On Network Application Representation and Controller Independence in SDN

Pedro A. Aranda Gutiérrez
Telefónica, Investigación y Desarrollo, S.A.U
Technology Exploration
pedroa.aranda@telefonica.com

Holger Karl
University Paderborn
holger.karl@upb.de

Elisa Rojas
Telcaria Ideas S.L.
elisa.rojas@telcaria.com

Alec Leckey
Intel Labs Europe
alexander.j.leckey@intel.com

List of Corrections

***Abstract*—While Software Defined Networking is starting to fulfill some of its promises, the danger of vendor lock-in still lurks: Network control applications are closely tied to a particular controller framework and not easily portable between them; multiple controller frameworks are usually not able to cooperate inside a single network. This paper describes architectural options to enable portability and cooperation inside a single network, recommending a master/client multi-controller approach at runtime, with some additional information about network control applications generated at development time.**

I. Introduction

Software-Defined Networking (SDN) is starting to show benefits. Lower equipment cost and regaining the control of the network are value propositions which Network Operators cannot ignore. However, we observe a high level of scattering in the market, since equipment vendors are adopting different controller frameworks with different programming environments and Southbound Interfaces (SBIs). This trend puts Network Operators *back to square one* with regards to vendor lock-in: once an SDN vendor is chosen, it will bring in its network gear, SDN controller and set of Network Applications (i.e. the applications that implement a certain network functionality and run in one SDN controller). Any attempt to create a multi-vendor network will, at the very best, result in the creation of multiple isolated network islands, each with the solution of a specific vendor. Moreover, reusing Network Applications is difficult or impossible, given the differences in programming language and model among SDN controllers.

There is hence a need for a solution that allows multiple Network Applications to cooperate inside a single network, irrespective of the controller for which they were developed. This paper discusses architectural options how this goal can be achieved and recommends on option that has shown promise in early prototyping. We shall structure this discussion along the typical steps in a software development and deployment cycle (Section III), after clarifying some terminology conventions in Section II. We shall consider architectural options for the controllers as such, as well as for the generation and usage of information about Network Applications. After these considerations, we will put our proposals into context with related approaches in Section IV before we conclude the paper.

II. Context: SDN architectural assumptions and terminology

Standard SDN terminology starts to coalesce [1]. Nonetheless, we want to highlight here one architectural aspect not commonly considered in existing terminology. In particular, it concerns how functionality is structured inside an SDN controller.

Commonly, an SDN system is regarded as a set of ***Network Elements*** (e.g., switch, router), sending network events using a control protocol (e.g., OpenFlow) to a controller process that interprets these events and computes suitable actions in response to them. This controller process is often regarded as a monolithic piece of software, and in many SDN architecture discussions this is fully sufficient and adequate. For the purpose of the following discussion, we want to highlight the following observations: Usually, the actually running controller is not a monolithic piece of software. Rather, it is composed of some basic functionality as well as some customized code; this customized code takes the actual decisions and realizes the specific behavior of a particular network.

This basic functionality is reusable across many different deployments. It realizes functions like parsing messages from switches, providing execution environments for the customized code (calling its functions in a proper environment, with means to store state, set timers, etc.), and send actions back to switches. This basic functionality can be realized in different ways, and examples for such different realizations exist: Projects like Beacon, POX, Ryu, Floodlight and others provide such basic functionality.

In isolation, basic functionality alone will not provide any network behavior – it is but an empty shell. Hence, most of the projects just mentioned also provide examples for custom functionality (e.g., topology discovery, spanning trees, etc.); but this is software that is qualitatively different from the basic functionality. The main point is that such custom functions must be easily extensible and replaceable. Only together – basic functionality and custom functions – is an actual SDN controller complete and ready to work.

It seems hence prudent to assign separate names to these different aspects of a controller. The reusable, basic, non-specific functionality of a controller constitutes a ***Controller Framework***; the specific, customized code that controls the processing of messages and determines the actual actions could be referred to as ***Network Control Applications***. Together,

once instantiated and running in a real system, they act as a ***Controller***, illustrated by the dashed box in Figure 1. [1]

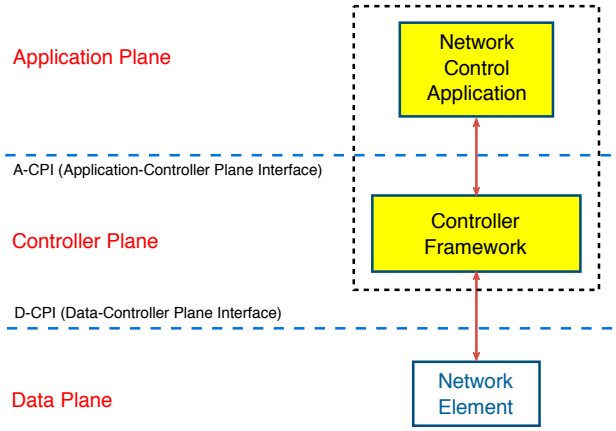

Fig. 1. ONF proposed SDN solution architecture

We largely keep to the standard terminology recommended by the Architecture & Framework working group of the Open Networking Foundation (ONF), shown in Figure 1, which provides a foundation for SDN solution development. We slightly simplify the term Network Control Applications to ***Network Applications***. These applications reside in the ***Application Plane***. The SDN Controller Frameworks reside in the ***Controller Plane***. Finally, all Network Elements are considered to be part of the ***Data Plane***.

III. Architectural options

The problem to solve is hence to run Network Applications developed for different controller frameworks inside a single network. Ideally, for a Network Application, this should be transparent: no changes are required. Rather, changes in the controller platform are acceptable; options are discussed in Section III-A. These considerations will conclude that additional information about a Network Application is helpful or required beyond the mere object code; options for suitable *Network Application representations* are presented in Section III-B. Finally, Section III-C briefly summarizes requirements to use said representations.

A. Architectural options for controllers

To realize such a multi-framework-supporting controller, several architectural options are conceivable.

- **One input language** Conceptually, one could replace all existing programming models by a single language to specify Network Applications, in the sense of an external Domain Specific Language.
  This is the goal of the Northbound Interfaces Working Group (NBI) in the Open Networking Foundation (ONF), which attempts to develop an information model and encodings for SDN Controller Northbound Application programming interfaces (APIs) in a programming-language-neutral manner, as a data model. For example, this approach is taken by the Merlin project [2], specifying a new external Domain Specific Language (DSL) for this purpose. If such an approach were to generally take hold, a lot of the complications ensuing from the multitude of different frameworks would disappear. However, we do not foresee this to become the domineering approach to design SDN Network Applications, nor would this solve the issue of supporting legacy Network Applications developed for already existing framework.
- **One meta controller framework** Create a new controller framework that is a superset, in features, programming models, and programming languages, of all the relevant controller frameworks in use/of interest. Instantiating the controller from this framework and a diverse set of Network Applications would then be a simple exercise. However, in practice, this is a futile endeavor: chasing a moving target as new controller platforms come and go; huge design and implementation complexity.
- **Translating into one "universal" framework** Choose one target framework and translate foreign Network Applications into this target framework; at run-time, execute only this target framework.
  This is indeed a tempting approach as it would fulfill all goals and the run-time benefits are huge. And if the different controller frameworks are simple enough in terms of programming model and execution semantics, and if the programming models are explicit enough to extract the structure from a given source code, then this is indeed a possible venue. In fact, we initially pursued this goal (in the context of the NetIDE project), but found the semantic and executional differences between different controller frameworks to be too large to easily reconcile.
  In addition, this approach is also burdened with other difficulties: Which target platform for choose? How to make it easy to develop the translation process for a new, upcoming framework? In total, this seems like a less promising alternative.
- **Run multiple controllers concurrently and independently** It would be trivial to instantiate one separate controller for each framework specified by any Network Application that an administrator wants to execute. But this falls short of the mark: Which controller would be responsible for which events? How to reconcile conflicting actions? How to enable Network Applications to exchange events and actions, to access a shared knowledge base, and to jointly take decisions on which actions should be triggered? All that is required to mimik the notion of "single controller", but none of that is easy to do with independent controllers.
- **Run multiple controllers, but coordinate them** The previous approach solved the issue of the execution semantics per controller framework. Amending it with additional coordination then seems to solve our problem. When running separate controllers per framework (each one hosting possibly multiple Network Applications), their access to the actual Network

[1] We conjecture that the common, yet slightly confusing practice to talk about a "Beacon controller", a "Ryu controller", etc. comes from the fact that most controller frameworks actually do come with sufficient default Network Applications to constitute a working controller, eg, L2 Learning Switch.

Elements is shunted through one single controller. The interaction between these multiple controllers can be coordinated by an exchange protocol. This appears to be a viable and pragmatically feasible option, which we will pursue in the remainder of the paper.

This leads to the following design: The Controller Plane will consist of multiple SDN Controller Frameworks, arranged into two layers, as shown in Figure 2. The ***Server Controller Framework*** is defined as the entity that is the primary controller physically connected and managing the network elements. The ***Client Controller Framework*** allows the Network Application to be executed on their targeted Controller Framework, which is abstracted from the actual Network Elements through the Server Controller Framework. [2] As a coordination protocol, our prototype currently uses Pyretic's [3] approach.

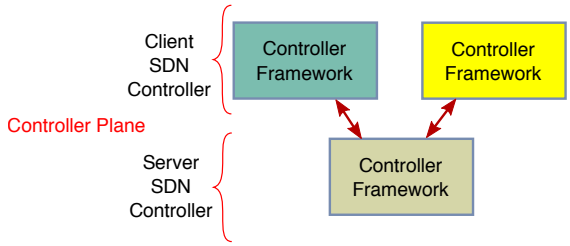

Fig. 2. Concept of Client/Server Controller Frameworks

We note that these multiple controllers serve a different purpose than hierarchical controller concepts as proposed by, e.g., Kandoo [4]. There, the goal is scalability and performance and heterogeneity of controllers is (to the best of our knowledge) not addressed. We hence believe these concepts to be orthogonal; hierarchical scaling of our architecture is left for future study.

B. Architectural options for Network Application representations

With such an architecture in mind, let us consider the principal options when, in the sense of a software development cycle, information about Network Applications can be created, to later assist in the execution of these applications. In the following, we consider the options for the *generation* of Network Applications in different phases of the software development cycle, whereas in the next Section III-C, we consider the options for *usage* in the same phases.

1) Generation at development time: When generating information about a Network Application at development time, the following information is available:

- The **source code** of the application, as this is the primary artefact with which the developer works.
- While developing, the developer also makes assumptions about the execution (or concurrency) model, the libraries, APIs, etc. that are at his/her disposal, provided by the particular controller framework in use. Moreover, most controller frameworks are strongly tied to a particular programming language. Hence, the programming language might also be fixed.
- A more or less explicit information about the topology or topologies to which this application is applicable – e.g., a load-dependent routing algorithm might only work in a Clos network but be inapplicable to fat trees. However, it is usually the case that a developer has a range of topologies in mind or that a topology is parameterized. In that sense, we assume that a **topology pattern** is available (and will provide editor support for such patterns in our tool development). Of course, a Network Application might make no specific assumptions about the network topology in which it works. To support such a case, an empty pattern is valid as well.

2) Generation at compile time: All the information from development time is available at compile time as well. Additionally, compilers (with a liberal interpretation of what constitutes a "compiler") generate **object code** from the source code. Here we assume standard language processing. Examples include generating executable files from compiled languages like C or producing intermediate representations like JAR files.

In principle, during this step, it might also be possible to analyse the source code and extract structure from it like, for example, which event handlers exist, to which events they react, which code they execute in response to which event handlers, what state information they keep and how they update it, and which actions they produce, etc. As discussed above, this task would be simple, perhaps even trivial, assuming a DSL in place to describe the Network Application.

However, when analyzing actual frameworks and their programming models, as well as typical example Network Applications we did not see a realistic subset of expressability that would enable such an approach. Obviously it would be possible to stipulate required structures, but that results in a considerable amount of limitation in the programming models of the different frameworks. Very little of the existing programs would conform to such a structure, and it seems highly questionable whether it would be possible to establish such a structure for existing controller frameworks – in a sense, it combines the disadvantages of a DSL (lack of flexibility) with the disadvantages of dealing with different controller platforms.

It therefore seems not very promising to expect too much from a compile-time approach to generate any valuable information that should assist in the interaction of Network Applications written for different controller frameworks. Rather, at compile time, it makes sense produce a representation of a Network Application that is still tied to its "original" controller framework, but has sufficient meta data in place to make it deployable in combination with Network Applications written for other frameworks.

3) Generation at deployment time: At deployment time, we are given a Network Application as well as a concrete system in which to deploy it. *Deploying* here means to take

[2] Although the Openflow 1.0 specification and above supports the concept of multiple Server SDN Controller Frameworks managing the Network Elements for reliability and failover (using the roles Master/Equal/Slave), this is orthogonal to the purpose of our architecture.

this application's code, start it and have it react to networking events by issuing actions. We conceive of this as a typical system administration step, triggered by an explicit administration option[3]. To do so, a few checks are necessary; these checks will produce additional information representing the *running* Network Application:

1) First, we need to check whether the Network Application's network pattern matches with the actual network of the system onto which we want to deploy it. It the simplest case, the application's pattern is empty and hence always matches. Also simple is a case where the application's pattern only matches the entire system's network; more generally, matching is simple when there is only a single way to do it. It is also simple if the pattern cannot be matched at all against the system's network: then, the Network Application cannot be deployed (and a corresponding error message is produced).
   If, on the other hand, matching the application's pattern against the actual network can be possible in several different ways – for example, consider a multi-level fat-tree network onto which a fat-tree routing algorithm should be deployed: because of the tree's recursive nature, multiple different embeddings are possible. In that case, we have to ask the administrator to select between different alternatives, either by specifying them beforehand as mapping/embedding parameters, or by entering into a dialog as a result of error messages.
2) Second, during deployment, it is necessary to check whether any open dependencies exist. Dependencies can be mainly of two types:
   - Dependency on a particular controller framework: when it is not assumed that an arbitrary controller framework can execute any arbitrary Network Application, Network Applications are still bound to their "original" framework. Hence, to execute them, the specific framework must be locally available. During deployment, this dependency must be fulfilled before a Network Application can be started.
   - A Network Application could depend on other Network Applications. These must also be available, in order to be able to execute the Network Application.

   More generally, dependency processing in the sense of bigger frameworks (e.g., Open Services Gateway Initiative (OSGi)) can be considered here as well.
3) As a third step before a Network Application is started at deployment, the framework it is written for must also start the controller (e.g., start a Beacon controller instance from the Beacon controller framework). We designate, for the sake of simplicity, this *running instance* as simply a *controller*; where necessary, we use *controller framework* to designate the collection of code, libraries, APIs etc. that realize this controller.
   This step is not always needed since another Network Application written for that framework might already be running. In general, the same controller can then be reused.

In summary, at deployment time, mostly checking activities will take place, as well as bookkeeping. This will generate information that describes the interplay of Network Applications, and can hence be regarded as a part of an intermediate representation. However, it will indeed only become available at deployment time since at development or compile time, the very combination of controllers and applications to handle is unknown.

4) Generation at run time: Once deployed in the network engine and paired with their corresponding controllers, Network Applications will start to react to events, collect state, and trigger actions. Events and actions are transported by the controllers, acting as a conduit from/to the controlled switches (or other network entities).

In a controller-agnostic scenario, some adaptation functionality between the client and server controllers is needed. This functionality, which intercepts platform specific actions and translates them to a generalised, platform agnostic representation and viceversa, can be isolated as a specific layer in each controller. In an ideal case, these layers are stateless and accumulate no data. However, we need to plan for stateful interception layers . The information generated in them would constitute the *dynamic part of an intermediate representation format*.

C. Requirements for Using Representations of Network Applications

Once we have pondered the different options and requirements for generating a hypothetical Intermediate Representation Format (IRF) that would hold our Network applications, we also need to see how it would be used in the different phases of the Software Life-Cycle as applied to Network applications and what additional requirements the mere usage of this IRF would impose.

1) Development time: At development time, access to metadata and to source code is necessary. Meta-data can be easily encompassed in a standard file description format (e.g., XML). This results in a shippable artefact for one Network Application.

2) Compile time: At compile time, access to metadata is partially necessary to use the correct compiler and to link against the right libraries of a controller framework. The outcome of the compile process will add object code to the artefact describing a Network Application. In interpreted languages, the compilation as such does not exist and we are considering the possibility of including the source code of the network application directly into the artefact describing the Network Application.

3) Deployment time: At deployment time, metadata describing topology templates are transformed by a network matching/embedding process from a pattern into a concrete topology. The topology just generated replaces the topology template in the artefact describing the Network Application. This is then taken and stored by the network engine.

[3]Automatic deployment is conceivable as long as there are no ambiguities to resolve.

4) Run time: At run time, additional state information is collected by the interception layers discussed previously. We foresee this information to be only dynamically updated; it is of course possible to generate a snapshot of the information of a running collection of Network Applications in there as well (which might be relevant for debugging, e.g.).

IV. Related work

Our main goal is to provide a framework that allows us to reuse Network Applications instead of having to rewrite them when changing the controller platform. There are several proposals for composing Network Applications in Software Defined Networks.

OpenVirtex [5] focuses at implementing multi-tenancy in OpenFlow (OF) networks. It provides isolation between different controllers that run tenant's Network Applications over a shared infrastructure. CoVisor [6] provides a hypervisor-alike approach and implement the two-layer controller approach that allow the composition of Network Applications running on multiple controllers. They have an initial implementation [7] derived from OpenVirtex. For their tests they use the Floodlight controller [8]. They provide a range of composition operators we are evaluating.

Flowbricks [9] provide an alternative architecture to CoVisor. They rely on the implementation of pipelines for flow tables, which go beyond the current capabilities of chained flow tables in OpenFlow 1.4. Their implementation relies on a modified version of OpenVSwitch. It can thus not be deployed on standard SDN equipment and its future heavily depends on the (improbable event of the) OF specifications following this concept.

Corybantic [10] is a system that allows to deal with conflicts between coexisting Network Applications. It defines an API to allow controllers to propose how network resources should be used and define how proposals of competing controllers should be evaluated. All the proposals are collected and evaluated by a Coordination Module to produce a global configuration. The process requires an interaction between the modules and the Coordinator, who finally mandates what the Network Applications have to do. The main drawbacks we see in this approach is that Network Applications have to be designed to use the proposed API and thus depending on Corybantic, while we target a Controller-agnostic approach.

V. Conclusion

In this paper, we have discussed the architectural options for a controller structure and for a corresponding representation of Network Applications that enable the reuse and coordinated deployment of Network Applications from different controller frameworks in a single network. We see an architecture evolving that consists of different aspects at development, compile, deploy, and run time.

Regarding the representation of Network Applications, we foresee the need to handle both static and dynamic information simultaneously. Some of the information about Network Applications is static; it describes aspects like source and object code as well as metadata (which controller framework, dependencies, etc.). Some of the information only starts to exists at deployment or run time, and we hence conceive of it as a dynamic part of a representation format.

We are currently in the process of prototyping such a two-layered controller architecture along with the necessary tooling and metadata structure. Early results leverage the API and protocol provided by Pyretic to implement the interaction between these two controller layers and already allow us to execute Pyretic applications on top of an OpenDaylight [11] or Ryu [12] controller. This platform will allow us to test different approaches and select those that best fit our needs. For more information, please visit our project sites [13], [14].

Acknowledgment

The work in this paper has been partially sponsored by the European Union under the umbrella of the FP7 Framework, under project NetIDE grant agreement 619543.

References

[1] O. N. Foundation, "Sdn architecture 1.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf, June 2014.

[2] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. D. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, A. Seneviratne, C. Diot, J. Kurose, A. Chaintreau, and L. Rizzo, Eds. ACM, 2014, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674989

[3] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *USENIX ;login*, vol. 38, no. 5, pp. 128–134, Oct. 2013.

[4] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342446

[5] "OpenVirteX," http://www.openvirtex.net/.

[6] "CoVisor: A Compositional Hypervisor for Software-Defined Networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jin

[7] J. Gossels, "CoVisor," http://covisor.cs.princeton.edu, dec 2014, note.

[8] "Floodlight Is An Open SDN Controller," http://floodlight.openflowhub.org/, 2011, last visited: Sun, 18 Nov 2012.

[9] 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. [Online]. Available: http://dx.doi.org/10.1109/ICNP.2014.50

[10] 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 *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, D. Levine, S. Katti, and D. Oran, Eds. ACM, 2013, p. 1. [Online]. Available: http://doi.acm.org/10.1145/2535771.2535795

[11] J. Medved, A. Tkacik, R. Varga, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, June 2014, pp. 1–6.

[12] "Ryu SDN Framework," https://osrg.github.io/ryu/.

[13] "NetIDE: An integrated development environment for portable network applications," http://www.netide.eu/.

[14] "FP7-NetIDE GitHub repository," https://github.com/fp7-netide.