Composition of SDN applications: Options/challenges for real implementations

Arne Schwabe
Paderborn University
Warburger Straße 100
33098 Paderborn, Germany
arne.schwabe@uni-paderborn.de

Pedro A. Aranda Gutiérrez
Telefónica, I+D
Zurbarán, 12
28006 Madrid, Spain
pedroa.aranda@telefonica.com

Holger Karl
Paderborn University
Warburger Straße 100
33098 Paderborn, Germany
holger.karl@uni-paderborn.de

ABSTRACT

In this paper we define the notion of composition for software-defined networking applications and show the theoretical and practical approaches to composition in software-defined networks and explain the challenges associated with it. We explore the feasibility of OpenFlow as an Application Programming Interface (API) for a composition engine and argue that its design as Southbound controller interface makes it unsuitable for this task.

1. INTRODUCTION

Software-defined networking (SDN) promises higher flexibility in the way networks are managed. However, by introducing software paradigms in networks, we also introduce the complexity of modern software systems in them. For example, the question of interfaces to control functions has to be addressed. Such control functions can be realized by agents – applications that access network devices – as proposed by Interface to the Routing System (i2rs) and other Working Groups (WGs) in the IETF. These architectures foresee multiple applications accessing the same device. Like in concurrent programming, a series of issues related with multiple access arises, including situations where several applications produce "conflicting" configurations. Current approaches to the "conflict" suggest that a generalized treatment is not possible. In this paper we provide a new framework to describe the interactions between applications and the network based on transactions and examine what approaches to composition may make the problem tractable.

We start by giving definitions in the next section and then explain the different approaches to composition in software-defined networks in Section 3. We will continue by presenting the general strategies to handle and implement composition in Section 4. We will look into the specific challenges of using OpenFlow in composition in Section 5 and how we addressed them in our implementation in following section. We look at related work that discusses composition in Section 7 and provide in Section 8 the conclusions and sketch future work.

ANRW '16, July 16 2016, Berlin, Germany

ISBN 978-1-4503-4443-2/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2959424.2959436

2. DEFINITIONS

For the scope of this paper, a software-defined network is a collection of interconnected nodes (switches) forming a graph $G = V(V, E)$, where E describes the connectivity between the switches. We define the state N_v of a node as the state of its Forwarding Information Base (FIB). A FIB entry is defined as tuple (p, m, i) specifying a priority, the packets to match and a list of instructions i.[1] The network state N is then defined as the collection of all node states

$$N = \{N_v \mid v \in V\}$$

The network state can be changed by a command C. A command is a sequence of basic commands $C = [c_1, c_2, \ldots]$ with $c_j \in \{F_{\text{inst}}, F_{\text{del}}\}$ that modify the FIB of a switch:

- $F_{\text{inst}} = (v, p, m, i)$: Install a FIB entry on node v with priority p, match m and list of instructions i.
- $F_{\text{del}} = (v, p, m, i)$: Remove the FIB entry that was installed by $F_{\text{inst}}(v, p, m, i)$

We define the function $a\colon N \times C \to N$ as the function that applies a network command to a network state and produces the new network state:

$$a(N, [c1, \ldots, c_k]) = a(a(N, c_1), [c_2, \ldots, c_k])$$

$$a(N, c_j) = \begin{cases} \text{Add (p,m,i) at } v & \text{if } c_j = F_{\text{inst}} \\ \text{Remove (p,m,i) at } v & \text{if } c_j = F_{\text{del}} \end{cases}$$

for a basic command $c_j = (p, m, i, v)$.

In a network, these commands are generated by a control application or *module* M_j. We define a network module as a state-based function M_j that reacts to an event ev with a network command (and possibly modifies its internal state):

$$M_j\colon ev \to C$$

Examples for network events are the arrival of certain packet types (like ARP request) or the arrival of a new flow currently not handled by the FIB.

[1] A typical FIB entry used in IP forwarding is (`100, if {ingress_port==*, dst_ip∈{192.168.100.0/24}}, set {src_mac 00:00:00:ab:cd:ef, dst_mac aa:bb:cc:00:11:22, egress_port 3}`).

3. TYPES OF COMPOSITION

The goal of composition is to run multiple modules on the same physical network and incorporate all their network commands into the network state.

3.1 Single module without composition

We start with the simplest case of a single module M_1. All commands are simply forwarded and applied to the network and the resulting network state N' is:

$$N' = a(N, M_1(ev))$$

3.2 Multiple modules without composition

A typical SDN controller runs multiple SDN modules and, commonly, all outputs of these modules are applied to the network without any explicit form of composition. In this case, each command is applied to the network as it happens, just like in the single module scenario. The resulting network state $N^{(k)}$ for multiple applications looks like this:

$$\begin{aligned} N' &= a(N, M_1(ev)) \\ N'' &= a(N', M_2(ev)) \\ N^{(k)} &= a(N^{(k-1)}, M_k(ev)) \end{aligned}$$

The simplicity of this approach is also its biggest problem. Since every network command is applied when it happens, the results depends on the order of transaction applied, $a(a(N, M_1(ev)), M_2(ev)$ is not necessarily the same as $a(a(N, M_2(ev), M_1(ev))$. But cases might occur in a non-deterministic fashion in the same network, for example caused by differences in execution speed of modules M_1 and M_2.

Another problem are transient states. In the time after the first module has answered but not the second, the transient network state N' is active. This transient is problematic since it only reflects the output of first module but not the others. These ill-specified, non-deterministic transient networks states are usually undesirable and constitute the main reason to explicitly define a composition logic.

3.3 Multiple modules with harmonizing

The output of the multiple modules might contain conflicting or overlapping commands. For example, two modules might instruct one switch to deal with the name packet by either forwarding or dropping it, at the same priority (see Section 4 for details). To deal with such conflicts, a stateful function

$$h\colon \text{command} \to \text{command}$$

to modify the commands can be used. An example for such a harmonizing function is a network hypervisor. The network state can be expressed as:

$$\begin{aligned} N' &= a(N, h(M_1(ev))) \\ &\vdots \\ N^{(k)} &= a(N^{(k-1)}, h(M_k(ev))) \end{aligned}$$

When the harmonizing function h is the identity we get the same result as in the previous subsection.

This harmonizing function can deal with some problems, but intermediate, hard-to-predict network states still exist. It is hence not a satisfying solution.

3.4 Parallel composition

To overcome the problem with transient state and varying order of applied results, parallel composition collects all commands and then resolves all conflicts betweens these commands, composing the results into a *single* command to be applied to the network. This is done by a special *resolving function* r that gets *all* command outputs and generates a conflict-free version that can be applied to the network.

$$r\colon \text{command} \times \ldots \times \text{command} \to \text{command}$$

The new network state N' can then be expressed as:

$$N' = a(N, r(M_1(ev), M_2(ev), \ldots, M_k(ev)))$$

This composition requires all command outputs of an event to be available; it must also be possible to tie a command output of a module to a specific event (necessary when multiple events are passed to a module before commands have been produced, compare challenges of OpenFlow, Section 5).

A big difference between the parallel composition and the harmonizing composition is that the parallel composition is reactive, i.e. it depends on the fact that the network commands generated by the modules are a response to a network event. The harmonizing composition works without this assumption and can also be applied to network commands that are sent proactively without an event ($C = M_j(\emptyset)$).

3.5 Serial composition

For the serial composition, one module is fed the output of a previous module. The desired network state of the first module only exists as an input to the second module.

To really support this behavior, we need to change the signature of the modules. They must accept a command as an additional input:

$$M\colon \text{event} \times \text{command} \to \text{command}$$

With that, we can define a new function

$$M_{12}\colon ev \mapsto (M_1 \circ M_2)(ev, \emptyset)$$

and use that in place of a normal module function (in the harmonizing composition or parallel composition). For example, if only the serially composed function is used, the new network state will be:

$$\begin{aligned} N' &= a(N, M_{12}(ev)) \\ &= a(N, (M_1 \circ M_2)(ev, \emptyset) \\ &= a(N, M_2(ev, M_1(ev, \emptyset))) \end{aligned}$$

Chaining more than two network modules, e.g. $M_1 \circ M_2 \circ M_3$, is defined by obvious induction.

Network programming languages like Pyretic [1] also define their function signature to have symmetrical input and output: $f\colon \text{policy} \to \text{policy}$.

The main distinction of the serial is that the last module in the composition chain can provide consistent network commands. It also allows to a module to incorporate the decisions of a previous module into its own decision. The downside of the serial composition is however that module need to explicitly designed and programmed to be used in this way. We will take a look how useful serial composition is with existing module in the next section.

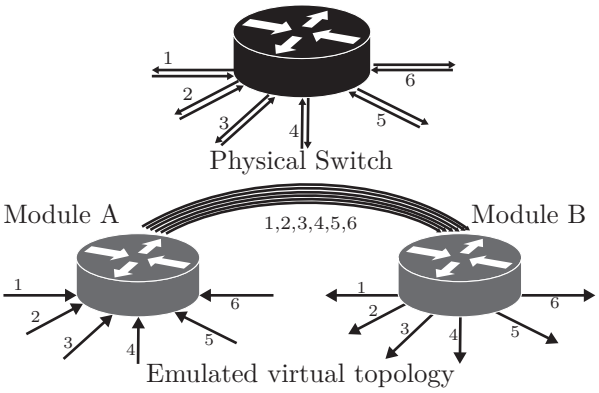

Figure 1: Using a virtual overlay network for composition of module A and B

3.6 Approximate serial composition

As most network modules are not designed for serial composition (i.e., they do not accept a command as an input), we define an approximate way to do serial composition with existing network modules. In this scenario, we need to incorporate as much as possible from the network command into the input event of the following module by a function

$$\alpha\colon \widetilde{N} \times \text{command} \rightarrow \text{event}$$

where $\widetilde{N}$ is the *approximated* network state resulting from applying the output of the first function to the current network state. This approximated state is a representation of the state in the controller; its manipulation does *not* involve manipulation of actual state in network devices.

What can be incorporated into the new event is often very limited as we will see in Section 5. The new network state using this function can be expressed as:

$$\begin{aligned} N' &= a(N, (M_1 \tilde{\circ} M_2)(ev)) \\ &= a(N, M_2(\alpha(\widetilde{N}, M_1(ev)))) \end{aligned}$$

Similarly, chaining three modules in an approximate serial composition works as well:

$$\begin{aligned} N' &= a(N, (M_1 \tilde{\circ} M_2 \tilde{\circ} M_3)(ev) \\ &= a(N, M_3(\alpha(\widetilde{\widetilde{N}}, \alpha(\widetilde{N}, M_1(ev))))) \end{aligned}$$

where $\widetilde{\widetilde{N}}$ is the approximated network state resulting from applying $M_1(ev)$ to $\widetilde{N}$.

Using an Overlays for approximate serial composition.

A conceivable variant to implement this approximation of the network state and the function α is to use an overlay of virtual switches to a real network as shown in Figure 1: For each physical switch, a number of virtual switches corresponding to the number of modules is emulated. Each module is assigned to one virtual switch. The approximated network states are the state of the virtual switches and the function α would "process" the packet that the module sends to its virtual switch output ports as event for the next module.

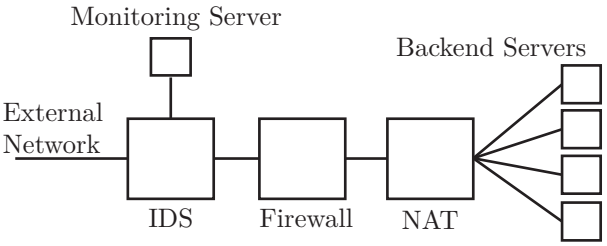

Figure 2: Typical order of IDS and Firewall and NAT load balancer middle boxes in a traditional network

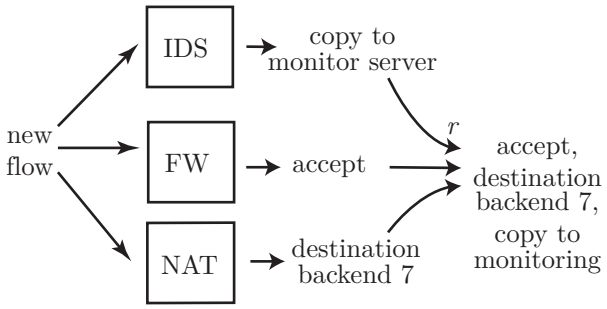

Figure 3: Parallel composition of IDS, FW and NAT load-balancer modules

3.7 Composition and order of middle boxes

A common misconception is that the placement of middleboxes (a networking device that transforms, inspects, filters, or otherwise manipulates forwarded traffic) always carries over to the composition order. If multiple middleboxes, for example a firewall and a monitoring/IDS system should act on all the same traffic, these boxes are setup in sequence to pass traffic to box after another. Figure 2 show an example of a traditional middle box setup with an IDS, a firewall and a load balancer.

In stark contrast, for a composition the modules would be typically setup in a parallel composition to allow all modules to base their decisions on decide on the original input packets. The merging process of all the outputs will then give an equivalent solution to the middle box solution. Fig 3 shows the setup of Fig 2 implemented with parallel composition.

4. COMPOSITION STRATEGIES

For the "true" serial composition, the mechanics required of the composition framework are simple and implemented in the modules themselves. For the more challenging approximate serial composition we will discuss strategies in the OpenFlow implementation section.

For the remainder of this section, we look at some general composition strategies to implement the resolving function r for parallel composition. We concentrate on resolving multiple flow install commands since it is the most interesting composition part and the ideas used here can be used analogously for resolving other basic commands.

The idea here is to handle as much as possible in a generalized way but allow to fall back to developer-specified logic where a general approach cannot work. For this strategy,

the function r performs the following steps:

1. Check for syntactic and general conflicts
2. Check for developer-specific conflicts using user logic
3. If no conflicts detected, perform generic composition
4. If a generic composition is not possible, abort or call developer-provided conflict resolution

The first step is to check for conflicts. If two commands do not act on the same switch, they do not conflict. Also, if one command has a higher priority than the other command, the one with the higher priority wins. So from now on, we will consider two FIB install commands that act on the same switch and have the same priority.

Both basic commands c_1 and c_2 have a match m_1 and m_2, which typically are not identical. Hence, we have three different matches to consider for the composition. The match for packets matched only by $m_{1*} = m_1 \setminus m_2$, the analogues match $m_{2*} = m_2 \setminus m_1$, and the match for packets that are matched by m_1 and m_2: $m_{12} = m_1 \cap m_2$. For the generalized approach, we assume that network modules respond to a new flow install with a FIB install command that also matches the new flow. It directly follows that the common match m_{12} is not empty and only for the common match m_{12} we have instructions from both modules for the new flow of the event. As an example, one module might want to install policies per IP address while the other module installs policies per network. For the generalized approach, we therefore opt to ignore the matches m_{2*} and m_{1*} and only generate a new FIB install command for the composed rule on m_{12}. A new flow that falls under the match m_{1*} or m_{2*} will trigger a new flow event and we restart the composition with its new flow event.

For the instruction list of the install command, the general idea is to combine both instructions lists into one big list of instructions. When combining these lists, we can encounter different conflicts in the combined list. We differentiate these into semantic and syntactic conflicts. Syntactic conflicts can be automatically detected, like two instructions setting the same fields to two different values. As an example, a misconfigured composition enables two load-balancing modules and both try to rewrite the destination IP address of a packet to two different server IP addresses. Different Instructions can also be mutually exclusive, like removing the VLAN tag and the same one changing the VLAN id. Or dropping the packet and any other action that modifies the packets. These syntactic conflicts can be detected by the generalized approach.

Semantic conflicts, in contrast, are not automatically detectable by a general approach but still cause problems. Assume again two load-balancing modules: the first module tries to redirect to a different port but leaves the IP address unchanged and the second module sets a different IP destination address. Since no syntactic conflict exists, the actions list can be merged and will redirect the packets to an IP/port combination that will not work. The only way to detect such conflicts is to call developer-provided logic.

5. COMPOSITION WITH OPENFLOW

OpenFlow is the most commonly used protocol used in real-world deployment and a lot of existing application logic is implemented using OpenFlow protocols. This makes OpenFlow desirable as a protocol on top of composition and conflict resolution. On the other hand, OpenFlow itself was never designed to used in a composition context. The implicit assumption that there is only one entity controlling an OpenFlow device[2] makes its use problematic.

This problem is aggravated by the fact that OpenFlow is not only used as a control protocol for switches as the southbound interface. Instead, its semantic has also left its mark on the design of northbound interfaces, which often more or less directly mirror the OpenFlow semantics. In this section, we will analyze the problems of OpenFlow in composition and conflict detection and detail how and to what degree they can be avoided and solved.

5.1 Definitions

We will briefly show the definition of the important packet types in OpenFlow for composition:

Packet_In The `PACKET_IN`, abbreviated `PKT_IN`, is the main event in OpenFlow and usually signifies the arrival of a new flow. Whenever a packet arrives at a switch that is not handled by one of the FIB entries (or a FIB entry explicitly states to generate a `PKT_IN`) a copy of the packet and the meta information of the packet (ingress port, etc.) are forwarded to the controller.

Flow_Mod The `FLOW_MOD`, abbreviated `FM`, is the OpenFlow command that is analogous to our FIB entry install command F_i.

Packet_Out The `PACKET_OUT`, abbreviated `PKT_OUT`, allows an OpenFlow controller to craft and send a packet to the network. A typical use case for this is to reply to an ARP Request. The `PKT_OUT` consists of a packet and action list that is identical in function and syntax to the `FM` action list.

5.2 Multiple modules

The "multiple modules" approach without harmonization (Section 3.2) is easy to support with OpenFlow. Adding a harmonizing function (Section 3.3) is possible, but requires to intercept `FM` commands before sending them to the network. Depending on the specific controller architecture, this is a more or less easy task. Correctly treating timeouts of FIB entires is also not a trivial task. Hence, even the first non-trivial composition approach is not entirely straightforward to support.

5.3 The run to completion problem

In the previous sections we defined the parallel composition to combine all commands triggered by the same network event. The definition of network events in OpenFlow is straightforward and consists of a small list of unsolicited messages of which the most important one is the `PKT_IN` event.

Unfortunately, in OpenFlow there is no relationship between a network event and the responses of a controller and

[2]OpenFlow does allows multiple connections per switch from multiple hosts for load sharing/backup of a single (distributed) entity.

thus also no reliable way to tie the responses obtained from a module to the original network events. PKT_OUTs may reference the original PKT_IN as optimization to avoid copying the packet but this captures only a fraction of the PKT_OUTs. Also, there is no way to tell if an OpenFlow module will respond to an event at all.

Hence, the basic assumption of composition – actions can be tied to events across multiple modules – is not guaranteed by OpenFlow.

5.4 Parallel composition

If ignoring the (major) run-to-completion problem, implementing a resolve function works as sketched in the last section.

The PKT_OUTs that are also generated as port of the output, a generalized solution is not possible since is standard approach to combine two arbitrary Ethernet packets into one. Here again either a user logic is needed or a simple approach that prefers packets from one module and drops packets of other modules if more than one packet out is present.

5.5 Serial composition

With OpenFlow we can at best try to achieve approximate serial composition – actual serial composition is impossible as an OpenFlow-oriented northbound interface cannot express both events and commands as input.

The input event in OpenFlow is the PKT_IN. The goal is to create a PKT_IN that carries as much information from the outputs (PKT_OUT and FLOW_OUT) of the previous module as possible.

The generated packets and functions involved in an OpenFlow serial composition chain with two modules looks like this:

$$\begin{aligned} \texttt{PKT\_IN}_0 &\rightarrow M_1 \rightarrow \texttt{PKT\_OUT}_1, \texttt{FM}_1 \\ &\rightarrow \alpha \rightarrow \texttt{PKT\_IN}_1 \\ &\rightarrow M_2 \rightarrow \texttt{PKT\_OUT}_2, \texttt{FM}_2 \end{aligned}$$

The meta-information part of the new $\texttt{PKT\_IN}_1$ (produced by the network emulation function α) is a match that only carries the input port and no other information. The input port is usually the same as the input of the $\texttt{PKT\_IN}_0$ unless an overlay composition is used in which case the input port is the output port designated by the first module.

For the packet part we have two options: (1) Modify the original packet of the original $\texttt{PKT\_IN}_0$ or (2) use the packet of the $\texttt{PKT\_OUT}_0$ if there is any and fall back to the input or stop the chain if there is none.

When choosing the first option and using the packet of $\texttt{PKT\_OUT}_1$, we can assume that all actions are either already applied or are in the action set of the packet out. As consequence, we will ignore $\texttt{FM}_1$ in this case. If we decide to use the packet of the original packet of $\texttt{PKT\_IN}_0$, we can apply the actions of $\texttt{FM}_1$ to it and thus ignore $\texttt{PKT\_OUT}_1$. No matter what option we choose, we always ignore a significant part of M_1 output.

In both cases we have to apply the instructions of the PKT_OUT or FM to preserve as much information as possible. Only the subset of instructions that mutates the packet itself (like adding a vlan id) can be preserved. Everything that is not directly related to the content of the packet cannot be represented in the new packet, which includes instructions like setting the output queue, rate limits, goto table x, etc.

The workaround to preserve the information contained in instruction is to remember them and then merge/intersect all actions from all modules of a sequential composition in the last step. But this creates an unintuitive, difficult to understand and predict hybrid between serial and parallel composition.

In summary, all these problem with generating a new PKT_IN make sequential composition in an OpenFlow only usable in very limited circumstances.

In a wider sense, we can conclude that *an OpenFlow-oriented northbound interface is ill suited to support composition of control modules!*

6. IMPLEMENTATION

The goal of the FP7 project NetIDE has been to implement composition for SDN application while supporting legacy controllers as much as possible. The NetIDE project has focused on OpenFlow and tried to overcome or work around the limitation of OpenFlow identified in the previous section.

In this project, we encountered many of the challenges described so far and made a number of restrictions and protocol enhancements to obtain a practically usable composition. We will give a short overview:

Intermediate Core.

In order not to modify legacy controllers, we collected all necessary functions for composition in a "Core". This core intercepts, among others, FM messages before handing them on to the actual network devices. Depending on which actual controllers is used (e.g., ONOS, Floodlight, Ryu), the core interacts with a controller-specific "backend" that realizes the actual interception of messages; this allows the core to stay controller-agnostic.

NetIDE protocol.

A custom protocol used between the core and the backends. It carries additional messages as well as the normal OpenFlow messages.

Fence messages.

To work around the "run to completion" problem (Section 5.3), we introduce *fence messages* to signal the end of a response to a network event.

Transaction Ids.

To be able to tie responses to a network event, we added a transaction id field to the our protocol that tracks events and their responses.

Concentrate on parallel composition.

Since serial composition semantics are difficult to define and implement, the current implementation is focused on parallel composition.

Restrict OpenFlow control module behaviour.

To allow all of the features to work, NetIDE imposes restriction on the modules' use of OpenFlow. A module must reply to events only and should not use proactive flows.

Modules also must not make assumption on the state of the network, e.g., they need to always reply to PKT_IN and must not assume that previously installed flow rules already

handle the flow.

For the serial composition we require always a `PKT_OUT` and a `FM` as output of all but the last module in the chain.

7. RELATED WORK

The idea of module composition in software-defined networking (SDN) is not new and has been presented in various forms. One of earliest form is FlowVisor[2], which partitions the network in different slices and also OpenVirtex [3], which improves on this approach. In our terminology, the slicing of the network is a harmonizing function that avoids conflicts by making the matches of all modules disjunct.

To focus on the composition itself, approaches like Frenentic/Pyretic [4, 1] introduce a functional programming language specifically aimed at SDN policy description. The languages were carefully designed to allow parallel and serial (sequential) composition on individual statements, for example by choosing a function signature that has a policy definition as input as well as output. This approach avoids the challenges and incompatibilities we faced when trying to use OpenFlow as basis for composition, but it pays the price of mandating a new programming paradigm.

Using OpenFlow as API for the composition is proposed by CoVisor [5]. CoVisor uses virtual switches to implement the serial composition. The CoVisor paper, however, leaves out many details how the challenges that we outlined here, most importantly the run-to-completion problem, are solved by CoVisor. Moreover, the available implementation of CoVisor only implements a very limited subset, namely a static composition that gets all network commands at the start of the program and thus avoid all the challenges with dynamic events/network commands. It is hence difficult to ascertain how CoVisor actually intends to address the difficult problems in composition.

Finally, the approaches used by SDN test tools such as SOFT[6] and NICE [7] to detect bugs in controllers can also be applied to enhance the semantic conflict detection.

8. CONCLUSIONS AND FUTURE WORK

Composition can be very useful tool if used in a restricted environment. We have also shown that for composition to work in a meaningful way, the underlying south-bound interface (SBI) should be designed to support it. While OpenFlow can be augmented and restricted to work for composition, doing so results in a customised protocol that, even then, is still lacking several aspects. The most notable aspect is that the OpenFlow module will get no feedback from composition about the conflict resolution result.

As a main conclusions, we point out that an interface between controllers and control modules that is oriented towards OpenFlow *is not suitable to support any but the most trivial composition semantics.* The challenge is to find an interface that carries enough information (commands and events in our parlance; policies in Frenetic lingo) between modules, but hopefully without having to mandate its own programming style as done by Frenetic/Pyretic.

To make practical progress, we are in contact with the i2rs WG in the IETF. Potentially, this could lead to an interface between modules and controllers that is better geared towards supporting composition.

From a practical perspective, this paper and the implementation work in NetIDE concentrates on the *parallel composition*. However, there are situations where it is more advisable to generate an initial configuration beforehand to put network elements in a known state. This behavior is known as *proactive applications*. This behavior can be supported by either implementing a harmonizing approach for these messages or treat the initial configuration of modules with a special "initial" event.

Acknowledgments

The work presented in this paper has been partially sponsored by the European Union through the FP7 project NetIDE, grant agreement 619543.

9. REFERENCES

[1] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN Programming with Pyretic. *USENIX ;login*, 38(5):128–134, Oct. 2013.

[2] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[3] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. OpenVirteX: Make your virtual SDNs programmable. In *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[4] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, 2010.

[5] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 87–101, Oakland, CA, May 2015. USENIX Association.

[6] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. A SOFT way for OpenFlow switch interoperability testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012.

[7] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.