Reusability of Software-Defined Networking Applications: A Runtime, Multi-Controller Approach

Roberto Doriguzzi-Corin[α], Pedro A. Aranda Gutiérrez[β], Elisa Rojas[γ], Holger Karl[δ], Elio Salvadori[α]
[α]CREATE-NET, [β]Telefónica, I+D, S.A.U., [γ]Telcaria Ideas S.L., [δ]University of Paderborn

***Abstract*—The Software-Defined Networking (SDN) ecosystem is still characterized by a multitude of different controller platforms, each with its own programming model, execution model, and capabilities. This creates a danger of a *controller lock-in* for both developers of SDN control applications and operators of SDN networks. Since no single controller platform appears to dominate the ecosystem for the foreseeable future, there is a need for portability of control applications between different platforms. We propose an architecture based on executing multiple instances of different controller platforms concurrently in a network to provide the SDN code the environment it was written for. It is built around a controller-independent network event routing element called Network Engine that provides composition and conflict resolution. Results obtained in realistic scenarios demonstrate the feasibility of the proposed approach, which increases both developer productivity and operational flexibility. A preliminary prototype of the architecture is available for testing as an open source project.**

I. Introduction

The current SDN ecosystem is still extremely fragmented. Different open and closed-source controller frameworks such as OpenDaylight (ODL) [1], Open Network Operating System (ONOS) [2] or Ryu [3] coexist and compete in the controller plane. Thus, network control application developers and administrators need to choose a specific platform, which might not fulfill their requirements completely. This also introduces uncertainty, because it is not obvious what platform will prevail in the foreseeable future. However, diversity in an ecosystem is always an asset and, therefore, easy porting of SDN control applications between controller platforms is a desirable target.

With this premise, the NetIDE project [4] funded by the European Commission has set out to introduce a new approach to SDN application development, where (a) applications can be reused as building blocks for more complex applications (i.e. introduce modularity), and (b) existing applications can be deployed across different controllers, becoming independent of a particular controller choice. These goals should be achieved without cost at development time; some additional overhead at deployment or runtime might be acceptable. In summary, NetIDE aims at facilitating Network Operators/Administrators to perform rapid deployment of existing SDN applications belonging to different controller frameworks directly on their operational SDN-based infrastructure.

The aforementioned goals, i.e. re-usability for network applications and cross-platform support, have implications for the deployment and runtime mechanisms of our architecture. We propose a novel SDN component called **Network Engine** to coordinate multiple SDN controllers, each executing one or more network control applications and typically based on heterogeneous controller frameworks. These controllers act as clients of a server controller, which controls the network equipment in the network domain.

In the literature, several solutions have been proposed to handle Network Applications running on the same infrastructure but implemented for different SDN control platforms. Hypervisors like FlowVisor [5] and OpenVirteX [6] split the traffic into "slices", thus impeding multiple applications to cooperate in processing the same traffic as a single Network Application. CoVisor [7] brings together the following features: (i) assembly of multiple controllers, (ii) definition of abstract topologies and (iii) protection against misbehaving controllers. In particular, it allows the administrator to combine different client controllers in parallel, in sequence, or in an override/default relationship. However, it has several limitations, e.g. it makes difficult the adaptation of an SDN operational set-up by forcing the replacement of a running SDN controller with a network hypervisor; it can not recognize when an SDN application has finished processing a network event, thus potentially leading to network deadlocks.

FlowBricks [8] is designed to compose client controllers, but it only runs on a emulated environment with heavy hacks on the OpenFlow switches and cannot be used over *standard* network hardware. Corybantic [9] acts as a module orchestrator, but it requires SDN applications to interface with it through specific Application Programming Interface (API), forcing the administrator to modify the applications s/he wants to reuse from different SDN environments.

Compared to such existing approaches, the proposed Network Engine combines *unmodified* SDN applications running on server and client controllers, thus organizing how actions are processed by the different controllers. The Network Engine generalizes the notion of "chaining" supported by some controllers, where individual SDN modules can transfer network events to other modules in the same controller, effectively creating chains of modules that handle an event. We generalise these chains by (a) supporting them with explicit and flexible composition semantics, which (b) the Network Engine can execute even across multiple controllers. Hence, we allow a *Network Application* to consist of modules written for different controller frameworks.

Beyond demonstrating a fully working architecture leveraging on open-source SDN controller platforms, we have made accessible to the general public both the functional specifications as well as some platform-independent libraries to facilitate the implementation of any Network Engine component by any vendor willing to adopt such architecture in a closed-source commercial SDN environment.

The results obtained on a set of tests performed in realistic scenarios demonstrate that the additional overhead introduced by the Network Engine at deployment or runtime is acceptable. In fact, reusing previously developed SDN applications and components, instead of developing them from scratch for every scenario, will increase both developer productivity and operational flexibility.

978-3-901882-85-2

II. NetIDE Network Engine architecture

A high-level view of the Network Engine's architecture is represented in Fig. 1. The Network Engine provides to unmodified SDN applications (called *Modules* in the figure) the runtime they expect. Additionally, it implements a composition mechanism to create new applications from previously implemented modules and to make them cooperate with modules running on top of the operator's controller (represented by the *Server Controller* in the figure), effectively building a single *Network Application*. The latter aspect is the main contribution of this work and sets NetIDE apart from similar approaches.

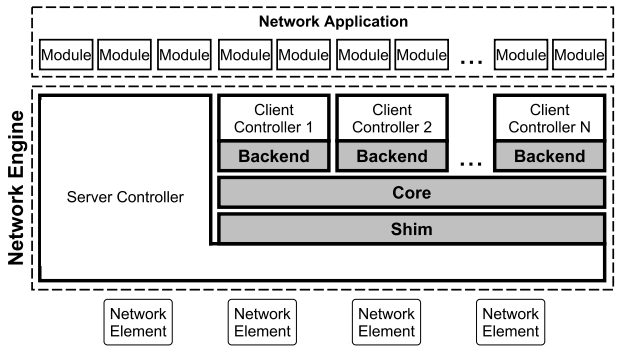

Fig. 1: Network Engine architecture.

Our Network Engine follows the layered SDN controller approach proposed by the Open Networking Foundation in their SDN Architecture Technical Reference TR-502 [10]. It comprises a client controller layer that executes the modules that form the Network Application and a server SDN controller layer that drives the underlying infrastructure.

The challenge is to integrate client and server controllers. A first idea is to connect the South-bound Interface (SBI) for the client controllers to the North-bound Interface (NBI) of the server controllers. But as these interfaces do not normally match, adaptation is necessary. This adaptation has to cater for the idiosyncrasies of the different controller frameworks we intend to support and has to be implemented for each of them. To maximize reuse, we use separate adaptors for the SBI, called *Backend*, and for the NBI, called *Shim*. This separation imposes a protocol between them, the *NetIDE Intermediate Protocol* (cf. Sec. II-D).

In our first implementations [11] we have proved that the Shim/Backend structure connected by an intermediate protocol is feasible and sensible. However, they still left the fundamental component in these modules: the composition logic. This implied that it needed to be re-implemented for each controller we wanted to support. To overcome this shortcoming, we introduce an intermediate layer, the *Core* (cf. Sec. II-A): it hosts all logic and data structures that are independent of the particular controller frameworks and communicates with Shim and Backends using the same NetIDE Intermediate Protocol. The Core makes both Shim and Backend light-weight and easier to implement for new controllers.

In the rest of this section we describe the Network Engine starting with its three main components, namely Core, Shim and Backend. Then, we introduce what we called *fencing*, a run-to-completion scheduling mechanism used by the Core to perform the composition. Finally, we present the NetIDE Intermediate Protocol as a means for the Core to drive the execution flow of the whole Engine.

A. The Core

The Core is a platform-independent component that implements three main functions: (i) interfacing with Backends and Shim, controlling the lifecycle of controllers as well as modules in them, (ii) orchestrating the execution of individual modules or complete applications, possibly spread across multiple controllers, (iii) handling the symmetric messages exchanged between modules and network elements.

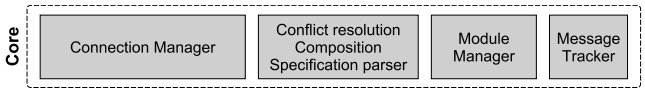

Fig. 2: Architecture of the Core.

Fig. 2 shows the building blocks of the Core. It leverages on the NetIDE Intermediate Protocol to interface with Backends and Shim and to keep track of the modules that compose the Network Application. The latter operation is accomplished by the *Module Manager*, which assigns unique identifiers to the application modules through a *module announcement-acknowledgement* process. These identifiers are used by the other components of the Core to link messages and modules.

The Core controls all messages exchanged between application modules and the network. In this sense, the operations of the Core can be divided into three categories: (i) handling the asynchronous events from the network, such as new flows, port status, flows removed, (ii) composing the configuration messages generated by the application modules and checking them for conflicts, and (iii) pairing *read-state* request messages and the corresponding replies.

Operations (i) and (ii) are executed by the *Composition/-Conflict Resolution* component based on predefined policies (cf. Sec. III).

With *read-state* messages we refer to requests issued by application modules to collect information from the network (e.g. `FEATURE_REQUEST`, `GET_CONFIG` in OpenFlow). The Core must ensure that the responses generated by the network for such requests are received by the right module, i.e. by the module that made the request. OpenFlow uses the Transaction Identifier (`XID`) to ease the pairing. However, in the context of the Network Engine, `XID`s are not sufficient, as different modules may use the same value, effectively making the reply/request pairing impossible. The *Message Tracker* avoids duplicated `XID` values by replacing the identifiers it finds in the requests with unique values.

B. Shim and Backend

The **Shim** is a platform-specific component that translates the NBI of the server controller to the NetIDE API, thereby exposing it to the other components of the Network Engine. Fig. 3 shows the internal architecture of the Shim integrated in a server controller platform. It includes hooks placed inside SBIs (small grey boxes in the figure) to route all messages from the network to the Shim itself, overriding the server controller's processing logic.

The *Core Connection* component handles the communication with the Core through the NetIDE Intermediate Protocol. The *API Translator* converts between NetIDE Intermediate Protocol and platform-specific NBI messages. The *Topology* module provides Network Engine's upper layers with access to network topology details and asynchronous updates in case

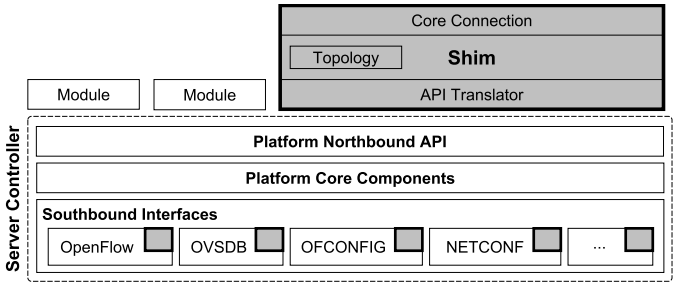

Fig. 3: Architecture of a Shim.

of changes in the network. Topology discovery is achieved by interacting with network through the server controller's SBI or by retrieving the topology information collected by specific core components available in advanced controller platforms (e.g. ONOS and OpenDaylight). Topology details include `datapath_id`s, number of supported flow tables, (administrative and operational) port information and other properties of the network elements.

The **Backend** is designed to be an additional southbound interface for the client controller that interacts with the underlying layers of the Network Engine. Fig. 4 shows the architecture of a Backend. At boot-time, the Backend starts the discovery of the application modules running on top of the client controller (*Module Discovery*) and registers them to the Core (*Announcement Handler*), which, in turn, assigns a specific identifier (`module_id`) for each registered module. As part of its initialization process, the Backend also queries the Shim for the physical topology. Backends store the topology information as instances of the network elements (*BackendDPs* in Fig. 4) which are exposed to the application modules.

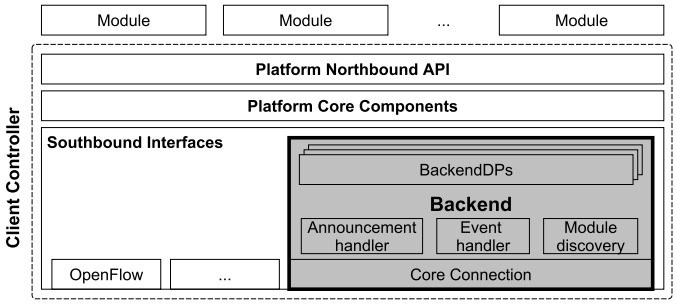

Fig. 4: Architecture of a Backend.

At runtime, Backends use the `module_id` in the NetIDE Intermediate Protocol header to tell the Core which module is sending the message. On the other side, the Core uses the `module_id` based on pre-defined policies to indicate which module handles the event. The *Event Handler* steers the event distribution inside the Backend, ensuring events are sent to the correct module.

Fig. 5 shows a detailed view of our architecture. We include in it the mechanism we use to integrate modules written for the server controller framework into our applications. Specifically, we place a Backend in the server controller to steer the message flow for the modules composing Network Application that run on the server controller, exactly like for any other module running on a client controller. In this case, the modules can only interact with the Backend, since the other SBIs are hidden to them by the Shim, as explained in Sec. II-B.

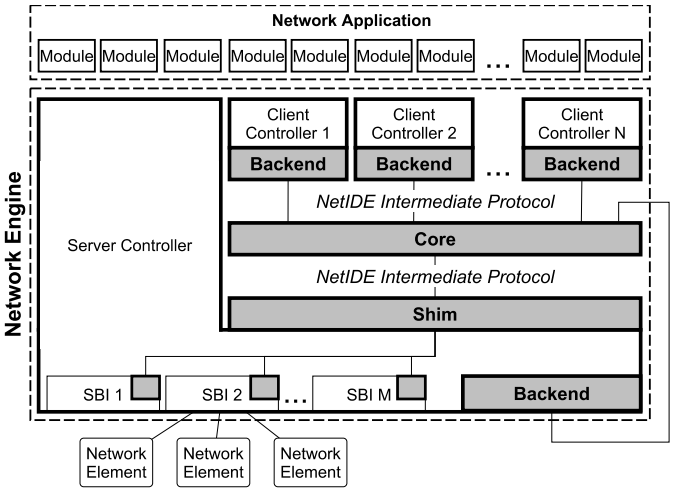

Fig. 5: Detailed Architecture of the Network Engine.

C. Fencing

The way different SDN frameworks, and particularly OpenFlow, work is that SDN modules receive network events and *optionally* produce network commands in response. This implies that modules may quit silently, without producing any tangible response. In order to implement application composition (as described in Sec. III), a prerequisite is to have a clear notion of when all modules involved in an application have finished processing a network event. Otherwise, there is a risk that the Core performs composition operations too early, i.e. when some modules are still processing the event, or that it freezes waiting for a response which will never arrive. We introduced *fences*, i.e. end-of-execution markers, to tackle this problem. We require that Backends monitor the execution flow within the client controllers.

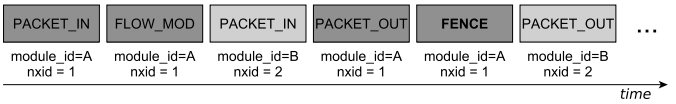

Fig. 6: *Fences* to delimit transactions.

As shown in Fig. 6, the network event, related network commands and the *fence* message, all use the same `module_id` and `nxid` values in the NetIDE header so that the Core can correlate them and create the notion of a *transaction*. It starts with an event (e.g. an OpenFlow `PACKET_IN` in the figure) sent by the Core to module *A* with `nxid=1`. The module replies with a combination of flow rules and specific forwarding actions (`PACKET_OUT`) or with nothing. The Core supports interleaved communication with the application modules to improve the performance of the Engine. In order to preserve the semantic of the network, the Core ensures that composed output goes back to the network consistently with the time ordering of events.

D. NetIDE Intermediate Protocol

The NetIDE Intermediate protocol implements the following functions: (i) to carry management messages between the Network Engine's layers (Core, Shim and Backend); e.g., to exchange information on the supported SBI protocols, to provide unique identifiers for application modules, implement the *fence* mechanism, (ii) to carry event and action messages

between Shim, Core, and Backend, properly demultiplexing such messages to the right module based on identifiers, and (iii) to encapsulate messages specific to a particular SBI protocol version (e.g., OpenFlow 1.X, NetConf, etc.) with proper information to recognize these messages as such.

The NetIDE Intermediate Protocol covers the communication between the different components of our architecture: (i) Module advertisement for the client controllers using the `NETIDE_MODULE_ANN`, `NETIDE_MODULE_ACK` messages (ii) Topology exchange between Shim and Backends using the `NETIDE_TOPOLOGY` message, (iii) Transaction control with the `NETIDE_FENCE` message, and (iv) `NETIDE_OPENFLOW`, `NETIDE_NETCONF` or others, to exchange SBI specific messages like OpenFlow packets, etc. The NetIDE Intermediate Protocol provides a 64-bit `datapath_id` field to uniquely identify network elements when SBI messages are carried. It also provides a 32-bit `module_id` to uniquely identify application modules running on top of each client controller. The composition mechanism in the Core leverages on this field to implement the execution flow of these modules. Finally, the `nxid` field is used in the context of the *fence* mechanism (cf. Sec. II-C).

III. Application composition

When composing an SDN application from pre-existing modules running on different client controllers, the challenge is to provide "chaining" semantics for client controllers that may not support it. We borrow the definitions of *Sequential* and *Parallel* operators from [12], [13] to introduce flexible forms of interaction between the components of a composed application (so-called *execution semantics*) and we provide additional functionality. We will use the processing pipeline shown in Fig. 7 to illustrate how our concepts are applied in a NetIDE application composed of several atomic functions or modules.

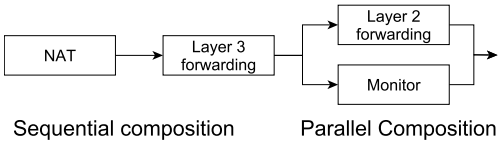

Fig. 7: An example for module composition.

A. Composition semantics

In order to provide a flexible and powerful way to reuse existing modules or applications to build new applications, the Core executes *composed applications* from the *composition specifications* provided in the application's *execution manifest*. The composition specification defines the modules that are used in the composition, the flow of execution between them, and *execution semantics* with *filters* and *merge policies*.

Execution semantics control both the order in which modules are invoked and their input. *Sequential execution* invokes modules in the order defined in the specification. The first invocation uses the original input (e.g. the original `PACKET_IN` event, ... in an OpenFlow environment); each subsequent call uses the original input modified with the actions returned so far (e.g. if the first module issued an action to rewrite the destination address, the next module will receive the packet with its destination set to the new one as input). *Parallel execution* invokes modules in parallel using the same input for all. Results are combined according to a *merge policy* when all invocations have finished.

Filters define matches on packet fields similar to those used by OpenFlow rules; they select a subset of network events. A module is only invoked if the input matches the filter (e.g., if the source address of the packet matches the specified subnet or if the invocation was triggered by a `PACKET_IN` event).

A *merge policy* is a function that maps a set of actions contained in a flow table modification message (e.g. an OpenFlow `FLOW_MOD`) to a resulting set of actions. It is used to determine how conflicting results are handled. Conflicts can be flexibly detected; options include checking when the same field of an action contains contradictory instructions, like `drop` and `forward` (see below for the actual resolution). Standard policies are *discard* (ignores any conflicting results by discarding them), *ignore* (installs conflicting rules without resolving the conflicts) and *priority* (resolves conflicts by picking the rule with the highest priority out of a set of conflicting rules). Other merge policies are easy to implement if needed, for example automatic rewriting of rules by priorities.

Listing 1 shows the composition specification file we would use for this application. The `Modules` element (line 2) lists all modules used in the application. This information prepare the Network Engine (i.e. start client controllers if necessary) and start the modules when the new application is loaded. The `ExecutionPolicy` element (line 17) defines the execution flow on network events.

B. Composition Execution

The Core intercepts the control traffic between the network elements and the client controllers. It receives network events from the Shim, encapsulated in the NetIDE Intermediate Protocol (as highlighted in Sec. II-D) and distributes them to the client controllers through the composition component.

Using the example above in Fig. 7, the execution flow is as follows:

1) A query is sent to the *NAT* module using the original packet as input. For the client application, the query behaves like a regular OpenFlow `PACKET_IN` event. The query returns with a list of actions. These are aggregated inside the composition component implemented in the Core.
2) A second query is sent to the *L3 forwarding* module, with the original packet, modified with the actions from the first query, as input (sequential execution semantic). The application might return an empty action set, which will be interpreted as an explicit drop (cf. the `emptyResultAction` attribute). Otherwise, the resulting actions are merged with prior results using the default merge policy (overwriting any conflicting actions with the newer ones).
3) A copy of the resulting query from the second module is sent to each of the modules defined inside the subtree `ParallelCall` (i.e. *L2 forwarding* and *Monitor*). This query uses the original packet, modified with the actions aggregated thus far, as input. The results of these two queries are then merged using the `priority` policy, i.e. non-conflicting actions are aggregated as usual and conflicts are resolved by taking only the action returned from the application with the highest `priority` (`L2 forwarding`, as indicated in Listing 1).
4) The merged actions are installed in the network elements through the Shim and server controller.

```
<ExecutionManifest>
  <Modules>
    <Module id="NAT">
      <Identifier>eu.netide.nat</Identifier>
    </Module>
    <Module id="L3">
      <Identifier>eu.netide.l3-forw</Identifier>
      <Filter>event=packet-in,source=131.23.4.*</Filter>
    </Module>
    <Module id="L2">
      <Identifier>eu.netide.l2-forw</Identifier>
    </Module>
    <Module id="Monitor">
      <Identifier>eu.netide.monitor</Identifier>
    </Module>
  </Modules>
  <ExecutionPolicy>
    <ModuleCall id="NAT"/>
    <ModuleCall id="L3" emptyResultAction="drop"/>
    <ParallelCall mergePolicy="priority">
      <ModuleCall id="L2" priority="2"/>
      <ModuleCall id="Monitor" priority="1"/>
    </ParallelCall>
  </ExecutionPolicy>
</ExecutionManifest>
```

Listing 1: Composition specification example.

IV. Implementation

We implemented the NetIDE Network Engine based on SDN controller platforms available as Free Open-Source Software (FOSS)[1]. However, we provide functional specifications [4] and platform-independent libraries [14] to facilitate the implementation of any Network Engine component by any vendor planning to adopt such an architecture in a closed-source commercial SDN environment.

We adopted ODL and ONOS as server controllers, Floodlight as client controller and Ryu as both. ODL and ONOS are carrier-grade controllers targeted to service providers, enterprises and mainstream deployments. Being able to scale to a large number of network elements, we recognized both ODL and ONOS as the most suitable SDN platforms for playing the server controller's role. On the other hand, Ryu and Floodlight are aimed at facilitating the rapid prototyping of SDN applications. We were able to isolate their SBI drivers to add our Backend alongside, and to hook into their event dispatching mechanism and drive the execution logic.

ODL and ONOS projects are based on Apache Karaf [15], so that we implemented the Shims for such platforms in form of OSGi bundles, leveraging their respective Java NBIs to interact with the network elements. Both implementations support OpenFlow versions 1.0 and 1.3 and make use of serialization methods to convert messages from the controller's NBI format into OpenFlow byte arrays before encapsulating them with the NetIDE header. Vice-versa, deserializers are used to convert byte arrays contained in the payload of NetIDE messages (OpenFlow commands) into NBI Java classes. While for ONOS we could leverage the implementation of serializer/deserializer methods provided by the *OpenFlowJ-Loxi* library [16], in ODL (which uses its own library *Openflow-Java* [17]) we had to implement the serializer from scratch and to extend the existing deserializer in order to cover all OpenFlow message types.

[1]Closed-source controllers might be considered as well, but are impractical to work within a research project.

The Shim for Ryu has been implemented as an application based on the Python northbound API. It includes serializer and deserializer methods based on the Python's *struct* module [18].

The Backends for Ryu and Floodlight have been implemented by duplicating and extending the existing OpenFlow SBIs. Following this strategy, we have been able to (i) easily understand how to distribute the events to (unmodified) application modules based on the Core's directives, i.e. overriding the default logic of the controller and (ii) intercept the exit point of event handlers executed by the application modules and implement the *fence* mechanism as described in Sec. II-C.

We implemented the Core as a platform-independent OSGi bundle for Apache Karaf to simplify the overall deployment procedure of the Network Engine. Recall that both ONOS and ODL are based on such a framework, therefore Core, server controller and Shim can be deployed in the same OSGi container. Message serialization and deserialization in the Core are based on the OpenFlowJ-Loxi library.

All aforementioned components are publicly available at [14] under the Eclipse Public License v1.0.

V. Evaluation

Re-usability of SDN applications has a cost in terms of latency introduced by the Network Engine on the control channel, affecting all the control messages exchanged between the Network Application and the network elements, and in particular the installation of new flow table entries. We identified two main sources of latency: (i) message serialization/deserialization and (ii) composition mechanism. In this section we evaluate the consequences of these operations on the data plane by measuring the Round Trip Time (RTT) of ICMP packets.

A. Methodology

Experimental Setup. The hardware used for the evaluation consists of a commodity PC equipped with a Intel i7-5600U quad-core CPU running at 2.60GHz and 16GB of DDR3 memory working at 1600Mhz. We run the Network Engine on Mininet [19] configured with a single-switch, OpenFlow 1.0 network and two virtual hosts attached to it.
For evaluation purposes, we force the emulated switch to send the packets to the Network Engine before forwarding them. This allows us to estimate how the Engine's operations affect the installation of flow entries onto the switch's flow tables.

Network Engine. We evaluate the Network Engine with modules running on client controllers only. This simplified configuration is sufficient to accurately measure the performance of all implemented components.

Metric. As anticipated above, we evaluate the efficiency of the Network Engine by measuring the RTT of ICMP packets exchanged between the two hosts in the emulated network. Since all the packets go through the Network Engine before being forwarded, the measured RTT values include twice the delay caused by the Network Engine's operations, once for each direction of the communication.

Please note that tests described in Sec. V-B and Sec. V-C are performed at 20 ICMP pkt/sec. Thus, 40 `PACKET_IN-FLOW_MOD` transactions per second, which corresponds to 40 new flows per second.

B. Serialization/Deserialization

Serialization/deserialization operations are performed by all the Network Engine's layers to convert control messages from controller's NBI format into byte arrays and vice-versa. If we disable composition, we count five serializations and five deserializations for each event-action transaction between a network element and the SDN application. Five times more than common controller frameworks like Ryu and Floodlight. Since the result of this evaluation depends on how such functions are implemented, we measure the latency with different combinations of client and server controller frameworks.

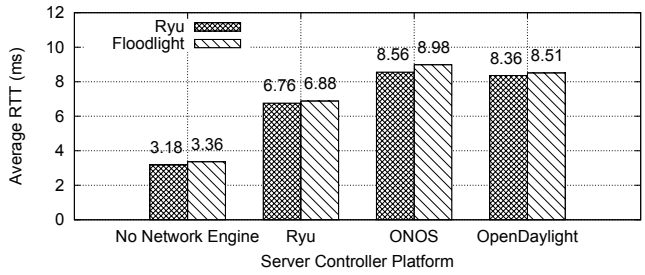

Fig. 8: RTT measured with different combinations of client-server controller frameworks.

Plot in Fig. 8 shows how the RTT of ICMP packets is affected by the serialization/deserialization operations performed on `PACKET_IN` and `FLOW_MOD` OpenFlow messages. Performance of stand-alone Ryu and Floodlight controllers are shown in Fig. 8 ("No Network Engine" cluster) as a reference. As expected, the RTT values are higher when the emulated network is controlled by the Network Engine. Nevertheless, in the worst case (Floodlight-ONOS combination) the additional delay for the installation of a new flow entry is 2.8ms on average, which is acceptable for operational networks.

C. Composition

We compare the performance of composed SDN applications with what we call "merged SDN applications". Thus, we run SDN modules on the Network Engine composed either in parallel or in sequence and we measure the RTT. Then, we merge the code of such SDN modules into a single SDN application and we run it on the Network Engine with the composition disabled. The difference in performance measures the cost of the composition.

Fig. 9 shows the CDF of the measured RTT for monitoring and L2 forwarding modules composed in parallel or merged into a single SDN application.

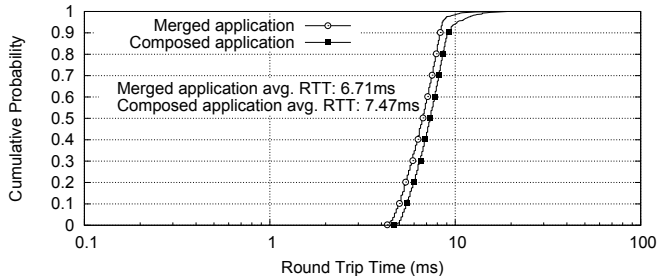

Fig. 9: Cumulative probability of the RTT for monitoring and L2 forwarding modules, either merged or composed in parallel.

We observe that parallel composition increases the RTT approximately by 700μs on average, i.e. by 350μs on average the one-way flow setup.

Fig. 10 shows the performance results of Network Address Translation (NAT) and L3 forwarding modules composed in sequence or merged into a single SDN application. In case of sequential composition, the one-way time increases approximately by 1.2ms. Please note that values obtained with the composed application include the time spent for the two sequential interactions between the Core and the two SDN modules to handle each ICMP packet.

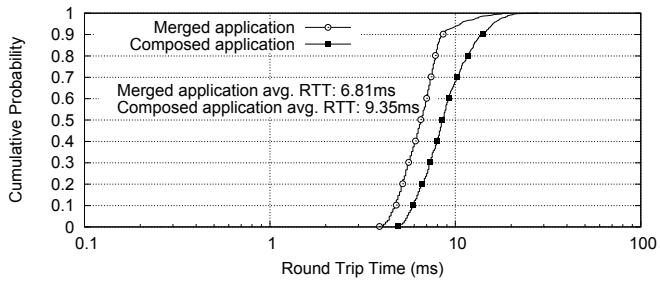

Fig. 10: Cumulative probability of the RTT for NAT and L3 forwarding modules, either merged or composed in sequence.

From results shown in Fig. 9 and 10, we can conclude that the impact of the composition on the new flow installation process is negligible. We expect it may increase with the number of modules, but it is a performance penalty that can be tolerated when considering the effort that would be needed to re-implement all modules for a different controller framework.

VI. Conclusion

In this paper, a novel SDN component called Network Engine has been proposed to perform rapid deployment of existing SDN applications belonging to different controller frameworks on operational SDN-based infrastructures. A preliminary version of such component has been implemented in a real prototype and several tests have ben performed in realistic scenarios to show how it may impact network operations. This allows us to conclude that composing a control application out of existing modules is feasible, even if the modules were written for different controller frameworks. Network administrators can not only *reuse* their existing applications when migrating between SDN platforms, but can chain heterogeneous modules to form more complex network control applications.

With the full NetIDE environment, which includes a full software development environment for composed network applications, we are providing network operators with the toolbox that will enable them to implement DevOps principles in their environments. This will shorten the time-to-market for services based on NetIDE Network Applications and reduce the cost of developing them.

Acknowledgments

The authors wish to thank Alexander Leckey, Antonio Marsico, Arne Schwabe and Giuseppe Petralia for the preliminary implementation of the Network Engine. The authors also thank Sergio Tamurejo and Raúl Álvarez Pinilla for their valuable support in the testing phase. The work presented in this paper has been partially sponsored by the European Union through the FP7 project NetIDE, grant agreement 619543.

REFERENCES

[1] "OpenDaylight - A Linux Foundation Collaborative Project," http://www.opendaylight.org.

[2] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, New York, NY, USA, 2014, pp. 1–6.

[3] "Ryu SDN framework," http://osrg.github.com/ryu/.

[4] "NetIDE Project," http://www.netide.eu.

[5] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving Research Slices out of Your Production Networks with OpenFlow," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 129–130, Jan. 2010.

[6] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make Your Virtual SDNs Programmable," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, pp. 25–30.

[7] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-defined Networks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 87–101.

[8] A. Dixit, K. Kogan, and P. Eugster, "Composing Heterogeneous SDN Controllers with Flowbricks," in *22nd IEEE International Conference on Network Protocols, ICNP 2014, Raleigh, NC, USA, October 21-24, 2014*, 2014, pp. 287–292.

[9] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, "Corybantic: Towards the modular composition of sdn control programs," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII, 2013, pp. 1:1–1:7.

[10] Open Networking Foundation, "TR-502: SDN architecture," https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf, 2014.

[11] R. Doriguzzi-Corin, E. Salvadori, P. A. Aranda Gutiérrez, C. Stritzke, A. Leckey, K. Phemius, E. Rojas, and C. Guerrero, "NetIDE: Removing Vendor Lock-in in SDN," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, April 2015, pp. 1–2.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11, 2011, pp. 279–291.

[13] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-defined Networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, 2013, pp. 1–14.

[14] "Network Engine source code," https://github.com/fp7-netide/Engine.

[15] "Apache Karaf," http://karaf.apache.org.

[16] "OpenFlowJ Loxi," https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi.

[17] "OpenDaylight OpenflowJava," https://github.com/opendaylight/openflowjava.

[18] "Python struct," https://docs.python.org/2/library/struct.html.

[19] "Mininet," http://mininet.org.