switch matching). Studies [9], [20], [28] propose rules placement solutions that achieve the best trade-off between performance and cost. Basically, each rule is assigned an importance value, based on its priority and its dependency to other rules. Afterwards, linear programming models and heuristics are used to decide the most profitable rules to keep on commodity switches and the remaining rules to be installed on software switches. The aim of objective functions can be to minimize the redirection cost [9], or to maximize the whole values of rules installed on TCAM [28], [20].

The *split and distribution* approach combines different types of resources to perform network-wide optimization and to reduce CAPEX. For example, a switch with a large flow table capacity can be more expensive than several switches with smaller flow tables. Most of the studies formalize an optimization model for rules placement to maximize or minimize an objective function while satisfying different constraints. The main advantage of this approach is the flexibility in objective functions it allows, and its capacity to handle many constraints in a single framework.

However, this approach usually induces a redirection overhead (e.g., redirecting packets causing a flow table miss to other nodes), computation overhead (e.g., solving the optimization model), or rules duplication. Some studies require prediction of the traffic matrix, or future location of users, to be able to solve some optimization models. Such an accurate prediction is costly, because it requires a large signaling overhead to collect network statistics and continuous calibration of the prediction model.

V. Reducing Signaling Overhead

As explained in Sec. III-C, the signaling overhead should not be neglected while solving the OpenFlow rules placement problem. Reducing the signaling overhead is a key factor to increase the scalability of any rules placement solution. In this section, we summarize the ideas that have been proposed to reduce the signaling overhead.

A. Reactive and Proactive rules placement

There are two approaches for rules placement in OpenFlow: *reactive* and *proactive*.

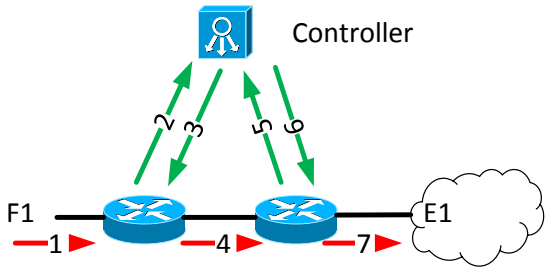

Fig. 9. An example of reactive rules placement. Rules are placed on demand, after flows arrive.

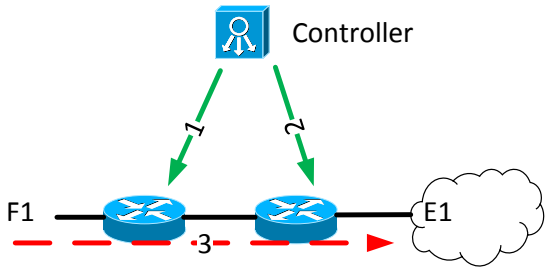

Fig. 10. An example of proactive rules placement. Rules are placed in advance, before a flow arrives.

1) Reactive: With the *Reactive* approach, rules are populated on demand to react upon flow events. As stated in the OpenFlow specification [13], for each new flow, the switch enqueues the packet and informs the controller about the arrival of a new flow. Afterwards, the controller computes the rules to be associated with the new flow and installs them in the network. So normally, a new flow requires $2n$ messages, where n is the number of path hops. Once the rules are installed on the switches, packets are dequeued and forwarded in the network. Any subsequent packet of the flow will then be processed by the freshly installed rules without further intervention of the controller. An example of reactive rules placement is shown in Fig. 9, in which a flow is queued at two switches (Arrow 1, 4). Four OpenFlow messages are required, including two new flow messages (Arrow 2, 5) and two rule installation messages (Arrow 3, 6), to forward the packets of flows towards the endpoint $E1$ (Arrow 7).

Reactive rules placement is required to adjust network configuration continuously with the current network state. For example, a new coming flow requires the controller to setup the path, whereas a down link event requires the controller to reroute all the affected flows.

However, using a reactive approach for all the flows is not the right solution because the controller and the switch buffer may be overloaded, e.g., in presence of large number of new flows (e.g., 100k new flows per second in a cluster [59]). Another drawback is the additional latency (e.g., 10ms to 20ms in data centers [27]). Therefore, the reactive approach should not be used for all flows.

2) Proactive: In this approach, rules are populated in advance, i.e., before the first packet of a new flow arrives. The proactive approach nullifies the setup delay of rules and reduces the overall number of signaling messages. An example of reactive rules placement is shown in Fig. 10. The controller installs rules for flow $F1$ in advance (Arrow 1, 2), before the flow $F1$ arrives (Arrow 3). So, two OpenFlow messages are required and there is no setup delay.

This *proactive* approach is common in studies focusing on access control [46], [19], [45], [28], as access control rules are predefined by operators independently of the traffic. The same approach can be used to decide forwarding rules in the network but it requires predicting or estimating in advance, the traffic demand or the user location [21], [20], [23], [22], [43]. In some practical situations, achieving accurate prediction is difficult as it incurs the collection of data and induces signaling messages [60]. Therefore, the proactive approach is suitable only for the flows that can be predicted with high accuracy.

We classify the related work that uses proactive and reactive approach in Table III.

B. Delegating functions to OpenFlow switches

Rules placement solutions need to be frequently updated often to adapt with current network state. But updating network and collecting statistics incurs load on the controller (e.g., CPU, bandwidth, memory) when done frequently. In this section, we discuss several solutions that can be used to reduce the signaling overhead,

Elementary network functions such as MAC learning and ICMP processing can be delegated to the switches, not only to reduce the signaling overhead, but also to keep basic network functions when controllers are not reachable [7].

To reduce both signaling overhead and delay caused by new flow setup, several studies [27], [7], [56] suggest delegating some functions to OpenFlow switches. Instead of querying the controller for each flow, switches can identify and process some flows (e.g., mice flows) and interrogate the controller when decisions are necessary.

Other mechanisms such as *rule cloning* and *local actions* also contribute to reduce the signaling overhead [27]. More precisely, *rule cloning* allows the switch to clone a rule from a pre-installed wildcard rule to handle a flow; *local actions* allows the switch to change the action field in rules, for example, fast re-routing to another path in case of link failures, without invoking the controller. Another approach is to use *authority* switches [26], which are built on top of OpenFlow switches. Authority switches can be used to handle flow table misses from edge switches, thus keeping the packets causing misses in the data-plane.

On rule removal (e.g., because of a timeout), signaling messages are required to inform the controller. To reduce the removal and re-installation overhead, eviction mechanisms like LRU or timeouts (mentioned in Sec. IV-A) can be directly implemented on the switches to keep rules with high matching probability in the flow table while automatically freeing space for new flows, everytime without invoking the controller.

Rules placement is computed using statistics queried from the network. For example, by collecting the number of bytes sending so far, the controller can detect that some flows are elephant and then install forwarding rules using the shortest paths. In general, high accuracy inputs require intensive collection of traffic statistics.

To reduce the overhead due to the collection of statistics, the default pull-based mechanism (i.e., the controller requests and receives statistics) can be replaced by a push-based mechanism [27] (i.e., the switch pushes the statistics to the controller when defined conditions are satisfied, for example, when the number of packets exceeds a threshold). Another complementary solution is to replace current OpenFlow counters by software defined counters [61], which support additional features such as data compression and elephant flows detection. In this manner, the statistics collection overhead can be further reduced.

Delegating elementary functions to switches is a way to reduce the signaling overhead between controllers and switches and to increase the overall scalability (e.g., the controller is less loaded) and the availability (e.g., basic network functionalities remain available upon controller failure). However, this approach requires more complex software and hardware than vanilla OpenFlow switches, which increases the cost of the device and may cause inconsistencies as each device makes its own decision [8].

VI. Future research directions

Existing studies have proposed different solutions for various use cases, however, the OpenFlow rules placement problem is still a challenging research area with many remaining questions. In this section, we discuss some open questions that may be worth future research attention.

A. Hybrid rules placement

Rules can be placed in advance (proactive) or on demand (reactive). Each approach has pros and cons as discussed in Sec. V-A. We believe that an efficient rules placement solution should combine both proactive and reactive, to benefit from their respective advantages without having to pay the cost of their drawbacks. For example, using proactive mode for access control rules, predictable flows, and using reactive mode for non-predictable flows.

How to combine these two approaches, which flows, which switches use reactive or proactive mode, how to obtain the best trade-offs between them remain interesting questions.

B. Robustness and fault tolerance rules placement

Upon changes (e.g., policies, network topology, user mobility), the rules placement needs to be updated to adapt with varying network conditions. However, updating rules placement comes with the cost of computation overhead, signaling overhead, and setup delay. Most of proposed solutions recompute the rules placement to maximize or to minimize some performance metrics (e.g., total number of rules, energy consumption).

Robustness and fault tolerance should also be taken into account when designing rules placement solutions. For this purpose, robust optimization techniques [62] might be a promising approach. Such techniques account for uncertainty in inputs and produce an output that is the best among all the possible inputs.

C. Impact of additional devices

Recently, additional devices such as controllers and software switches have been used to offload the memory burden of OpenFlow switches (see Sec.IV-C2). In such a case, default rules are often configured to redirect flow table miss from OpenFlow switches toward these devices [9], [28], [20].

The impact of the additional devices location, the default rules on the efficiency of rules placement are not well understood. It would be interesting to combine the controller placement [63] and the OpenFlow rules placement problem to overall optimize the network.

D. Multilevel table rules placement

OpenFlow 1.0 uses a flat table model that cannot handle possible state explosion. To address this issue, OpenFlow 1.1+ [13] supports multi-level flow tables and pipeline processing. Consequently, a large flow table on a switch can be split into smaller flow tables with less entries. Furthermore, some rules require to be placed in TCAM tables while the remaining rules are placed in non-TCAM tables. The multi-tables feature has been implemented in commercial devices, such as NoviSwitch 1248 [64].

How to benefit from multi-level architecture, how to leverage pipeline processing ability, which rules to place in which sub-tables should be taken into account for future research.

E. Network Function Virtualization

Networks today consist of a large number of various middleboxes and hardware appliances (e.g., Firewall, Deep Packet Inspection). Usually, launching a new service requires other hardware-based appliances, which increases the cost of investments, energy, integration, operation, and maintenance.

Network Function Virtualization (NFV) [65] aims to replace network equipments by software-based functions on high volume servers, switches, and storage devices. NFV requires flexible routing, dynamic instantiations and placement of network functions, which can be provided by OpenFlow. More precisely, some network functions are possible to implement by placing appropriate rules in OpenFlow switches (e.g., firewall functions). Furthermore, forwarding rules can be placed on switches to redirect flows through different network functions. How and where to place network functions, how traffic is routed through them are key challenges towards the deployment of NFV. Indeed, NFV is a fascinating use case and new rules placement solutions must be designed to implement NFV.

VII. Conclusion

Software Defined Networking and OpenFlow offer the ability to simplify network management and reduce costs by raising the level of network abstraction.

An abstraction layer between operators and the network is desired to transform the high-level policies from operators into low level OpenFlow rules. To that aim, it is important to solve the *OpenFlow rules placement problem*, that decide the OpenFlow rules that must be deployed and where to install them in order to efficiently use network resources, such as bandwidth and memory, while respecting operational constraints and policies.

In this survey, we present the body of the literature that investigates the OpenFlow rules placement problem. We first formalize the problem and identify two main challenges: resource limitations and signaling overhead. We then discuss existing ideas and solutions to solve these two challenges. Finally, we point out several research directions for further exploration.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[2] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[3] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[4] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM.

[5] B.A.A. Nunes, M. Mendonca, Xuan-Nam Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *Communications Surveys Tutorials, IEEE*, 16(3):1617–1634, Third 2014.

[6] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, D. Niyato, and Haiyong Xie. A survey on software-defined networking. *Communications Surveys Tutorials, IEEE*, 17(1):27–51, Firstquarter 2015.

[7] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 103(1):14–76, Jan 2015.

[8] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Xie Geoffrey, and Laurent Vanbever. On the Co-Existence of Distributed and Centralized Routing Control-Planes. In *IEEE INFOCOM*, April 2015.

[9] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *USENIX HotCloud*, pages 23–23, Berkeley, CA, USA, 2012.

[10] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. PAST: Scalable Ethernet for Data Centers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[11] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013.

[12] Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan van Reijendam, Paul Weissmann, and Nick McKeown. Maturing of OpenFlow and Software-defined Networking through deployments. *Computer Networks*, 61:151–175, March 2014.

[13] OpenFlow. OpenFlow Switch Specification. http://www.opennetworking.org/, 2015. accessed 19-Aug-2015.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM CCR*, 38(3):105–110, 2008.

[15] Floodlight. Floodlight. http://floodlight.openflowhub.org/, 2015.

[16] David Erickson. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[17] Ryu. Ryu controller. http://osrg.github.com/ryu/, 2015. accessed 19-Aug-2015.

[18] OpenDaylight. OpenDaylight Controller. http://www.opendaylight.org/, 2015. accessed 19-Aug-2015.

[19] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[20] Xuan Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. OFFICER: A general Optimization Framework for OpenFlow Rule Allocation and Endpoint Policy Enforcement. In *IEEE INFOCOM*, April 2015.

[21] F. Giroire, J. Moulierac, and T.K. Phan. Optimizing rule placement in software-defined networks for energy-aware routing. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 2523–2529, Dec 2014.

[22] Huawei Huang, Peng Li, Song Guo, and Baoliu Ye. The Joint Optimization of Rules Allocation and Traffic Engineering in Software Defined Network. pages 141–146, May 2014.

[23] He Li, Peng Li, and Song Guo. Morule: Optimized rule placement for mobile users in sdn-enabled access networks. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 4953–4958, Dec 2014.

[24] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[25] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Optimizing Rules Placement in OpenFlow Networks: Trading Routing for Better Efficiency. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 127–132, New York, NY, USA, 2014. ACM.

[26] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *SIGCOMM CCR*, 41(4), August 2010.

[27] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, August 2011.

[28] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite CacheFlow in Software-defined Networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, New York, NY, USA, 2014. ACM.

[29] OpenvSwitch. OpenvSwitch. http://openvswitch.org/, 2015. accessed 19-Aug-2015.

[30] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable Rule Management for Data Centers. *NSDI'13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 157–170, 2013.

[31] Daekyeong Moon, Jad Naous, Junda Liu, Kyriakos Zarifis, Martin Casado, Teemu Koponen, Scott Shenker, and Lee Breslau. Bridging the Software/Hardware Forwarding Divide, 2010. UC Berkeley.

[32] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[33] A.S. Iyer, V. Mann, and N.R. Samineni. SwitchReduce: Reducing switch state and controller involvement in OpenFlow networks. In *IFIP Networking Conference*, pages 1–9, May 2013.

[34] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow macs: Scalable label-switching for commodity ethernet. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 157–162, New York, NY, USA, 2014. ACM.

[35] Eun-Do Kim, Seung-Ik Lee, Yunchul Choi, Myung-Ki Shin, and Hyoung-Jun Kim. A flow entry management scheme for reducing

controller overhead. In *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, pages 754–757, Feb 2014.

[36] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. Effective switch memory management in OpenFlow networks. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*, pages 177–188, 2014.

[37] Adam Zarek, Yashar Ganjali, and David Lie. OpenFlow Timeouts Demystified, 2014. Master Thesis, Department of Computer Science, University of Toronto, Canada.

[38] Bu Sung Lee, Renuga Kanagavelu, and Khin Mi Mi Aung. An efficient flow cache algorithm with improved fairness in Software-Defined Data Center Networks. *Proceedings of the 2013 IEEE 2nd International Conference on Cloud Networking, CloudNet 2013*, pages 18–24, 2013.

[39] Miguel Cardoso Neves. On Time-based Strategies for Optimizing Flow Tables in SDN, Dec 2014. Master Thesis, Department of Computer Science, The Federal University of Rio Grande do Sul, Brazil.

[40] Kalapriya Kannan and Subhasis Banerjee. FlowMaster: Early Eviction of Dead Flow on SDN Switches. In *Distributed Computing and Networking*, pages 484–498. Springer Berlin Heidelberg, 2014.

[41] Liang Xie, Zhifeng Zhao, Yifan Zhou, Gang Wang, Qianlan Ying, and Honggang Zhang. An adaptive scheme for data forwarding in software Defined Network. In *Wireless Communications and Signal Processing (WCSP), 2014 Sixth International Conference on*, pages 1–5, Oct 2014.

[42] Huikang Zhu, Hongbo Fan, Xuan Luo, and Yaohui Jin. Intelligent timeout master: Dynamic timeout for SDN-based data centers. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium*, pages 734–737, May 2015.

[43] Xin Wang, Cheng Wang, Changjun Jiang, Lei Yang, Zhong Li, and Xiaobo Zhou. Rule Optimization for Real-Time Query Service in Software-Defined Internet of Vehicles. *CoRR*, abs/1503.05646, 2015.

[44] R.P. Draves, C. King, S. Venkatachary, and B.D. Zill. Constructing optimal IP routing tables. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 88–97 vol.1, Mar 1999.

[45] Shuyuan Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik. An Adaptable Rule Placement for Software-Defined Networks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 88–99, June 2014.

[46] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM*, pages 545–549, Apr. 2013.

[47] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing Rectilinear Pictures and Minimizing Access Control Lists. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1066–1075, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[48] A.X. Liu, C.R. Meiners, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. *Networking, IEEE/ACM Transactions on*, 18(2):490–500, April 2010.

[49] C.R. Meiners, A.X. Liu, and E. Torng. Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs. *Networking, IEEE/ACM Transactions on*, 20(2):488–500, April 2012.

[50] W. Braun and M. Menth. Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 25–30, Sept 2014.

[51] M Rifai, N Huin, C Caillouet, F Giroire, D Lopez-Pacheco, J Moulierac, and G Urvoy-Keller. Too many SDN rules? Compress them with MINNIE. In *Global Communications Conference (GLOBECOM), IEEE*, pages 1–6, Jul 2015.

[52] Yasunobu Chiba, Yusuke Shinohara, and Hideyuki Shimonishi. Source Flow: Handling Millions of Flows on Flow-based Nodes. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 465–466, New York, NY, USA, 2010. ACM.

[53] Jon Postel. Internet protocol. STD 5, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt.

[54] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. *SIGCOMM Comput. Commun. Rev.*, 44(4):539–550, August 2014.

[55] Shouxi Luo, Hongfang Yu, and Le Min Li. Fast incremental flow table aggregation in SDN. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–8, Aug 2014.

[56] Yukihiro Nakagawa, Kazuki Hyoudou, Chunghan Lee, Shinji Kobayashi, Osamu Shiraki, and Takeshi Shimizu. DomainFlow: Practical Flow Management Method Using Multiple Flow Tables in Commodity Switches. In *ACM CoNEXT*, pages 399–404. ACM, 2013.

[57] J. Huang, G. Chang, C. Wang, and C. Lin. Heterogeneous Flow Table Distribution in Software-defined Networks. *Emerging Topics in Computing, IEEE Transactions on*, PP(99):1–6, Jul 2015.

[58] Mianxiong Dong, He Li, Kaoru Ota, and Jiang Xiao. Rule caching in SDN-enabled mobile access networks. *Network, IEEE*, 29(4):40–45, July 2015.

[59] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM.

[60] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[61] Jeffrey C. Mogul and Paul Congdon. Hey, You Darned Counters!: Get off My ASIC! In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 25–30, New York, NY, USA, 2012. ACM.

[62] Dimitris Bertsimas, David B. Brown, and Constantine Caramanis. Theory and Applications of Robust Optimization. *SIAM Review*, 53(3):464–501, 2011.

[63] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[64] NoviFlow. NoviSwitch 1248 High Performance Switch, Jan 2013. http://www.nvc.co.jp/pdf/product/noviflow/NoviSwitch1248Datasheet.pdf.

[65] Margaret Chiosi. Network Function Virtualisation White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012. accessed 19-Aug-2015.

TABLE I
TERMINOLOGY USING IN THIS PAPER

Term	Definition
Access control rule	Rule having actions field *drop/permit packets*
Additional devices	Devices (e.g., software switches, controllers) that store rules in non-TCAM (e.g., RAM)
Commodity switch	OpenFlow switch that stores rules in TCAM
Default path	Sequence of nodes from an ingress switch to the additional devices, formed by default rules
Default rule	Rule that have the lowest priority and matches all the packets
Elephant flow	Flow that sends a large number of bytes
Endpoint policy	Policy that defines the endpoints for flows
Exact-matching rule	Rule that does not contain ternary elements ($*$) in its matching pattern.
Flow	A sequence of packets that have common header fields (e.g., destination IP address)
Flow table	Set of prioritized rules on the switch
Flow table hit	An incoming flow is processed by a non-default rule
Flow table miss	An incoming flow is processed by the default rule
Forwarding rule	Rule having the actions field *forward packets to an interface*
Mouse flow	Flow that sends a few bytes
Routing policy	Policy that specifies the path the flow must follow
Rule	An instruction for the OpenFlow switch specifying how to process the packets.
Rule space	Set of all possible rules
Rules placement	A configuration that indicates which rules are placed on each switch
Wildcard rule	Rule that contains ternary elements ($*$) in its matching pattern

TABLE III
COMPARISON OF RELATED WORK BY RULE PLACEMENT **MODE** (R: REACTIVE, P: PROACTIVE), MEMORY MANAGEMENT TECHNIQUES (**EVICTION, COMPRESSION, DISTRIBUTION**), **USE CASES** AND **VALIDATION** METHODOLOGY

Related work	Mode	Eviction	Compression	Distribution	Use Cases	Validation
Zarek et al. [37]	R	v			-	Simulation
Kim et al. [35]	R	v			-	Emulation
Xie et al. [41]	R	v			Traffic Engineering	Simulation
Zhu et al. [42]	R	v			Traffic Engineering	Simulation
Vishnoi et al. [36]	R	v			Traffic Engineering	Prototype
Curtis et al. [27]	P		v		Flow Management in Data Centers	Simulation
Chiba et al. [52]	R		v		-	Prototype
Luo et al. [55]	-		v		-	Simulation
Braun et al. [50]	P		v		BGP Flow Table Management	Simulation
Yu et al. [26]	R		v	v	Flow Management	Prototype
Agarwal et al. [34]	R		v	v	Data Forwarding in Data Centers	Prototype
Moshref et al. [9][30]	P		v	v	Rule Management in Cloud, Data Centers	Prototype
Nakagawa et al. [56]	P		v	v	Traffic Engineering	Prototype
Iyer et al. [33]	P		v	v	Traffic Engineering	Emulation
Kanizo et al. [46]	P		v	v	Distributed ACLs	Simulation
Kang et al. [19]	P		v	v	Distributed ACLs, Load Balancer	Simulation
Katta et al. [28]	P		v	v	Distributed ACLs	Prototype
Huang et al. [22]	P		v	v	Traffic Engineering	Simulation
Zhang et al. [45]	P		v	v	Distributed ACLs	Simulation
Huang et al. [57]	P		v	v	Distributed ACLs	Simulation
Giroire et al. [21]	P		v	v	Energy efficiency routing	Simulation
Li et al. [23]	P			v	Data forwarding in Mobile networks	Simulation
Nguyen et al. [20]	P		v	v	Traffic engineering	Simulation
Wang et al. [43]	P	v	v	v	Data forwarding in Vehicle networks	Simulation
Rifai et al. [51]	R		v	v	Traffic Engineering	Prototype