OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch

Giuseppe Bianchi, Marco Bonola
CNIT / Univ. Roma Tor Vergata
{name.surname}@uniroma2.it

Antonio Capone, Carmelo Cascone
Politecnico di Milano
{name.surname}@polimi.it

ABSTRACT

Software Defined Networking envisions smart centralized controllers governing the forwarding behavior of dumb low-cost switches. But are "dumb" switches an actual strategic choice, or (at least to some extent) are they a consequence of the lack of viable alternatives to OpenFlow as programmatic data plane forwarding interface? Indeed, some level of (programmable) control logic in the switches might be beneficial to offload logically centralized controllers (de facto complex distributed systems) from decisions *just* based on local states (versus network-wide knowledge), which could be handled at wire speed *inside* the device itself. Also, it would reduce the amount of flow processing tasks currently delegated to specialized middleboxes. The underlying challenge is: can we devise a *stateful* data plane programming abstraction (versus the stateless OpenFlow match/action table) which still entails high performance and remains consistent with the vendors' preference for closed platforms? We posit that a promising answer revolves around the usage of extended finite state machines, as an extension (super-set) of the OpenFlow match/action abstraction. We concretely turn our proposed abstraction into an actual table-based API, and, perhaps surprisingly, we show how it can be supported by (mostly) reusing core primitives already implemented in OpenFlow devices.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Network Architecture and Design

Keywords

OpenFlow, programming interfaces, SDN, state machines

1. INTRODUCTION

Just a few years ago it was normal to configure network devices using proprietary interfaces, differing across vendors, device types (switches, routers, firewalls, load balancers, etc.), and even different firmware releases for a same equipment. Managing heterogeneous multi-vendor networks of non marginal scale could be extremely difficult, and could require a huge expertise.

OpenFlow [1] emerged in 2008 as an attempt to change this situation. OpenFlow's approach was the identification of a *vendor-agnostic* programming abstraction for configuring the forwarding behavior of switching fabrics. Via the OpenFlow Application Programming Interface (API), network administrators could remotely reconfigure at runtime forwarding tables, probe for flow statistics, and redirect packets not matching any local flow entry towards a central controller for further processing and for taking relevant decisions; in essence "program" the network from a central control point, clearly separated from the forwarding plane.

Although this vision, which we today call *Software Defined Networking* (SDN), finds its roots in earlier works [2], does not nearly restrict to OpenFlow as device-level API, and goes well beyond data plane programmatic interfaces, OpenFlow is recognized as the technology which brought SDN in the real world [3]. OpenFlow was immediately deployable, thanks to its pragmatic balance between open network programmability and real world vendors' and deployers' needs. Starting from the recognition that several different network devices implement somewhat similar flow tables for a broad range of networking functionalities (L2/L3 forwarding, firewall, NAT, etc), the authors of OpenFlow proposed an *abstract model* of a *programmable* flow table which was "*amenable to high-performance and low-cost implementations; capable of supporting a broad range of research; and consistent with vendors' need for closed platforms*" (quote from [1]). Via the OpenFlow "match/action" abstraction, the device programmer could broadly specify a flow via an header matching rule, associate forwarding/processing *actions* (natively implemented in the device) to the matching packets, and access bytes/packet statistics associated to the specified flow.

"dumb" switches: choice or compromise?

Almost six years have now passed since the OpenFlow inception, and the latest OpenFlow standard, now at version 1.4 [4], appears way more complex than the initial elegant and simple concept. To fit the real world needs, a huge number of extension (not only the initially foreseen functional ones, such as supplementary actions or more flexible header matching, but also structural ones such as action bundles, multiple pipelined tables, synchronized tables, and many more [5]) were promoted in the course of the standardization process. And new extensions are currently under discussion for the next OpenFlow version, among which flow states which we will further discuss later on.

All this hectic work was not accompanied by any substantial rethinking in the original programmatic abstraction (besides the abandoned Google OpenFlow 2.0 proposal, considered too ambitious and futuristic [6]), so as to properly capture the emerging extensions, simplify their handling [7], and prevent the emergence of brittle, platform-specific, implementations [6] which may ultimately threaten the original vendor-independency goal of the OpenFlow inventors.

As a result, even if an OpenFlow device is now rich of functionalities and primitives, it remains completely "dumb", with all the "smartness" placed at the controller side.

Someone could argue that this is completely in line with the spirit of SDN's control and data plane separation. And, to avoid misunderstandings, we fully agree that network management and control can be (logically) centralized. However we posit that several stateful tasks, just involving *local states* inside *single links/switches* are *unnecessarily* centralized for easy management and programmability, for the only reason that they cannot be deployed on the local OpenFlow devices without retaining the explicit involvement of the controller for any state update (for a notable example, think to the off-the-shelf Layer 2 MAC learning operation[1]). As a result, the explicit involvement of the controller for *any* stateful processing and for *any* update of the match/action rules, is problematic. In the best case, this leads to extra signaling load and processing delay, and calls for a capillary distributed implementation of the "logically" centralized controller. In the worst case, the very slow control plane operation a priori prevents the support of network control algorithms which require prompt, real time reconfiguration in the data plane forwarding behavior.

In essence, dumbness in the data forwarding plane appears to be a *by-product* of the limited capability of the OpenFlow data plane API compromise, rather than an actual design choice or an SDN postulate. Can we then emerge with better data plane APIs which permit to program some level of smartness directly *inside* the forwarding device?

Contribution

As argued above, our belief is that a major shortcoming of OpenFlow is its inability to permit the programmer to deploy states *inside* the device itself. Adding states to OpenFlow (as currently under discussion in ONF) is however not sufficient: the programmer should be entitled to formally specify how states should be handled, and this specification should be *executed inside the device* with no further interaction with the controller. Moreover, a viable solution must come along with two fundamental attributes. First, it must be amenable to high speed implementation. Second, it must *not* violate the vendor-agnostic principle which has driven the OpenFlow invention, and which has fostered SDN; in essence, it must emerge as a concrete and pragmatic "abstraction", rather than as a technical approach.

Our work mainly focuses on this second aspect: although some hints are provided in section 3 on how our proposed approach can be efficiently supported by existing OpenFlow hardware, we *do not claim* to have fully addressed this aspect (as a compelling answer would require to exhibit an actual high speed HW implementation, which we do not yet have at the time of writing). Rather, our main contribution consists in the proposal of a *viable abstraction* to formally describe a desired stateful processing of flows *inside* the device itself, without requiring the device to be open source or to expose its internal design. Our abstraction relies on *eXtended Finite State Machines* (XFSM), which have been recently shown to be effective in a very different networking field, platform-agnostic wireless medium access control programmability [8, 9]. More specifically, we first introduce in section 2 a simplified XFSM called *Mealy Machine*, and a relevant programmatic interface which can be interpreted as a somewhat natural generalization of the OpenFlow match/action abstraction. In section 3 we discuss viability and implementation issues, showing that XFSM support can largely reuse existing OpenFlow features. Finally, section 4 discusses extensions towards support of "full" XFSMs [10] and the possible benefits.

Related work

Despite OpenFlow's data plane programmability, the need to use advanced packet handling for key network services has led to the proliferation of many types of specialized middleboxes [11]. The extension of programmability and flexibility features to these advanced network functions is a crucial aspect [12, 13], and a recent trend is that of virtualizing them in data centers on general purpose hardware platforms and to make them programmable and configurable using SDN approaches [14].

It is quite evident that SDN for general purpose and specialized hardware has radically different constraints and objectives on the abstraction for configuring packet handling functionalities [13]. We argue that, extending the OpenFlow switch abstraction allows to offload on high performance switches a pretty large set of functions reducing the need to relay on controllers and middleboxes.

The need to extend the OpenFlow data plane abstraction has been recently recognized by the research community [15, 16, 6]. In [15], the authors point out that the rigid table structure of current hardware switches limits the flexibility of OpenFlow packet processing to matching on a fixed set of fields and to a small set of actions, and introduce a logical table structure RMT (Reconfigurable Match Table) on top of the existing fixed physical tables and new action primitives. Notably, the proposed scheme allows not only to consider arbitrary width and depth of the matching for the header vector but also to define actions that can take input arguments and rewrite header fields. In [16], the approach is more radical and, similarly to the early work on active networks, packets are allowed to carry a tiny code that define processing in the switch data plane. A very interesting aspect is the proposal of targeted ASIC implementations where an extremely small set of instructions and memory space can be used to define packet processing.

OpenFlow standardization has so far significantly extended the set of actions (including also action bundles) and functionalities [4], and there is some debate on the inclusion of flow states in a next version [5]. However, extensions appear to add more and more capabilities, but with limited attention to how to duly accommodate them in a clean API [7], or event rethink its foundational principles (e.g., the Google's OpenFlow 2.0 proposal [6], which was considered too disruptive).

[1] Ironically, MAC learning is frequently invoked to motivate OpenFlow extensions [5]. For instance, flow table synchronisation (different views of the same data at different points of the OpenFlow pipeline), to permit learning and forwarding functions to access the same data. Or flow monitors (tracking of flow table changes in a multi-controller deployment), to permit a device natively implement a legacy MAC learning function to inform the remote controller of any new MAC address learned; in essence to permit to break (!) the original OpenFlow vision of general purpose forwarding device configured only through the data plane programming interface...

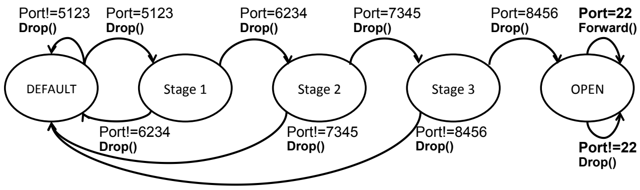

Figure 1: port knocking example: State Machine

Finally, the usage of XFSMs was initially inspired by [8] where (bytecoded) XFSMs were used to convey a desired medium access control operation into a specialized (but closed [9]) wireless interface card. While the abstraction (XFSM) is similar, the context (wireless protocols versus flow processing), technical choices (state machine execution engine versus table-based structures), and handled events (signals and timers versus header matching), are not nearly comparable.

2. BASIC ABSTRACTION

2.1 An illustrative example

The OpenFlow data plane abstraction is based on a single table of match/action rules for version 1.0, and multiple tables from version 1.1 on. Unless *explicitly* changed by the remote controller through flow-mod messages, rules are *static*, i.e., all packets in a flow experience the same forwarding behavior.

Many applications would however benefit from the ability to evolve the forwarding behavior on a packet-by-packet basis, i.e., depending on which sequence of packets we have received so far. A perhaps niche, but indeed very descriptive example is that of *port knocking*, a well known method for opening a port on a firewall. A host that wants to establish a connection (say an ssh session, i.e., port 22) delivers a sequence of packets addressed to an ordered list of prespecified closed ports, say ports 5123, 6234, 7345 and 8456. Once the exact sequence of packets is received, the firewall opens port 22 for the considered host. Before this event, all packets (including of course the knocking ones) are dropped.

As any other stateful application, such an operation cannot be configured *inside* an OpenFlow switch, but must be implemented in an external controller. The price to pay is that a potentially large amount of signalling information (in principle up to all packets addressed to all closed ports!) must be conveyed to the controller. Moreover, a timely flow-mod command from the controller is needed for opening port 22 after a correct knocking sequence, to avoid that the first "legitimate" ssh packet finds port 22 still closed. On the other side, implementing this application in the controller brings no gain: it does not benefit from network-wide knowledge or high level security policies [17], but uses just local states associated to specific flows on a single specific device.

Anyway, let us postpone the discussion on *where* this operation is implemented, and let us rather focus on *how* we can *model* such a desired behavior. Arguably, the most natural way is to associate, *to each host*, the finite state machine illustrated in Figure 1. Starting from a DEFAULT state, each correctly knocked port will cause a transition to a series of three intermediate states, until a final OPEN state is reached. Any knock on a port different from the expected one will bring back to the DEFAULT state. When in the OPEN state, packets addressed to port 22 (and only to this port) will be forwarded, whereas all remaining packets will be dropped, but without resetting the state to DEFAULT.

2.2 Extended Finite State Machines

A closer look at Figure 1 reveals that each state transition is caused by an *event*, which specifically consists in a packet *matching* a given port number. Moreover, each state transition caused by an event match, is associated to a forwarding *action* (in the example, drop or forward). A state transition thus reminds very closely a legacy OpenFlow match/action rule, but placed in a more general framework, characterized by the following two distinguishing aspects.

2.2.1 XFSM Abstraction

We remark that the match which specifies an event not only depends on packet header information, but also depends on the state; using the above port knocking example, a packet with port=22 is associated to a forward action when in the OPEN state, but to a drop action when in any other state. Moreover, the event not only causes an action, but also a transition to a next state (including self-transitions from a state to itself).

All this can be modeled, in an *abstract* form, by means of a simplified type[2] of *eXtended Finite State Machine* (XFSM), known as *Mealy Machine*. Formally, such a simplified XFSM is an *abstract* model comprising a 4-tuple (S, I, O, T), plus an initial starting (default) state S_0, where i) S is a finite set of states; ii) I is a finite set of input symbols (events); iii) O is a finite set of output symbols (actions); and iv) $T : S \times I \rightarrow S \times O$ is a transition function which maps <state,event> pairs into <state,action> pairs.

Similarly to the OpenFlow API, the abstraction is made concrete (while retaining platform independency) by restricting the set O of actions to those available in current OpenFlow devices, and by restricting the set I of events to OpenFlow matches on header fields and metadata easily implementable in hardware platforms. The finite set of states S (concretely, state labels, i.e., bit strings), and the relevant state transitions, in essence the "behavior" of a stateful application, are left to the programmer's freedom.

2.2.2 State Management

Matches in OpenFlow are generically collected in flow tables. The discussion carried out so far recommends to *clearly* separate the matches which define *events* (port matching in the port knocking example) from those which define *flows*, meant as entities which are attributed a state (host IP addresses). While event matches cause state transitions for a given flow, and are specified by an XFSM, flow matches are in charge to identify and manage the *state* associated to the flow the arriving packet belongs to. Two distinct tables (**State Table** and **XFSM table**), and three logical steps thus naturally emerge for handling a packet (Figure 2).

1. State lookup: It consists in querying a State Table using as key the packet header field(s) which identifies the flow, for instance the source IP address; if a state is not

[2]While in this section, for concreteness, we limit to Mealy Machines, in section 4 we discuss further possible extensions towards the most general XFSM abstraction as defined in [10]. Unless ambiguity emerges, we will loosely use the term XFSM also to refer to the special case of Mealy Machines.

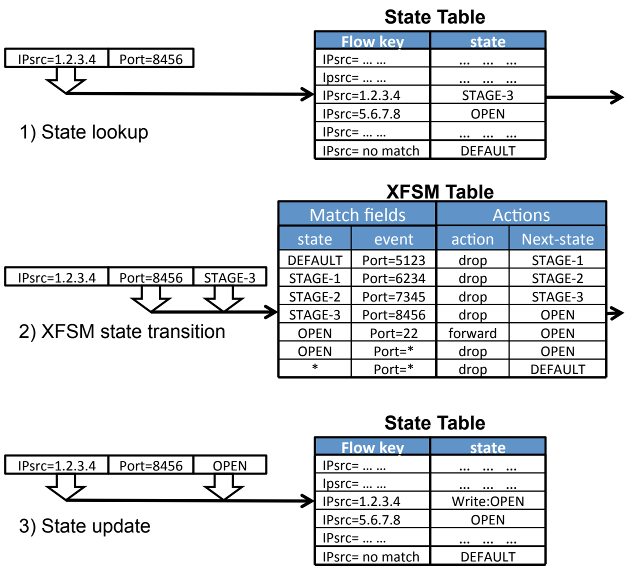

Figure 2: State Table, XFSM table, and packet handling for the port knocking example.

found for a queried flow, we assume that a default state is returned;

2. XFSM transition: The retrieved state label, added as metadata to the packet, is used along with the header fields involved in the event matching (e.g., port number), to perform a match on an XFSM table, which returns i) the associated action(s), and ii) the label of the next state;

3. State update: It consists in *rewriting* (or adding a new entry to) the state table using the provided next state label.

The example in Figure 2 shows how the port knocking example is supported in our proposed approach. The "program" contained in the XFSM table (7 entries) implements the port knocking state machine. Assume the arrival of a packet from host 1.2.3.4; the state lookup (top figure) permits to retrieve the current state, STAGE-3. Via the XFSM table (middle figure), we determine that this state, along with the knocked port 8456, triggers a drop action and a state transition to OPEN (middle figure). The new state is written (bottom figure) back in the state table for the host entry. In the XFSM table, we assume an ordered matching priority, with the last row having the lowest priority. As a result, all the four transitions to the default state for packets not matching the expected knocked port are coalesced in the last entry. A notable characteristic of the proposed solution is that the length of the tables is proportional to the number of flows (state table) and number of states (XFSM table), but *not* to their product.

2.3 Flow identification

Unfortunately, the above described abstraction still misses a fundamental further step which permits to model a subset of important stateful operations, in which states for a given flow are updated by events occurring on *different* flows. A prominent example is MAC learning: packets are forwarded using the *destination* MAC address, but the forwarding database is updated using the *source* MAC address. Similarly, the handling of bidirectional flows may encounter

MAC learning XFSM

lookup-scope $= \{macdst\}$
update-scope $= \{macsrc\}$

State	Event	Actions	Next-state
DEFAULT	in_port=1	flood	PORT-1
PORT-1	in_port=1	output 1	PORT-1
PORT-2	in_port=1	output 2	PORT-1
…			
PORT-N	in_port=1	output N	PORT-1
…			
…			
DEFAULT	in_port=N	flood	PORT-N
PORT-1	in_port=N	output 1	PORT-N
PORT-2	in_port=N	output 2	PORT-N
…			
PORT-N	in_port=N	output N	PORT-N

Table 1: XFSM table for a MAC learning Layer 2 switch with N ports;"in_port"= switch input port of the packet, action "flood"= replicate and forward packet to all switch ports except in_port; state labels = port number to which the flow shall be forwarded.

the same needs; for instance, the detection of a returning TCP SYNACK packet could trigger a state transition on the opposite direction. And in protocols such as FTP, a control exchange on port 21 could be used to set a state on the data transfer session on port 20.

The root cause of this issue is that, so far, we have not yet conceptually separated the *identity* of the flow to which a state is associated, from the actual *position* in the header field from which such an identity is retrieved. Since, in our proposed abstraction, flow identification is needed to *lookup* and to *update* the state table, we simply need to provide the programmer with the ability to use an eventually *different* header field in these two accesses to the State Table. We thus define as "lookup-scope" and "update-scope" the ordered sequence of header fields that shall be used to produce the key used to access the state table and perform, respectively, a lookup or an update operation.

With such feature, programming, say, a MAC learning operation, becomes trivial. We start by defining the state associated to a flow identity (namely, a MAC address) the current switch port to which packets should be forwarded (or DEFAULT if no port has been yet learned). During state lookup, the *lookup-scope* is set to be the MAC destination address. During state update, we define as *udpate-scope* the MAC source address. Finally, we fill the XFSM table with the transitions given in Table 1. Thanks to the *udpate-scope*, the <key,value> pair used in the State Table update is thus <macsrc,next-state>. In this example table, we on purpose assume compatibility with the current OpenFlow specification, and the $N^2 + N$ size of the table (being N the number of switch ports) thus depends on the OpenFlow limitations, and not on our XFSM abstraction. Indeed, we remark that the usage of parameters permitted by the Reconfigurable Match Tables recently introduced in [15] would yield an XFSM table comprising only two entries: the default one plus the entry $state : port(i) \times event : in\_port(j) \rightarrow action : output(i) \times next\_state : in\_port(j)$.

2.4 Application Programming Interface

As a summary, our basic data plane programming abstraction to formally specify a stateful operation comprises the specification of two tables in terms of:

1. an XFSM table comprising four columns: i) a **state** provided as a user-defined label, ii) an **event** expressed as an OpenFlow match, iii) a list of OpenFlow **actions**, and iv) a **next-state** label; each row is a designed state transition;

2. the **lookup-scope** and **update-scope** used to access and update the State Table, respectively.

It is not yet clear, at this stage, whether it could be practically convenient to further generalize such an API by permitting to bind a different *update-scope* to different entries in the XFSM table; in other words, associate each next-state entry with its *update-scope* which may then differ depending on the specific transition considered (a row in the XFSM table). Indeed, this extra flexibility, for which we have not yet identified a clear use case, would be paid in terms of additional internal hardware complexity.

3. IMPLEMENTATION ISSUES

3.1 Feasibility analysis

We first discuss how a switch architecture should be conceptually extended to support our proposed stateful operation. Our specific focus is to gain insights on which currently available OpenFlow primitives can be reused (and how), and which new primitives need to be added.

3.1.1 Architecture and primitives

A crucial feature needed by our scheme is the ability to perform matches using state labels and use more than one table. These features are indeed available: since Openflow 1.1, table pipeline processing and metadata support have been introduced. A packet entering an OpenFlow switch is processed through a set of linked flow tables that provide matching, forwarding, and packet modification. Metadata are used to extend packet header so as to carry arbitrary information from one table to the next. The controller can install/remove flow entries by sending *flow-mod* messages.

We indicate with the term *stateless stage* the processing operated by a single flow table. Conversely, we define as *stateful stage* (Figure 3) a logical block comprising a State Table and an XFSM table, and implementing our abstraction. A packet is first processed by a *key extractor* which produces a string of bits representing the key to be used to match a row in the state table. The key is derived by concatenating the header fields defined in the *lookup-scope*. The matched state label is appended to the packet headers as metadata. In case of table-miss (the key is not matched) then a *DEFAULT* state will be appended to the packet headers. If the header fields specified by the lookup-scope are not found (e.g. extracting the IP source address when the Ethernet type is not IP), a special state value *NULL* is returned.

The XFSM table can be implemented in OpenFlow v1.1+ as a standard flow table whose entries are matched using the relevant header fields representing the event and the (metadata) state label. We only need to specify, along with the action set to be executed, a supplementary command developed as an OpenFlow instruction, specifically a new `SET_STATE` instruction that will immediately trigger an update of the previous state table. The usage of an instruction guarantees that the state update is performed at the end of the stage, even when action bundles are configured, and permits to pipeline our stateful stage with supplementary stages, including other stateful ones.

Rewrite of the state is handled by processing the packet header trough a *key extractor* that will now refer to the *update-scope*, the key thus obtained will be used to rewrite or add a new row in the state table. State updates can be performed also by the controller similarly to flow-mod, for this reason we name them as *state-mod* messages.

3.1.2 Configuration

We assume that by default all the flow tables that a switch provide for the pipeline processing are intended as stateless (i.e. standard Openflow). The controller can hence enable stateful processing for one or more *flow table* by sending a special control message to the switch. Configuring a stateful stage is made by associating a state table to an existing flow table and defining the *lookup-scope* and the *update-scope*. Obviously, the two *lookup-scope* and *update-scope* must provide same length keys, which is coherent with the definition of XFSM on a homogeneous set of flows.

Once the stateful stage has been configured, the controller can proceed installing entries in the flow table that will now match also on the current state of the flow. It is important to note that a complete description of the XFSM can be find just looking at the set of flow entries installed in the flow table as a combination of event and state matching, state transitions, and actions.

3.1.3 Support for multiple XFSMs

Multiple XFSM programs operating on different lookup scopes can be trivially configured using pipeling of multiple stateful stages. More interesting is the case of different XFSMs that must be configured on a same scope. As an example, in the port knocking example we could wish to have a set of addresses, say those originated from the subnet 131.175/16, for which we would like to have a different knock sequence, or even port 22 opened by default, without going through the knocking process. This can be easily accomplished by adding to the State Table the ability to match prefixes (e.g. match IPsrc=131.175.*.*), and use priority ordering (or longest match) to determine the matching to be used for retrieving an associated state.

3.2 Software datapath implementation

Being a more compelling HW implementation a much longer term goal, we tried to gain further insights by developing a prototype software implementation. We extended the Openflow 1.3 software switch [18] with our proposed stateful operation support. Our implementation is available at [19], so we limit to summarize here the main modifications (very few, as a further proof of the low impact of our proposal).

To support advertisement and configuration of the proposed state management feature, a new switch capability bit `OFPC_TABLE_STATEFUL` has been defined, as well as a new table configuration bit `OFPCT_TABLE_STATEFUL`. The basic flow table data structure has been extended with support for the state table and key extractors. A new Openflow instruction `OFPIT_SET_STATE` has been added to allow the Openflow extended datapath to update the state table with a given next-state parameter. A new state modify messages called `OFP_STATE_MOD` have been defined along with the relevant message structure to allow the controller to respectively configure the state entries and key extractors (lookup-scope and update-scope). As already briefly anticipated, the ac-

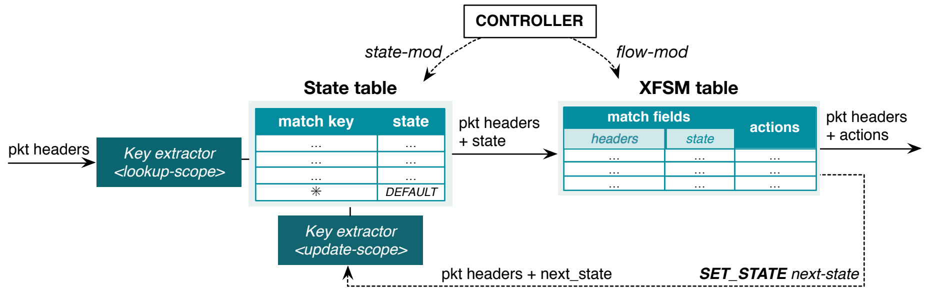

Figure 3: Architecture of the stateful stage. The XFSM table is represented by a standard Openflow table, while a SET_STATE action is used to trigger updates on the state table.

tual implementation and configuration of the XFSM table has not required any modifications to the existing code as it simply relies on the standard Openflow match table structure and flow-mod message (beside the already discussed support for the new OFPIT_SET_STATE instruction).

4. BEYOND THE BASIC ABSTRACTION?

While outlining in the previous sections our basic idea, we tried to remain grounded to the current OpenFlow functionalities, so as to hopefully convince the reader that what we do propose is not futuristic, but can be readily deployed. In what follows, we abandon prudence and we try to outline (with no pretense of making any firm claim, but rather with the goal to stimulate discussion) how the *same* programmability model could be extended along complementary directions, so as to provide device programmers with further network function programming abilities.

We point out that we do not claim our propose is able to implement all possible functionalities that are currently supported by complex middleboxes. Nevertheless we expect that some of their less complex functions can be shifted to switches for a more responsive reaction of the network.

4.1 Improving state handling

Soft states and event timers. Adding timeouts as described in OpenFlow to a state table entry is straightforward and the API could be extended to permit the programmer to specify a different timeout for each state transition. Managing timeout expiration is also trivial, but only if we do assume that *all* states return to the DEFAULT one upon timeout expiration. Indeed, timeouts could be *implicitly* managed by assuming that a state lookup which retrieves an expired state shall return a DEFAULT state. Rather, handling timer expirations as *explicit events*, which trigger meaningful (non default) transitions, might open very interesting scenarios (support for exponential backoff operation, enforce a different TCP forwarding on the basis of whether an ACK returns before or after a time window, etc), but arguably requires a significant leap in the implementation.

Using state labels as function parameters. The MAC learning example illustrated in Table 1 requires an entry for each possible port of the switch. By permitting the forwarding action to receive as input a parameter provided in the meta-data associated to the packet (in our specific case, the state label interpreted as a switch port number) it is possible to implement the same program with only two entries: one for the default state (MAC destination not found in the forwarding database) and one which forwards the MAC frame to the switch port found in the state label. Note that efficient technical ways to pass (extended) header parameters to actions have been recently discussed in [15].

Simple arithmetic operations on labels. The combination of states and simple arithmetic operations permits several interesting extensions. For instance, we could trivially program a state machine which thwarts some IP fragmentation attacks by forwarding IP fragments only if they are received in strict order. We recall that a very basic IP fragmentation attack consists in sending a first small fragment (fragment offset = 0, more fragment = 1), and then send a second IP fragment claimed to be the last small fragment of a large (64 KB) packet. This could be easy detected by "computing", from the length of the first IP fragment, the expected offset for the second fragment, use it as a temporary state label (or a numerical value associated to a "fragment" state), and forward packets only if they match such a computed offset field (i.e. if they are in sequence). We stress that, albeit apparently compelling and possibly inspiring a broad range of extensions, such "arithmetic computations" may become critical at high speed, and a much closer look at their viability is needed.

4.2 Enforcing conditions: "full" XFSMs

Flow statistics can be considered as "memory registers" associated to a flow. Similarly, device-level states, such as the current occupancy of an output queue, or device level statistics, such as the amount of bytes delivered through a given switch port, can be as well interpreted as "global registers". And, finally (as implied in the previous section), the association of a numerical value to a state can be interpreted as a value stored in a per-flow register. The values stored in such "registers" could be used as further *conditions* to trigger an action associated to an event.

Quite interestingly, the definition of eXtensible Finite State Machine given in [10] provides a formal model which generalizes the Mealy Machines introduced in section 2, and which permits to explicitly accounts for *conditions* taken on registry values. And, indeed, the ability to set conditions in an XFSM was actually proven to be vital in [8], for formally specifying wireless MAC protocols.

We recall from [10] than an XFSM is an abstract 7-tuple (S, I, O, T, D, U, F), where the states S, the input events I and the output action O are the same as defined in section 2, and where:

- D is an n-dimensional linear space $D_1, \cdots D_n$ which describes all possible configurations of n "registers";
- F is a set of *enabling functions* $f_i : D \rightarrow \{\text{TRUE}, \text{FALSE}\}$ which models *conditions* to be verified on the configuration registers to enable transitions;

- U is a set of *update functions* $u_i : D \to D$ which permits to model changes in the deployed registers; and
- $T : S \times I \times F \to S \times U \times O$ is a transition function which takes as input the current state, event, and conditions, and outputs i) the next state, ii) the associated action, and iii) the associated registry update.

We argue that under the (restrictive) condition of predefined (hard coded in the device and exposed via the programming interface, rather than freely programmed) set of available "registers", "enabling functions", and "update functions", this abstraction appears at reach with current technology and hence promising to be explored further. Indeed, conditions could be exploited in many scenarios, such as QoS (differentiated packet treatment when above a given count threshold - currently addressed in OpenFlow with a dedicated new extension, the meters), load balancing (set forwarding port for a new flow, based on queue status or link load statistics), monitoring, and so on.

5. CONCLUSIONS

This paper aims to propose a first step in the direction of supporting stateful per flow processing over closed platforms. In our proposal, and in full adherence with the OpenFlow strategy, we took a very pragmatic approach: we compromised on generality and we "restricted" to the stateful handling of *standard* OpenFlow match/action rules. This permitted us to emerge with an apparently immediately deployable programmatic abstraction, relying on core primitives and data structures mostly present in OpenFlow implementation. Our abstraction generalizes the OpenFlow match/action rules in terms of an extensible finite state machines, which are *directy executed inside the switching device*, thus offloading controllers and, perhaps more interestingly, entailing control functions which require wire-speed, packet-by-packet, operation, i.e. which cannot be delegated to the slow (logically) centralized control plane operation.

As a compromise, we obviously have *no pretense* to claim that our abstraction can support *all* (!) the possible flow processing needs. Nevertheless, we believe that stateful handling of OpenFlow rules can be beneficial in many scenarios (some of those illustrated via use case examples), and we hope to stimulate a broader discussion on the many questions that our paper opens (high speed implementation should restrict to Mealy Machines, or could support more general XFSMs? Which network processing function, today implemented in the controller or in dedicated middleboxes, can be described using XFSMs? And how the ability to dynamically control flow states directly in the device may influence broader SDN frameworks?).

6. 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 Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *ACM Queue, To appear*, 2014.

[3] K. Greene, "TR10: Software-defined networking, 2009," MIT Technology Review, available online at http://www2.technologyreview.com/article/412194/tr10-software-defined-networking.

[4] O. N. Foundation, "Openflow switch specification ver. 1.4.0," in *Oct 14, 2013*.

[5] B. Mack-Crane, "Openflow extensions," in *US Ignite ONF GENI workshop, October 8, 2013*.

[6] D. Meyer, "Openflow: Today's reality, tomorrow's promise? an architectural perspective," in *SDN Summit 2013, Paris, France, March 2013*, 2013.

[7] P. Peresini, M. Kuzniar, and D. Kostic, "OpenFlow Needs You! A Call for a Discussion About a Cleaner OpenFlow API," in *2nd EU Workshop on Software Defined Network (EWSDN)*, 2013.

[8] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware." in *IEEE INFOCOM*, 2012, pp. 1269–1277.

[9] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, and I. Tinnirello, "MAClets: active MAC protocols over hard-coded devices," in *8th ACM CoNext '12*, 2012, pp. 229–240.

[10] K. T. Cheng and A. S. Krishnakumar, "Automatic Functional Test Generation Using The Extended Finite State Machine Model," in *ACM Int. Design Automation Conference (DAC)*, 1993, pp. 86–91.

[11] Z. Qazi, C.-C. Tu, R. Miao, L. Chiang, V. Sekar, and M. Yu, "Practical and incremental convergence between sdn and middleboxes," in *Open Network Summit, Santa Clara, CA, April 2013*, 2013.

[12] G. Gibb, H. Zeng, and N. McKeown, "Initial thoughts on custom network processing via waypoint services," in *WISH - 3rd Workshop on Infrastructures for Software/Hardware co-design, CGO 2011, April 2011, Chamonix, France*, 2011.

[13] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *HotSDN 2013 (Hot Topics in Software Defined Networking)*, 2013.

[14] M. e. a. Ciosi, "Network functions virtualization, ETSI White Paper." in *SDN and OpenFlow World Congress, Darmstadt, Oct. 22-24*, 2012.

[15] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM 2013*. ACM, 2013, pp. 99–110.

[16] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazieres, "Tiny packet programs for low-latency network control and monitoring," in *ACM Workshop on Hot Topics in Networks (HOTNETS 2013)*, 2013.

[17] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *1st ACM Workshop on Research on Enterprise Networking (WREN09)*, 2009.

[18] "OpenFlow 1.3 Software Switch," http://cpqd.github.io/ofsoftswitch13/.

[19] "Prototype implementation of an OpenFlow 1.3 software switch with XFSM support," http://bit.ly/Mte0zb.