NetIDE: First steps towards an integrated development environment for portable network apps

Federico M. Facca*, Elio Salvadori*, Holger Karl†, Diego R. López‡,
Pedro Andrés Aranda Gutiérrez‡, Dejan Kostić§, and Roberto Riggio*
*CREATE-NET, Via Alla Cascata 56/D, 38122 Trento, Italy
Email: ffacca@create-net.org, esalvadori@create-net.org, rriggio@create-net.org
†Universität Paderborn, Warburger Straße 100, 33098 Paderborn, Germany
Email: hkarl@mail.uni-paderborn.de
‡Telefónica I+D, GCTO Unit, D. Ramón de la Cruz 82, 28006 Madrid, Spain
Email: diego@tid.es, pedroa.aranda@tid.es
§ IMDEA Networks, Avenida del Mar Mediterráneo, 22, 28918 Leganés, Madrid, Spain
Email: dkostic@imdea.org

***Abstract*—Nowadays, while most of the programmable network apparatus vendors support OpenFlow, a number of fragmented control plane solutions exist for proprietary Software–Defined Networks. Thus, network applications developers are forced to re-implement their solutions every time they encounter a new network controller. Moreover, different network developers adopt different solutions as control plane programming language (e.g. Frenetic, Procera), severely limiting code sharing and reuse. Despite having OpenFlow as candidate standard interface between the controller and the network infrastructure, interoperability between different controllers and network devices is hindered and closed ecosystems are emerging. In this paper we present the roadmap toward NetIDE, an integrated development environment which aims at supporting the whole development lifecycle of vendor–agnostic network applications.**

Keywords*—*software-defined networking; ide; network applications portability; network applications development cycle

I. Introduction

A. The need for better development support in SDN

Software-Defined Networking (SDN) has brought about a proliferation of network control plane solutions [1]. Various control planes support different APIs, often at different, still typically low, abstraction levels. For a developer of a customized network solution, it remains a challenge to develop directly using these APIs; consequently, there is a considerable momentum in the community for developing standalone tools to support SDN development, e.g., debuggers, profilers, simulators, or specialized editors. While these individual tools are indubitably helpful, their partial lack of maturity and missing integration entails high development and management costs for SDN programs. Also different SDN APIs relies on different tools, e.g., JunOS has is own IDE that cannot be used for programming networks based on Cisco ONE, thus limiting the portability of network programs. Combined, these issues threaten wider success of SDN as a whole.

The current state of affairs in SDN development is sketched in Figure 1 (we shall use this figure as a basis to later explain what changes are needed and intended). Currently, different SDN visions, each with their own merits, coexist and provide partial solutions to networking problems. However, there is no sensible integration of these solutions – there is some freedom in choice due to the introduction of standard control protocols like OpenFlow, yet no harmonized solution across the network gear/control logic gap. Thus, a network operator that deals with different SDN-like gear and controllers needs to develop a number of different solutions for all the various controllers (or even non-OpenFlow gear) it uses in operating its infrastructure[1]. This is even more evident in the case of cloud software developers that aim to closely integrate (for optimization purposes) their software with the underlying network infrastructure that may change from customer to customer. This technological diversity turns quickly into an implementation and deployment nightmare, fostering vendor-lock-in with all its known downsides.

We stipulate that there is considerable benefit in integrating, simplifying, and harmonizing the development process for SDN, e.g. code portability will be improved, development time will be reduced and network software management procedures

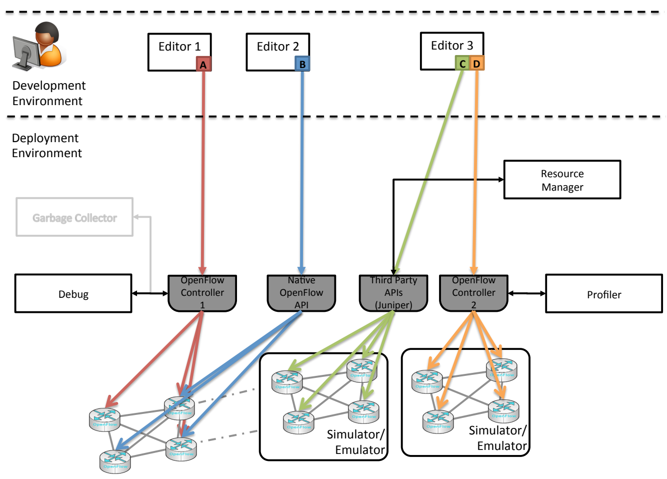


Fig. 1: State of the art in SDN software development.

[1]Unless the operator chooses a costly option to replace the entire gear or controllers with a single solution.

will be more automated. That integration should take place with respect to different tools (e.g., debuggers and profilers), but also with respect to the multitude of different control protocols (OpenFlow or proprietary), control platforms (open source or proprietary), and tools. In particular, access to these tools and platforms should be presented to a network developer in a fashion similar to what a software developer is used to. We claim that there is a need for a single Integrated Development Environment (IDE) that supports the whole software development cycle for Network Applications, running on top of a Software-Defined Network (Section I-C will make the notion of "Network Apps" more precise). Specifically, such an IDE should support the typical design, code & compile, test & evaluate cycle that still prevails in much of software development, as well as more advanced approaches (e.g., model-based design). For the design phase, such an IDE should support the use of high-level languages for SDN (e.g., Frenetic [2], Procera [3]). For the code & compile phase, different target platforms (e.g., different controllers) should be supported from the same source for the network application. For the test & evaluate phase, both evaluation tools like Mininet [4] (or other simulators) should be tightly integrated in the development process, as well as debugging and profiling tools for life networks.

Yet apart from the support and tight integration of a wide range of tools, an efficient development cycle is also characterised by an access to a wide range of standard services. In conventional software development, this is usually achieved by access to standard libraries. Extending our analogy from conventional software development to network applications development, we see a need for a rich set of standard, feature-rich, extensible system-level services that can be tightly integrated in a development process[2]. Examples for such run-time services include:

- *Flow table optimisation*: when a sufficiently high number of flows share the same path, it could be beneficial to use the same network state for all these flows. This can be done by replacing shorter-lived, per-flow switching entries in the switch with longer-lived entries that are valid for these flows.
- *Garbage collection*: in this scenario, there might be table entries that are only used by low data rate flows, which just consume resources in the switching table. A process is needed to identify such entries in the table and trigger a table clean-up, e.g., merging such entries and jointly rerouting low-priority flows on secondary routes.

We hence believe it is necessary explore the definition of new network-layer services that are independent of the underlying SDN flavour, and provide the support for emulator-in-the-loop and simulator-in-the-loop configuration.

B. An example: Network Programs for Big Data Services

Consider, as example, a Big Data solution developer who provides tools and services to optimize and control the network interconnecting its different Big Data nodes. Currently, he would develop specific software according to the appliances and network control plane solutions adopted by his customers. This requires the manual combination of different sets of tools for each SDN solution to cover the whole development cycle (e.g. different IDEs, different debug tools, etc.). The only reusable part of his code would be the APIs the service exposes to control the network.

With the introduction of NetIDE, this developer will use a single development environment to support the development of network control programs for different SDN frameworks. Moreover, suppose he needs to support an SDN framework that is not yet supported by NetIDE (e.g. Cisco ONE). In this case, he will only have to develop a new driver to integrate the new SDN framework into NetIDE (where this driver could be a product in its own right, rather than just a clumsy porting exercise like today). In this way, he will reduce the cost to develop tools and services to manage SDN-like networks for his Big Data solution. Moreover, he will be assured forward compatibility with new SDN solutions coming to the market.

C. Our contribution

In this paper we present "NetIDE", a concept for a one-stop solution for developing network control plane programs. The main ingredient is an IDE that *supports the whole development lifecycle for software-defined networks*, from the design of protocols and rules to testing and deploying them, and predicting or evaluating their performance. The approach will be independent from the specific programmable network apparatus vendor and it will be independent from the programming language used to write network control programs.

The simplification of network programming goes along with the concept of Network Applications or **Network Apps**. A Network App is a customized code that is able to dynamically control the behavior of a set of resources in the network. In the context of an SDN architecture, such Network Apps would control and parameterize the actual SDN controller. A Network App can be fairly close to an actual application (e.g., optimizing a network for a particular mixture of MapReduce jobs in a data center); it can also be more oriented to lower-level system tasks – in this latter case, we typically talk about Network Services as a special kind of Network Apps. Moreover, a Network App is not restricted to interacting with the SDN controller only. It can also offer interfaces via which other Network Apps can interact with it. In the end, this can turn a mere SDN into an *Application-Defined Network* where the application(s) on top of the network directly influence the re-programming of underlying networks through a dialogue with the control plane. In our vision, a network developer should be able to:

- Perform the whole network programming development lifecycle from a single IDE. The IDE should cover: requirement collection, design, network program coding, network program deployment, testing and debugging.
- Program any SDN-like network using his favorite network programming language, while sharing code with developers using other programming languages to maximise ease of development and reuse.
- Investigate and ascertain the likely performance of the resulting network for various load patterns by having performance analysis and debugging tools integrated in the development cycle.

[2]Strictly speaking, even a "system-level service" is an application running inside the network, yet it seems advisable to distinguish between more system-oriented and more user-level oriented network apps.

- Deploy the generated network programming code on top of different and coexisting SDN frameworks.

The remaining paper is structured as follow. In Sec. II we briefly discuss the related work. Section III presents the concepts that we will use to build NetIDE Interchange Representation Format. In Sec. IV we discuss a reference architecture for the implementation of NetIDE concept. Finally, Sec. V presents our plans to realize the NetIDE concept.

II. Background & State of the Art

A. *Raising the abstraction level in SDN*

High-level languages such as Frenetic [2], SNAC [5], The Flow Management Language [6], and Nettle [7] cover part or the entire set of control loop requirements. Frenetic includes a network state querying language together with a policy definition language, and a methodology to instantiate rules in an OpenFlow network. It also introduces the concept of composition that allows different application to operate on an isolated slice of the network concerning both routing and monitoring aspects. An evolution of the Frenetic language called Pyretic [8] introduces the concept of sequential composition, as opposed to the parallel composition exposed by Frenetic. Sequential composition allows different applications to process the same packet. SNAC and the Flow Management languages focus more on the definition of Access Control Lists using a custom pattern-matching based language. Nettle provides an abstract, although low-level language for programming OpenFlow switches, but lacks any querying language and an intermediate runtime system capable of composing different network application. Procera [3] is a high level reactive network programming language which allows the network developer to implement policies that can adapt to changes in the underlying network. For example, Procera can be used to implement user authentication, or policies that need to react to the time of the day or the amount of traffic generated by the user. Procera can in principle run on top of any OpenFlow controller and even act as a policy layer for solutions such as Frenetic.

SDN allows multiple applications to control the same network, therefore, security and correct applications isolation is very important. Along the security and access control research direction we find FlowVisor [9] and FortNOX [10]. Both approaches provide a system for enforcing security and authorization constraints. However, while FortNOX is built on top of the NOX controller, FlowVisor acts as proxy between the controller and the actual network and provides a controller-agnostic solution that slices the network between a range of packet subspaces and provides isolation between them. Each application can access and operate only on its own slice.

OpenDaylight [11] is a recent initiative by major companies to contribute to a new unified SDN stack. The participants plan to provide a new controller platform. At the low, network-facing, level it is capable of communicating with the switches supporting different protocols (e.g., OpenFlow). At the high, application-facing, level it provides a REST API to communicate with user-provided applications.

B. *Tools*

To evaluate or predict performance of an SDN setup, only a limited set of tools exist. First, there are common discrete event simulators (e.g., NS/3, OMNeT++), but they lack specific features that would allow their simple application to the SDN world. SDN-specific tools like Mininet [4], OFLOPS [12] exist, yet some suffer from limited scalability. E.g., Mininet, as shown by benchmarks, did not yet achieve the original goal of "1000 nodes in a laptop", or in simpler words of simulating a real wide network on commodity hardware [13].

There exists a large number of tools that allow SDN testing before deployment, and debugging it afterwards. First, there are solutions that model the network and allow controller testing. AntEater [14] and Header Space Analysis (HSA) [15] can identify typical connectivity errors in a given configuration by statically analyzing the dataplane configuration, i.e. the forwarding tables. NICE [16] provides a systematic approach to testing of multiple possible event orderings by taking into account existence of race conditions in a network that is inherently a distributed, asynchronous system. SOFT [17] tests the interoperability of OpenFlow switches.

When the network is in operation there are other tools that allow simple monitoring and troubleshooting. Examples include ATPG [18], NDB [19], OFRewind [20] (with automated filtering [21].) to record both control channel as well as selected portion of data packets. It then allows administrators to replay selected parts of the recorded trace in order to locate a sequence of messages and events that lead to an error. Further, there are tools that allow quick evaluation of basic network forwarding properties when facing quickly changing forwarding rules, e.g. NetPlumber [22] and VeriFlow [23].

This richness in tools is both an advantage and a challenge for a typical developer. We aim to harness this richness by integrating and structuring it.

III. NetIDE Interchange Representation Format

NetIDE will define a mechanism to abstract SDN programming independently from the underlying SDN flavor. The abstraction mechanism will range from networking frameworks based on open standards like OpenFlow to closed programmable networking frameworks relying on specific APIs provided by the vendor (e.g. Junos XML API by Juniper Networks). This approach allows us to define Network Apps independently of the actual network gear and controller technology. This assertion is valid as well for *generic* Network Apps such as Resource Manager and Garbage collectors that can be used as well as *services* by other Network Apps.

At the heart of this approach lies the NetIDE **Interchange Representation Format (IRF)**, a *lingua franca* (i.e., a unifying language) that covers orthogonal aspects of deployment models of different SDN approaches (e.g. floodless vs flood-based) and that is executable and translatable across different SDN flavors (e.g. OpenFlow versus Junos). The NetIDE IRF can be deployed on the actual SDN substrate in the same way as OpenFlow or vendor–specific applications are deployed today. At the same time, it provides a common representation of the network that allows our tools to be SDN platform independent. Thus, different components of the IDE, such as the debugger, can be designed independently from the underlying SDN, since they will be manipulating objects in the IRF. Initially, the IRF will consist of two essential parts:

- Topology information. Topology information will capture the structure and parameters of different network layers and their relationships. For instance, physical network topology will form a basis for logical network topology, and Layer 3 topology will be defined on top of it. Topology information has to be extensible so as to enable defining special layers in support of particular use cases.
- Policy descriptions (e.g. routing, security and monitoring). Policy description will be based on the concept of functional reactive programming (FRP). This way, the network will be able to react both to changes in quasi-continuous signals, such as aggregate traffic volumes, and to sequences of discrete events. The policy rules may be formulated for a single network switch or its interface, or expressed in terms of the entire network. It will be the task of NetIDE software to translate network-wide rules into a coordinated set of fine-grained rules and instructions for individual network elements, using also the topology information. IRF will also provide combinator operators that will allow for combining simple rules into more complex ones.

From a software-engineering perspective, we will take a Model Driven Architecture (MDA)-like approach. We will define Network Apps using a platform-independent model (PIM) using an appropriate domain-specific language (DSL). IRF as matter of fact will play the role of DSL that we will use to describe SDN solution-independent Network Apps that will act as PIM. Then, at deployment time the PIM is translated to one or more platform-specific models (PSMs) that network controllers can run. In this case the PSMs will be the set of instructions that can be understood by OpenFlow controllers or specific vendors network controllers. Moreover, in line with the principle of recursively applying MDA transformation patterns across different abstraction layers, we will allow the developers to use different DSL (e.g., Frenetic) to build PIMs that are then transformed to IRF.

Figure 2 shows the transformation flow (abstracting from MDA) we will adopt in NetIDE. Network programs described through different languages (e.g. Frenetic or PI-Calculus) can be transformed to IRF and vice-versa allowing developers to select their favorite network programming language. Network programs described through IRF can be executed on top of different controllers thanks to a set of drivers that convert IRF to controller-specific instructions (e.g., Floodlight or Trema).

The tools developed by the NetIDE will generate and process IRF content. On the input side, the tools will translate selected network programming languages into IRF, and the main task on the output side will be to transform IRF rules into a coordinated set of instructions for individual network elements, using the topology information as well as other technical details, such as sampling rates of quasi-continuous signals.

IV. NetIDE Architecture

The transformation flow described above will be supported through a reference architecture implementing the development

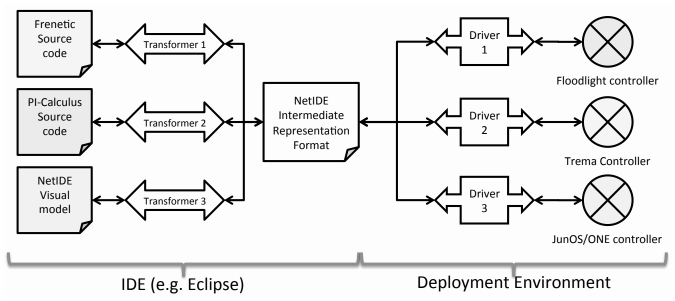
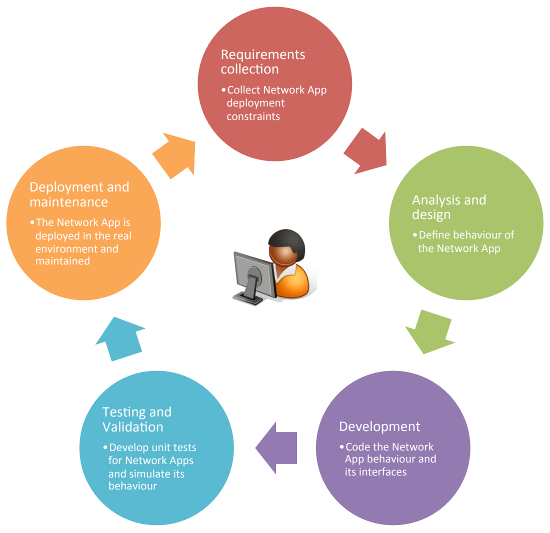


Fig. 2: NetIDE network abstraction flow



Requirements collection
•Collect Network App deployment constraints
Analysis and design
•Define behaviour of the Network App
Development
•Code the Network App behaviour and its interfaces
Testing and Validation
•Develop unit tests for Network Apps and simulate its behaviour
Deployment and maintenance
•The Network App is deployed in the real environment and maintained

Fig. 3: A reference development cycle for Network Apps

cycle for Network Apps sketched in Fig. 3. The main activities in such cycle are:

- **Requirements collection**: in this phase the developer defines the scope of the Network App: it may be a service application, e.g., the garbage collector that cleans up unused flows; it may be an interface towards other applications (e.g. Cloud Management tool) to facilitate the dynamic reconfiguration of the network; or a simple network control plane application like a load balancer allocating traffic flows in the network in order to minimize traffic congestion events. In this phase he can collect information on the topology (or topology pattern) and other aspects relevant to the design of the Network App.
- **Analysis and Design**: following the results of the requirements collection phase, the developer provides a specification of the actual behavior of the Network App and models relevant aspects that will constitute both variables and constraints of the Network App. For example, he can formalize (or retrieve it from the controllers) the topology as a initial model on top of which he defines the traffic flows or other network programming aspects.
- **Development**: the developer actually codes the network program using his favorite network programming language, translates it to the IRF, and develops the APIs that the Network App will expose to third parties for

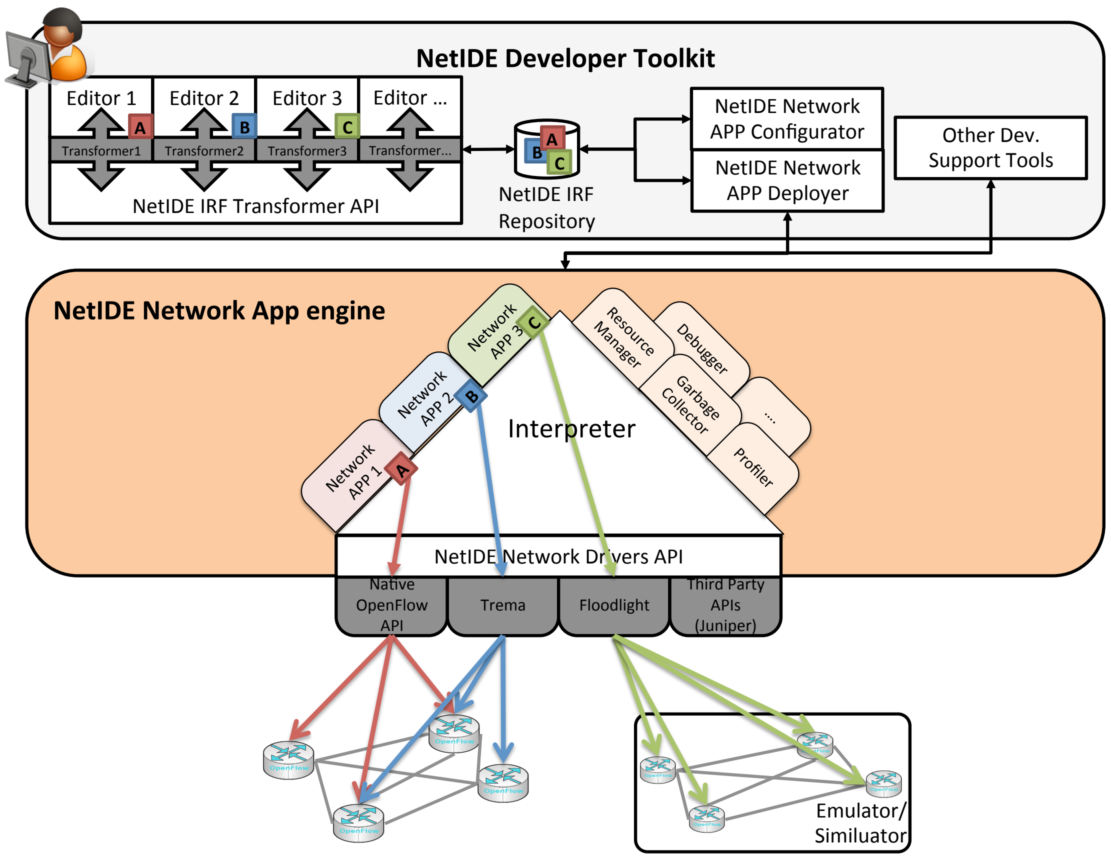


Fig. 4: NetIDE Reference Architecture

interacting with its functionalities.

- **Testing and Validation**: testing is an integral and important phase of the software development process. This step strives to ensure that defects are recognized as soon as possible. With support of Unit tests and simulators, the developer will be able to experiment in advance with the behavior of the Network Application (or of its parts) before deployment in the production environment. In this way, he will be able (for example) to simulate the traffic flow in the network and check whether the rules he defined for the Network App bring the desired effect. Also, he will be able to test passing parameters from third applications and see if they influence correctly the Network App behavior. Finally, he will be able to use systematic exploration tools, such as model checkers, to subject the Network App and the network it controls to a large number of possible event orderings.
- **Deployment and maintenance**: the deployment phase starts after the Network App is appropriately tested. At this point in time the developer can configure different deployment environments (if not occurred in the design phase) and test the installation of the application. In this phase testing and interaction as well with simulator is still possible. If the Network App needs refinement, the development cycle restarts with a new iteration.

These steps are supported by the NetIDE reference architecture (see Fig. 4) which includes two main components:

- **NetIDE Developer Toolkit**: a set of integrated tools in an Eclipse-like environment that allows the developers to code, configure, and deploy Network Apps. The Developer Toolkit will include: editors for network programming languages like Frenetic, an IRF Transformer API that supports the development of transformers from programming languages to IRF, a number of tools that support the configuration and deployment of Network Apps embedding IRF programs, and other support tools.
- **NetIDE Network App Engine**: a runtime environment that hosts Network Apps and acts as virtual controller of the network, leveraging existing network controllers. The Network App Engine will include: a container for Network Apps deployed and managed through the Developer Toolkit; an interpreter acting as a virtual controller and execute IRF logic over the real and simulated networks; a NetIDE Network Driver API that allows the development of different drivers to connect the interpreter to different OpenFlow controllers (e.g. Floodlight) and SDN frameworks (e.g. Junos); a number of services developed on top of the interpreter to facilitate the management and debugging of network resources and Network Apps independently from the underlying SDN-like technology.

With such an IDE, the developer uses his preferred editor (and underlying language) to write the network control plane logic for his Network App. The control plan logic is neutrally represented using the NetIDE IRF through a language–specific transformer. The developer then uses the NetIDE Network App

Configurator to wrap the control plane logic in an App container and defines how to deploy it. As result, the control logic wrapped in a Network App is now ready to be deployed in the NetIDE Network App Engine. The deployed control logic is then executed by the interpreter that issues instructions. The instructions, through a specific network driver, are passed to a given controller or network device, e.g. using the OpenFlow API in case of OpenFlow devices. During the execution of the control logic, heterogeneous network resources are managed to ensure consistency across the network. System services like the garbage collector ensure that resources are freed if no more in use; profiler and debugger help monitoring the control logic behavior, while the resource manager provides feedback about network resources usage to the Network App (or the virtual controller) in pursuit of higher performance.

V. Conclusion & Future Works

Despite the big hype behind Software-Defined Networking, most of the proposed frameworks suffer from the limitations of being hardly interoperable. This issue severely limits the huge potential of innovation SDN may unveil. One of the most critical limitations of current solutions is the need for network applications developers to re–code from scratch their solutions if they want to have them running on a different controller platform. The adoption of vendor-agnostic software-development frameworks supporting the whole development lifecycle for SDN is therefore highly desirable.

In this paper we presented the concepts behind NetIDE, an integrated development environment for portable network apps, which aims at supporting network program developers by reducing the development time and complexity through a single, integrated solution that covers requirements, design, coding, deployment, testing and debugging of SDN applications. NetIDE aims at becoming the one–stop solution for developing network control plane programs which are independent from programmable network apparatus vendors as well as from the programming language used to develop network programs.

In next future – January 2014 – the concepts presented in this paper will be the starting point for a FP7 research project – called as well NetIDE. Through the project we will explore the feasibility of our concepts and focus on the core research of defining IRF and on building the tools to support our vision. The IRF, the NetIDE Developer toolkit and the NetIDE Network App engine will be as well validated into three industrial scenarios. This will allow us to drive the research focusing on actual requirements and constraints posed by solutions used nowadays by telco and datacenter operators.

Acknowledgment

The authors would like to thank Joe Butler (Intel), Ladislav Lhotka (CZ.NIC Labs), Gerard Nguengang (Thales), Bernhard Schräder (Fujitsu) and Maciej Kuzniar (EPFL) for their valuable feedbacks and inputs about the NetIDE concept.

References

[1] H. Chao and B. Liu, *High Performance Switches and Routers*. Wiley, 2007. [Online]. Available: http://books.google.it/books?id=dsw1faspYiAC

[2] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison, "Languages for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 128–134, 2013.

[3] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proc. of ACM HotSDN*, 2012.

[4] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. of ACM CoNEXT*, 2012.

[5] B. Pfaff, B. Forbes, C. Anderson, D. Wendlandt, E. Lazutkin, J. Pettit, K. Amidon, M. Casado, M. Kobayashi, N. Gude, P. Balland, R. Price, S. Seetharaman, T. Koponen, and T. Rice.

[6] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. of ACM WREN*, 2009.

[7] A. Voellmy and P. Hudak, "Nettle: taking the sting out of programming network routers," in *Proc. of ACM PADL*, 2011.

[8] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. of USENIX NSDI*, 2013.

[9] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the Production Network Be the Testbed?" in *Proc. of USENIX OSDI*, 2010.

[10] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proc. of ACM HotSDN*, 2012.

[11] The Linux Foundation. OpenDaylight Project. [Online]. Available: http://opendaylight.org/

[12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: an open framework for openflow switch evaluation," in *Proceedings of the 13th international conference on Passive and Active Measurement*, ser. PAM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 85–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_9

[13] V. Yazici. (2013) Benchmarking mininet. [Online]. Available: http://vlkan.com/blog/post/2013/04/19/benchmarking-mininet/

[14] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. of ACM SIGCOMM*, 2011.

[15] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *Proc. of USENIX NSDI*, 2012.

[16] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proc. of USENIX NSDI*, 2012.

[17] M. Kuźniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostić, "A SOFT Way for OpenFlow Switch Interoperability Testing," in *Proc. of ACM CoNEXT*, 2012.

[18] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic Test Packet Generation," in *Proc. of ACM CoNEXT*, 2012.

[19] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proc. of ACM HotSDN*, 2012.

[20] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *Proc. of USENIX ATC*, 2011.

[21] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker, "How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software," UC Berkeley, Tech. Rep., 2013.

[22] P. Kazemian, M. Change, H. Zheng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *Proc. of USENIX NSDI*, 2013.

[23] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *Proc. of USENIX NSDI*, 2013.