Simplification of Internet Ossification through Software Defined Network Approach

Gaurav Kulkarni

Abstract- Software-Defined Networking (SDN) has received a great deal of attention from both academia and industry in recent years. Studies on SDN have brought a number of interesting technical discussions on network architecture design, along with scientific contributions. Researchers, network operators, and vendors are trying to establish new standards and provide guidelines for proper implementation and deployment of such novel approach. It is clear that many of these research efforts have been made in the southbound of the SDN architecture, while the northbound interface still needs improvements. By focusing in the SDN northbound, this paper surveys the body of knowledge and discusses the challenges for developing SDN software. We investigate the existing solutions and identify trends and challenges on programming for SDN environments. We also discuss future developments on techniques, specifications, and methodologies for programmable networks, with the orthogonal view from the Software Engineering discipline.

Keywords: software defined networking, SDN programming languages, software engineering.

I. Introduction

The Internet architecture has become complex and hard to manage. Due to its large development and level of maturity, implementing strategies with a high degree of innovation is risky because the success of the Internet depends on the accurate operation of all of its subnets. The Internet became static and difficult to change its structure, a phenomenon known as *Internet Ossification* [1]. The need for making networks more dynamic, robust, and able to be experimented with new ideas and protocols in realistic scenarios brought a new paradigm called Software-Defined Networking (SDN). SDN enables a new network architecture that makes possible for computer networks to be programmable [2]. In its essence, SDN decouples the control plane from the forwarding plane. It enables researchers and software developers to create and deploy network applications, by abstracting the underlying infrastructure and even complex protocols present in traditional and legacy networks. Programmable networks have been the subject of active research in the past (e.g., Open Signaling [7], Active Networking [8], and Ethane [9]). However, they failed to be fully adopted by the industry due to many reasons, such as focusing on the data plane programmability as well as enabling programmability for specific network devices vendors. Although some of the SDN concepts are not new, it integrates the concepts of programmability in the network architecture in order to offer better network management strategies. In this scenario, Open Flow [2] has been considered the *de facto* and widely accepted solution to implement SDN. It is worth emphasizing that Open Flow and SDN terms cannot be used interchangeably.

Open Flow is a protocol that defines an open standard interface for SDN, and uses a programmable controller to communicate with the forwarding plane, manage the network, and possibly receive instructions from a network application. Such an interface has a low-level implementation, which offers basic features to developers. The complexity involved in developing advanced SDN software applications needs to be addressed by other means (e.g., via new programming languages), in order to increase its level of abstraction. In this scenario, full development and deployment of such applications in staging and production environments remains a challenge for network operators [10].

Although some previous studies [11] [12] [13] [14] have surveyed the state-of-the-art on SDN programmability, we take a different perspective on the topic by describing the techniques, methodologies, and challenges to develop and deploy SDN software applications. We provide a unique view from the perspective of the Software Engineering discipline in which we present the evolution, current maturity, and point out prospective research directions and challenges to develop applications for SDN.

II. Software Defined Networking

The separation of the control plane from the forwarding plane is one of the pillars of the SDN paradigm. Its decoupled architecture enables network programmability. Historically, the research community made several attempts to provide network programmability, where Active Networking (AN) and Open Signaling (Opening) are considered the seminal approaches [7].

a) SDN Architecture

When the control logic is decoupled from the forwarding devices, all the network intelligence (e.g., decisions about routing, permissions) is moved to the controller. The SDN controller becomes the network component responsible for network management, as

Author: Assistant Professor, Institute of Technology and Management Universe, Vadodara Gujarat, India. e-mail: kulkarnigaurav@yahoo.com

Figure 1 depicts. Management then occurs through a flow table present in the network switches, which receive and register network rules defined by the controller (cf. section II. C). In other words, the SDN controller adds flow table entries in the switches for proper packet or flow handling. The controller has all the necessary network information (e.g., where the hosts are connected, topology, and the like) that it uses to deal with possible conflicts involving policies or to avoid misbehaviour of network elements. As Figure 1 depicts, the controller has two main interfaces, namely i) the northbound interface, for higher-level elements to support the development of network applications and services, or to program the SDN controller through a well-defined API and ii) the southbound interface, for the communication between controllers and network switches.

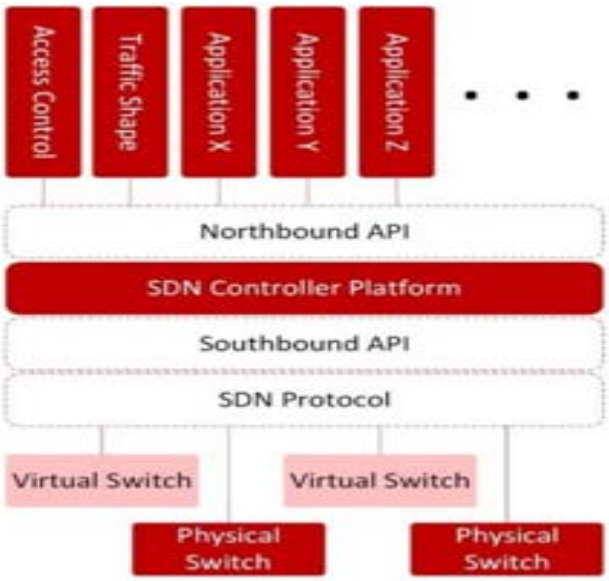

Figure 1: Northbound and Southbound Interfaces in an SDN Architecture

b) Controllers in the SDN Architecture

The SDN controllers are strategic control elements that communicate with the underlying switches (via SI) and with applications on the top (via NI). An SDN controller sends messages to switches disseminating specific or general packet handling rules, which are generally defined by a developer or administrator through the controller's northbound API [13] [14].

c) The open flow protocol

The Open Flow protocol defines how the exchange of information between control-plane and data-plane must occur .When an Open Flow switch receives a packet, its header fields are verified and compared to related fields in the flow table entries. If an entry corresponds to this packet header, the switch will perform the set of instructions or actions related with the flow entry.

III. Programming Paradigms, Languages Specification, and Software Engineering in SDN

The paradigm for programming languages applications development is the *declarative*, used in most research papers in the literature [04] [10] [14]. *Declarative programming languages* have been characterized by its extremely formal nature, often based on logic, but without arithmetic [42]. This paradigm allows a developer to define *what* action needs to be done in the network, but not *how* this action will do it. Please note that this definition applies to all declarative programming languages. To make it possible, a language interpreter is used to translate the "what" into "how". An example involving this approach in an SDN scenario is shown below, using the Frenetic notation [10]:

```
Select(packets) *
        GroupBy([srcmac]) *
        SplitWhen([inport]) *
        Limit(1)
```

Figure 2: Frenetic declaration to filter packets

The example presented in Figure 3 demonstrates a high-level declaration to filter packets in a given flow, which does not require the programmer's knowledge to implement how the Select(packets) clause will receive and direct the packets to some program or service that is requesting it.

Another widely used paradigm present in SDN programming languages is the *Functional Reactive Programming* (FRP). FRP is a well-suited solution for the development of event-driven applications, such as SDN applications, enabling programs to capture the *time flow* property pertinent to SDN systems [13].The reactive characteristic of FRP is direct related to the SDN environment, where switches and controllers continuously exchange information upon packet arrival and apply rules to the corresponding flow. When an SDN language follows the FRP paradigm, it automatically administers the time flow and the dependencies between data and computation.

The main idea behind FRP is to define everything in terms of *signals*. A signal is an element in which its values change in the course of time [14] (e.g., if a variable switch is equal to false, its value might changes to true due to emission of a signal). Figure 4 depicts a code example in the context of FRP.

```
def ip_monitor():
return(Select(counts)*Where(inport_fp(1))*
GroupBy([srcip]) * Every(INTERVAL))
```

Figure 3: FRP characteristic of Frenetic

IV. Analysis of use Cases and Applications for SDN Programming Languages

Prospective environments for SDN scenarios drive us to analyze a number of specific applications and use cases for SDN programmability. All the languages analyzed in this survey have use cases and evaluation scenarios in their respective publications. This section then presents an overview of the SDN programming languages and their possible applications to be developed. Initially we describe and categorize the applications in the use cases previously defined in this survey. Then, we map these applications and use cases to the SDN programming languages that may be used by developers to write them, as shown in Table 3. This mapping defines the lessons learned in this survey, providing directions on what language to use in developing SDN applications.

Admission Control: An admission control application enables the administrator to specify the authentication rules for hosts and users that try to access the network. Admission control applications can be implemented through an SDN programming language to define what default connectivity is allowed and which authentication mechanisms will be used.

Load Balancing: The load balancing use case might be seen as a congestion-aware routing for networks [76]. With a load balancing application, the controller prevents overload instructing the switches how to balance the incoming traffic among the network paths.

Quality of Service (QoS): For QoS applications, developers may use how resources should be allocated to different users and flow classes. This is done by setting some network properties, such as latency and available bandwidth. These applications to fit in the Applications-based Network use case. This is because end-user software can communicate with the SDN controller, which must be running a QoS application, to request some network resource.

NAT Administration: The Network Address Translation (NAT) Administration is generally used to enable multiple machines within a private IP range to share a single public IP address, mapping two pools of IP addresses. This translation requires an implementation which alters the IP and port number of each packet in the private network. This is the basic difference between NAT and others applications mentioned. In NAT administration, each packet in the flow must be modified, therefore requiring the network switches to support this functionality. In the SDN scenario, the NAT administration application may be executed on the controller, which installs rules into switches to perform the modification of headers of certain packets corresponding to IP addresses and port numbers that should have a specific quality [11].

Security Rules: A typical example of security rules is the implementation of an IP addresses black list module that prevents a malicious IP source addresses from sending traffic.

Fault Tolerance: An interesting use case involves network resilience scenarios. For instance, in the case of a link failure, the network should be able to choose a backup path dynamically.

Deep Packet Inspection: It is a network application which examines packet's payload looking for patterns, such as from well-known applications and services, viruses, attacks, and the like. In SDN, the controller executes some algorithm to perform DPI. SDN languages as Frenetic [10] and NetCore [13] have features to implement DPI applications.

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

Cloud Orchestrator: The Cloud Orchestration use case needs a software orchestrator in order to manage the network and the virtual machines. All SDN languages partially enable the implementation of such a software, because they only provide methods to implement a network application, which in this case may create the network orchestrator. The orchestrator of virtual machine needs to be developed with third parties programming languages or obtained from vendors.

Policy Specification: The most basic feature of an SDN application and environment is the specification of policies. All the analyzed SDN programming languages enable the implementation of policies in several ways, as well as applications to define the network behavior through policies. However, they differ in the way of writing and implementing these policies in practice.

Network Monitor: Foster *et al.* [16] argue that querying network state is one of the fundamental elements in programming SDNs. A Network Monitor application in SDN can observe and request several types of information (e.g., packet counter state in a switch). All languages analyzed allow the implementation of applications that monitor network states.

Correctness: The verification and validation of network applications are desired features [14] [15]. SDN programming languages might offer constructs that help developers to avoid network misbehavior (i.e., verification), and to build correct applications (i.e., validation), according to the specified requirements.

V. Future Work

How to handle network failures? A recurrent discussion on SDN research involves handling of failures. Failures can occur in the availability of a controller or even in wrong policy rules defined by an SDN application. The authors of FatTire argue that programmers do not write programs using the primitive fast-failover OpenFlow mechanisms directly due to the increment of complexity in failure-handling control, which might make code more complex. In order to handle failures in SDN programming, the language needs to support an abstraction of the OpenFlow forwarding table called a *group table*. Group table consists of group entries. The ability for a flow entry to point a group enables OpenFlow to represent more methods of forwarding[16]. It enables multiple conditional rules in OpenFlow. One of the group table types is the *fast failover* (FF). The fast failover determine that if a flow entry belongs to this group type, the first *action bucket* (an ordered list of actions) will be performed.

FatTire [14] abstracts the construction of a fast failover group table, generating the entries in such group table automatically. This approach avoids the error-prone development made by programmers when interacting with fast failover group table directly [14].

From the *Software Engineering* perspective, the development of fault-tolerant applications must be based on languages that define dependable features or build rules created from formal methods. For instance, a language that provides modular development may enable an SDN application to run as redundant modules in replicated controllers, thus improving the recovering time of a network failure. However, synchronizing such modules is not a trivial task [13].

How to avoid conflicting rules? This is a challenge investigated by some research studies (e.g., PANE [80], Pyretic [16]). Avoiding conflicts means that a policy rule X does not invalidate a policy rule Y, and vice-versa, simultaneously, so that at least one policy rule should be correctly applied. In [16], Hinrichs *et al.* proposed two conflict resolution mechanisms, which we consider a valuable path to effective SDN programming, i.e. one has its features at the level of keywords, identifying the conflicting policies. The other mechanism is a schema that defines priority to each keyword (e.g. the keyword *deny* has precedence over the keyword *allow*). A similar approach can be also found in [15]. One possible approach to address conflicts in policies could be based on a DSML. In such an approach, invalid policies that result in conflicts could not be created due to the constraints contained in an underlying *metamodel*.

How can one realize automated tests? In order to identify inconsistencies or unexpected states in an SDN application, Canini *et al.* [12] and Vissichio *et al.* [12] propose approaches to realize tests in SDN applications. End-host applications and switches affect the program running on the controller. In [10] Canini *et al.* address this challenge by generating flows with several possible events occurring in parallel. It also enables the programmer to verify generic correctness properties (e.g., forwarding loops or black holes) and code validation (i.e., global system state determined by code fragments). On the other hand, in [82] Vissichio *et al.* use *Test-Driven Development* (TDD) to perform tests on SDN applications.

How to abstract the complexity in SDN development efficiently? The low level of abstraction used by OpenFlow and its releases makes it hard to program applications and to define a desired behavior into the network. The studies analyzed suggest that a decomposition of the controller, through one relationship with the OpenFlow protocol and adding a layer to specify policies, reduces the complexity to develop and deploy SDN applications, mainly due to the readiness to build applications without the need to worry about maintaining consistency of various rules present in an SDN environment. Therefore, such an abstraction is more than only adding more layers for SDN architecture or controllers; it also provides smart structures that reduce the complexity in SDN applications development, and not just encapsulating the methods from the underlying structures. Furthermore, this layering and efficient structures can be used by some DSML, further increasing the level of abstraction, enabling the concrete visualization of network behavior.

Be reactive or proactive? The proactive or reactive behavior and structure of a certain SDN language will depend closely on the controller and how packet handling occurs. It is worth emphasizing that one could follow a hybrid approach, where a combination of both strategies allows the flexibility from reactive paradigm to particular sets of traffic control, while proactively providing low latency routing for ordinary traffic. Creating a framework or SDN language to support these two main approaches seems to be the most correct way to achieve completeness. As far as we are concerned to create an SDN language, the possibility of defining a DSML enables developers to develop high-quality SDN applications. This isdue to the ability of DSML to raise the level of abstraction in software programming, because its visual representations are easier to understand than the syntax of textual programming languages.

How to improve the SDN programmability? Although this question allows a number of answers, we aim at presenting and discussing the four most important issues that need improvements: i) verifying and validating applications (e.g., consistent updates, rules, and the like), which could be achieved by using DSMLs or constraint checkers in compilers; ii) offering

high-level tools for developers, since there is no widespread tool (e.g., Integrated Development Environment – IDE, CASE tool) for creating SDN applications; iii) providing programming languages independent from the underlying controllers or southbound protocols, which fortunately there are some efforts in this direction, such as P4; and iv) writing applications that meet network dependable requirements.

VI. Conclusion

Some current challenges show that the programming of SDN applications is still complex and not completely standardize. Although there are several abstractions at application level for SDN there are still some issues to be addressed such as interoperability, fault handling, conflict resolution or detection. SDN offers the opportunity of innovative and powerful networking scenarios, the development of correct application with efficiency and efficacy is still work in the progress. In particular advance study MDD/DSML is a possible research path in order to achive correctness, completeness and ease of use and productivity.

References Références Referencias

1. R. C. Gronback, Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, Addison-Wesley Professional, 2009.
2. T. Özgür, "Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling In the context of the Model Driven Development," School of Engineering. Ronneby, Sweden: Blekinge Institute of Technology, p. 56, 2007.
3. T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell and S. Shenker, "Practical Declarative Network Management," WREN, 21 August 2009.
4. A. Voellmy, A. Agarwal and P. Hudak, "Nettle: Functional Reactive Programming for OpenFlow Networks," PADL, July 2011.
5. A. Monsanto, N. Foster, R. Harrison and D. Walker, "A Compiler and Run-time System for Network Programming Languages," POPL, 25-27 January 2012.
6. A. Monsanto, J. Reich, N. Foster, J. Rexford and D. Walker, "Composing Software-Defined Networks," NSDI, 2013.
7. N. P. Katta, J. Rexford and D. Walker, "Logic Programming for Software-Defined Networks," ACM SIGPLAN Workshop on Cross-Model Language Design and Implementation, 2012.
8. T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, "Network virtualization in multi-tenant datacenters," 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pp. 203-216, April 2014.
9. T. Nelson, A. D. Ferguson, M. J. Scheer and S. Krishnamurthi, "Tierless Programming and Reasoning for Software-Defined Networks," Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, 2-4 April 2014.
10. N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. K. Praveen, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story and D. Walker, "Languages for Software-Defined Networks," IEEE Communication Magazine, February 2013.
11. T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," USENIX OSDI, pp. 351-364, Octuber 2010.
12. M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," Journal Software—Practice & Experience , pp. 1253-1292, Oct 2009.
13. A. Voellmy, H. Kim and N. Feamster, "Procera: a language for high-level reactive network control," HotSDN '12 Proceedings of the first workshop on Hot topics in software defined networks, pp. 43-48, 2012.
14. D. C. Schmidt, "Model-Driven Engineering," IEEE Computer, 39(2) February 2006.
15. A. Van Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," Sigplan Notices 35.6, pp. 26-36, 2000.
16. F. Case, "Computer-aided software engineering (CASE): technology for improving software development productivity," ACM SIGMIS Database. Volume 17 Issue 1, pp. 35-43, 1985.
17. D. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, New York, NY, USA: John Wiley & Sons, Inc., 2002.
18. T. Stahl, M. Voelter and K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.

This page is intentionally left blank