Software Design Principles to Enhance SDN Architecture

Iyad Alazzam
Department of Computer Information Systems
Yarmouk University, Irbid, Jordan

Izzat Alsmadi
Department of Computer Science,
University of New Haven, West Haven, CT, USA

Khalid M.O Nahar
Department of Computer Science
Yarmouk University,
Irbid, Jordan

***Abstract*—SDN as a network architecture emerged on top of existing technologies and knowledge. Through defining the controller as a software program, SDN made a strong connection between networking and software engineering. Traditionally, network programs were vendor specific and embedded in hardware switches and routers. SDN focuses on isolation between control and forwarding or data planes. However, in the complete SDN network, there are many other areas (i.e. CPU, memory, hardware, bandwidth and software). In this paper, we propose extending SDN architecture and propose isolation layers with the goal of improving the overall network design. Such flexible architecture can support future evolution and changes without the need to significantly change original components or modules.**

Keywords—component; SDN; OpenFlo; Software design; SDN architecture; Design principles; Design patterns

I. INTRODUCTION

Software Defined Networking (SDN) has recently evolved as an alternative flexible network architecture to traditional network systems. The flexibility that software programs offer over the hardware is one major SDN feature.

Traditional network switches and routers that route data from and to local networks include two in-cohesive functional components; data plane that includes information about traffic, and how to deal with it and control plane for control and management functionalities. SDN proposes to decouple those two in-cohesive planes and include in new switches the data plane only. Control plane is moved to a separate software-based module. A new protocol, OpenFlow is designed to handle communication between the newly separated modules or planes. As the main protocol used in SDN, OpenFlow is used in many research and technical documentations as a synonym to SDN. We will also follow this acronym in this paper and use the two terms interchangeably.

Through such separation, developers and network administrators can have now full control over their networks. Routing algorithms that were closed and vendor specific are not any more. Applications can be developed on top of the controller to communicate and interact with the controller. Those applications, also called middle-boxes can be provided, through the controller, with customized flow-based information.

As a new architecture or technology, SDN comes with both challenges and opportunities. In this paper we will focus on the rise of software roles in SDN in comparison with traditional network. In software engineering, software design principles and patterns proposed how we can design software products that are easy to use, reuse, update and maintain. Modularity is a core software design concept related to developing a program with different software modules. Those modules should be highly cohesive from the inside (i.e. the inner components of modules) and at the same time coupling between the different modules should be minimized. Interfaces are software components that exist to support modularity goals. Different modules should interact with each other only through well-defined interfaces.

Many network appliances such as: Firewalls, IDS, traffic optimizers, load balancers, etc. are going to be developed to communicate with SDN controller. Without proper well-defined interfaces between each one of those applications and the controller, intruders can easily have back doors through those applications to access security sensitive controller resources. Using well-defined interfaces, controller and its modules can then provide very specific services to those applications. In addition, those specific services should be called after fulfilling several pre-conditions from the application side including pre-conditions related to self-identity and authorization proof. Currently, those applications can be defined and can expose controller modules or resources directly.

Specification and design should be separated from implementation which shows one way to fulfill those specifications. While in those SDN goals implementation was referring to physical configuration in switches, we believe that this can also be applied to code concrete implementation that should be isolated from network high level configuration that can exist in policies. Currently, many software controllers include mixed functionalities between core controller modules, quality assurance, monitoring, management, policies, or security modules, etc.

We will focus here on software design principles. Abstraction is one of the software design concepts that is heavily investigated in the software engineering field. Abstraction is about focusing on relevant information and ignoring irrelevant information suitable or relevant to the problem domain or to the level of the current system details. In software construction and implementation, abstract classes or interfaces are always proposed at the top of a library or a hierarchy to make the structure more stable and able to cope with changes or modifications. This is to acknowledge that software programs and their requirements are very volatile and change rapidly. Consequently a good software design should allow and accommodate such changes without the need to restructure the software or the system.

SDN architecture facilitates tasks' delegation. In traditional networks, network administration cannot be delegated and it has to be controlled from network devices. Three SDN features together (centrality, ability to monitor all network components from one location and programmability) can make it possible for network administrators to delegate different administration tasks to different users without losing the ability to manage or monitor them.

In this paper, we will revisit SDN architecture based on software design principles and patterns and show how such architecture can be improved.We believe that SDN separation of control and data plane is not enough. We will focus on abstraction layers in software parts.

The rest of the paper is organized as the following: In section two we will introduce several research papers that are relevant to the paper subject. In section three we will present goals and approaches for an enhanced SDN architecture. Paper is then concluded with a summary section.

II. Literature Review

Software engineering can bring several advantages to SDN. For example, Software engineering has mature knowledge, tools and experience in software design, development and testing. Brining those to SDN can be a very important beneficial joint venture.

Modularity is about developing software or system components that can be easily used, reused, modified, updated or maintained. It is very important to build an SDN architecture that allows developers to easily add new applications or middle-boxes without the need to significantly cause a system configuration/reconfiguration.Monsanto et al 2013acknowledge that current OpenFlow architecture has limited support for creating modular applications[1].

Reitblatt et al 2012 [2] discussed one problem related to SDN configuration updates. This is since current networks continuously change and evolve. Therefore it is very important for a good SDN design to cope with those frequent changes. Authors presented mechanisms to handle packet or flow level updates' consistency checking. In other words, a mechanism should exist to check whether a recent flow or packet update is consistent with the network state and its flow rules in flow tables. Authors proposed an abstract interface to offer solid mechanisms for handling configuration updates. Authors showed also several case studies of why such updates should to be investigated.

In the subject of policies' isolation, Monsanto et al 2013 discussed the ability of SDN architecture or its flows to accept commands from different policies. A policy orchestration abstraction is a possible a solution to orchestrate the process of handling several policies that may come from different applications or departments (e.g. security policies, business process policies, financial policies, audit or monitoring policies). Pyretic or other policy programming languages are proposed as tools to allow users to define policies in a common language. However, in addition to policy language, an abstraction or orchestration layer is still necessary and should not be mixed with policy languages that should be used to describe policies. Network slicing is another layer of abstraction or isolation. However, slicing is performed to isolate logical networks from each other and not policies from each other. In other words, in one slice, we expect to see several policies that need to interact with each other or need to enforce different aspects on the same flows. Different slices deal with different flows and consequently the slice isolation itself should guarantee that policies from different slices should be isolated from each other.

Casado et al 2012 [3]report can be considered as a reassessment of SDN proposed separating the network into three layers or interfaces in terms of control transformation: Hosts, operators and packets. Each one of those layers should have their own control on packets while they traverse the network. Authors argued that traditional Internet has no differentiation among all those interfaces. On the other hand, MPLS has distinguished two of them: Host and packet interfaces. SDN tackled requirements for network operators that were not acknowledged by Internet or MPLS networks. However, SDN did not distinguish between host-network and packet-switch interfaces. Authors proposed a hybrid approach of SDN-MPLS to get the advantages of both and have those three as clearly defined and separated interfaces. Rather than having one controller in original SDN, authors proposed two controllers: A fabric controller to provide basic packet transport (Host-network interface) and an edge controller responsible for complex network services (Operator network interface). Authors here focused on the design of the network and the interfaces related to traffic transportation. In other words, authors focused on the carrier and ignored the fact that the data or the content can have the same problem. Specifically, information is only considered from the network port and on. However, information is created and controlled before that (in the middle-box or the controller). Network control is separated between Fabric and edge controllers. On the other hand, software components that are using the network should be also functionally decomposed. The key idea here is that if two things are functionally not cohesive or that they solve different problems then they should be separated from each other.The edge controller still has several functions that are not cohesive.

In addition to the fact that SDN has different modules that are not functionally cohesive, there is another issue related to the levels of abstractions. A policy that users or administrators understand is at a very different level of abstraction from a policy, or rule, that a firewall or a switch can understand. For example, existing research discussed the challenge between dealing with high level policies at the application level and having to write very low level flow rules in switches or firewalls. The gap between those two can be very large. In addition, low level flow rules should be allowed to change easily and dynamically. On the other hand, high level policies are expected to be more stable and change infrequently. Approaches that tried to give more details to policies suffer from building policies that are network dependent. This makes those policies very complex to reuse or to be able to accommodate network changes. On the other hand, designing policies with that are very abstract can make them very hard to implement or be interpreted in terms of flow or firewall rules. Some papers already proposes an abstraction or adaptor layer between high level policies and flow or firewall rules to isolate them from each other and allow each one to change without impacting the other (e.g. Pan et al 2013 [4], Kang et al 2013 [5], and Moshref et al 2013 [6]).

Abstraction (i.e. network virtualization) was the main reason to bring networking to software programmability. Abstraction in SDN tries to extract simplicity from the control plane (Shenker et al 2011 [7]). It can produce a design that is modular; easy to change, evolve, reconfigure, etc. Authors claimed that network layers include abstraction in the data plane. However, control plane lacks having such layering or abstraction. Authors' report in SDN defined three abstractions: Distribution (controller), forwarding (network virtualization and switches) and configuration (NOS). Distribution of control allows a global view of the network. Forwarding abstraction separates the functionalities in switches between management and forwarding and takes the management task to the controller. The key idea was that those two functions are not cohesive and consequently there is a need to separate or isolate them from each other.

Existing papers or technical reports discussed also abstraction in network or hardware parts of OpenFlow networks (e.g. Danilewicz et al 2014 [9]).

Kind et al 2012 proposed an enhanced SDN architecture where several new splits should be introduced in addition to the split between control and data plane originally proposed in SDN. This includes the split between the control plane and the NOS, which is a modified version of hypervisors' architecture where a basic filter layer can be an alternative to a hypervisor. They also proposed a split between forwarding (network edge) that requires only basic functionalities and processing entities (network core) that requires more intelligent processing and analysis. They focused on applying this SDN modified architecture in carrier grade networks.

Software Defined Internet Architecture (SDIA) is proposed based on SDN to solve the problem of Internet evolution and the need for a flexible architecture (Raghavan et al 2012 [10]). The main idea is to decouple the architecture from the infrastructure so that changes on one will not affect the other. Architecture refers to the current IP based model or any other alternative; while infrastructure refers to physical network resources and equipment. Authors claimed that SDN by its current architecture can help but to a limited range.

Pan et al 2013 proposed FlowAdapter as a middle layer between OpenFlow data plane in switches and the controller [4]. Authors described the goal of such adapter which is to support having flexible rules that can be handled by "inflexible" hardware. In other words, this layer shields both data and control plane from each other and allows changing one of them without a significant overhead on the other. In software design, this is a recommended design principle "Find what is varying and encapsulate it" (Shalloway and Trott 2005 [11])

Sugiki 2013 [8]proposedan integrated management framework to standardize SDN development. This can also contribute to making the development in SDN programming mature. However, our focus in this paper is SDN programming design and not implementation.

Design patterns concepts are also used in traditional networks for network architecture to best layer network components based on some quality aspects (e.g. Dart et al 2013 [12]). Smith et al 2014proposed policy-controlled management patterns in SDN [21]. This is a framework to provide abstraction for orchestrating different services implemented in SDN and that require policy information or interaction. While one of the major goals of SDN was to make the architecture open and vendor independent, however, the fact that currently the area is premature and the effort to develop controllers and applications is not formalized, this may take SDN development to another problem of lacking united or standard architecture. In software engineering, formal methods suffered from such problem for years and this problem is considered one of the main reasons why formal methods are struggling to gain more popularly as it was sought.

III. Goals and Approaches

The goal of good software design patterns and principles is to improve the quality of the developed product. Good design can help the software applications now and in future. This is since a well-designed software should have high qualities such as performance, security, usability that are important for the current usage. In addition for future software maintenance and expansion, good software design can help in maintenance, reusability, testing, etc.

The controller or one of its modules should be able to orchestrate the communication between the different application that are built on top of the SDN network and those applications should be able to share the information without coupling those modules with each other or without security problems.

The applications need not to be aware of each other or communicate directly with each other. The key mechanisms to achieve this are isolation and abstraction. Ideas from software design principles and patterns can be utilized in this aspect. Design principles in general focus on the following main design quality aspects: Abstraction, encapsulation and reusability. Isolation can be also a major benefit to security as it limits the expansion of security intrusions. Figure 1 shows a recent SDN architecture (Alsmadi and Xu 2015).

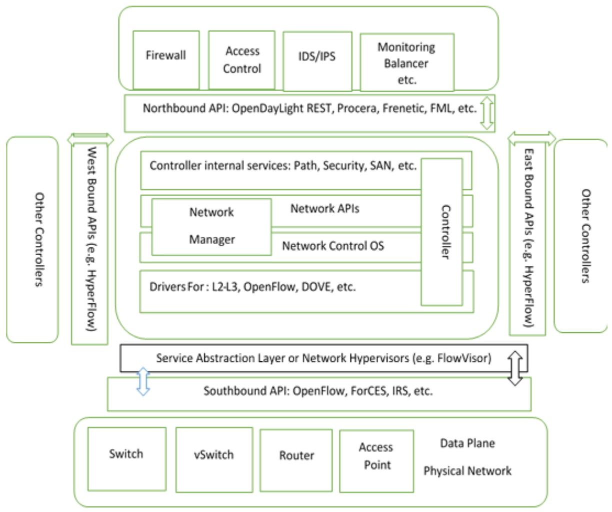

Firewall
Access Control
IDS/IPS
Monitoring Balancer etc.
Northbound API: OpenDayLight REST, Procera, Frenetic, FML, etc.
Other Controllers
West Bound APIs (e.g. HyperFlow)
Controller internal services: Path, Security, SAN, etc.
Network Manager
Network APIs
Network Control OS
Drivers For : L2-L3, OpenFlow, DOVE, etc.
Controller
East Bound APIs (e.g. HyperFlow)
Other Controllers
Service Abstraction Layer or Network Hypervisors (e.g. FlowVisor)
Southbound API: OpenFlow, ForCES, IRS, etc.
Switch
vSwitch
Router
Access Point
Data Plane
Physical Network

Fig. 1. SDN architecture (Alsmadi and Xu 2015)

In comparison with early SDN architectures, Figure 1 shows that interfaces and isolations between the different layers are already in evaluation and development. For example, Service Abstraction Layer (SAL) exists to isolate south bound APIs from the controller. It can help integrating the same controller with more than one south bound API or protocol.

Many researchers and domain experts acknowledge that SDN architecture itself is recent and premature. On the other hand, SDN came with no new technologies or inventions. Rather, it came to clarify, accumulate, and coin findings in the networking field over the past years. From the software engineering or programming side, similarly, SDN does not need to start everything from scratch and can learn from areas where there have been accumulated knowledge and experience over the years. Software design in particular is considered a matured field and concepts related to: Object Oriented Design (OOD), software construction, testing, design principles and patterns have a rich inventory of: tools, methods, etc. that can be utilized.

OpenFlow protocol itself can be considered, in a very simple manner, an abstraction layer or adapter to allow software programs to interact with switches. This is since for commercial or business, not technical reasons, vendors of switches and routers don't allow developers to program or interact with traditional switches. In that sense, OpenFlow protocol provides that well-defined interface to program or communicate with switches. This communication or programming can be conducted through the controller.

We will present all areas in SDN architecture that should include separatesoftware communication adaptors. Those adaptors should include interfaces to facilitate communication between their edge modules. In some cases where communication is two-ways, two different interfaces should be designed. We also showed some contributions already in some areas to indicate that research is already going in this direction.

South Bound Interface: An abstraction layer or adapter should exist between controller and its switches. A well-defined interface or public methods should be defined on switches to allow the controller to access switches only through those public interfaces. The interface that switches should expose depends on the type of services that they provide.For example, controller should be able to read flow rules, add new flows, delete or update flow rules (i.e. CRUD on flow rules). As software classes, there should be two main classes for the interaction between controller and switches: Flows and flow rules. Controller should be allowed to change some of those classes' attributes through setters and getters.

From a software design perspective, in the flow rules insertion process, the controller, as a client should fulfill all flow insertion constraints or network invariants before being allowed to insert a new flow rule. Controller can then have a separate monitoring module that will be queried to retrieve those invariants. However, some of those constraints such as rules-conflict can be only judged after adding the flow to the switch flow tables. Abstraction layer module should then orchestrate the process and start a roll-back process where after the insertion if a rule-conflict case occurs between the new added rule and existing ones, the addition process is reversed with all related activities.

Existing research already proposed a software abstraction layer between SDN controller and switches for several different purposes. For example, Khurshid et al 2013 proposed VeriFlow as a verification layer between controller and switches [13]. The goal of this layer to verify that flow rules inserted in the switches from the controller do not violate certain network wide invariants (e.g. reachability, loop freeness, consistency).

Policies should be isolated from flow rules. Policies should include high level information about resources (e.g. user, host, application, etc.). They should not include information that are low level dependent (e.g. IP address, port, MAC address, etc.). In networking terminologies, L1-L3 information are considered low level information, L4-L7 information are considered high level information. An abstraction layer should exist to separate and isolate those two layers from each other.

Policies in network security serve three different levels:

Application level policies: At the application level, users write policies to regulate users-applications-systems interactions. They can specify who can do what, when and how. However, at this level, users are not identified as individuals but as groups. Network, systems and applications are only identified by general names without any technical terms. Typically, at this level, we expect policies such as:

Employees should not be able to access accounting services remotely.

Students should not be allowed to use smart devices during exams.

Users can have unlimited Internet download speed only after working hours.

In those examples, we showed that at this level, policies or policy sets should be for groups and not individuals (as individuals represent instances of their groups which can be specified in level2 policies). Similarly, applications and devices are known by general categories that can have several instance examples (e.g. accounting, smart devices, Internet).

For simplicity we will call them at the first level as policy sets, at the second level as policies and at the third level as rules. Policy sets include policies and policies include rules.

Middle level policies: Level two should include information typically included in Access Control Lists (ACLs). This is an intermediate stage between high level policy sets and low level rules. Every authorized person, application, or service should have an entry in this access control system. There are currently several examples of ACLs such as those that exist in operating systems, databases or websites active directory or user management, ACLs in firewalls, port control, and routers.

Low level policies: Rules in flow tables and firewalls in particular. Those should have the same attributes exist in flows so that checking and matching those rules with flows can be simple, dynamic and direct. Since those rules will talk to and direct low level network components, for performance issues, they need to be simple and straightforward. Unlike, high level policies, those rules and location dependent and include network level information (e.g. IP, MAC addresses, port number, etc.).

Two-way communication should be orchestrated between each two consecutive layers. From top to bottom, special tools should be developed to allow automatic translation from high to low level terminologies. On the other hand, information from bottom up should be used to improve policies. ACLs in the middle layer provide constraints on flows at the low level. On the other hand, a special module should be developed to support a feedback control where information from network flows can be used to trigger future rules in ACL. Data mining, Artificial Intelligence (AI) and patterns' recognition methods can be used to analyze network traffic and make rules' recommendations. Those can be triggered for security purposes such as breaches or attacks or they can be triggered for QA purposes (e.g. performance). Between ACLs and high level policies, modules should be developed to allow automatic translation of policies to ACLs. On the other hand, feedback control is also recommended to reevaluate existing policies or trigger adding new ones based on network traffic and environment.

In this specific category, Qazi et al 2013 proposed SIMPLE as a layer between security policies and flow rules [15]. This layer is required to isolate L2-L3 low level layers' required information from L4-L7 policies' information.

Controller Internal Interfaces: Controller and its internal modules should be separated from high level middle-boxes and applications (e.g. firewall, IDS, load balancer, etc.). Controller can provide services to those applications through well-defined interfaces.

This is currently the most important abstraction layer to provide. This abstraction should exist between the controller as a complete module and any other applications that should be developed and that will interact with the controller. In other words, those are not built-in modules in the controller Opendaylight and many other controllers are currently using REST API for this specific purpose.

Existing research proposed several examples of either security or other types of applications that should be developed on this top level or also called southbound section (e.g. load balancer, monitoring tools, etc.). Existing research also discussed security concerns when developing such applications and showed that there is a need to make interactions between the controllers and those applications in such manners that guarantee isolation in terms of security, configuration, reconfiguration, reuse, etc.

In the controller itself also, Network Operating System (NOS) should be isolated from the controller. Those are two different units with two different functionalities or responsibilities. Network hypervisors already exist to isolate those two components from each other and allow the communication between them. In this section in particular terms such as (network APIs) are used to refer to this abstraction or isolation layer between controller services and NOS.

Existing research already proposed different structures for such layer. For example (Porras et al 2012 [16] and Shin et al 2013 [17]) proposed a security control or interface between controller and security applications to allow deploying composable security services. Fayazbakhsh et al 2013 proposed FlowTag to tag flows in middle-boxes so that controller can know which application originates a particular flow [18].

Controller internally includes several functions that are not cohesive with each other and consequently should be isolated from each other. Those can be largely divided into: Control and security, administration and management, flow management and communication with switches, monitoring, and load balancing. A high level application which needs only one of those components to communicate with should not be coupled with all other components. In addition for security purposes, isolating those components from each other can limit the spread of attacks and help detecting them. As an example, let's assume we have the monitoring and control separated in two different modules, if control module is compromised, monitoring can help administrators see the details and take counter actions. Separating those modules can be logical only or it can be also physical (i.e. on different controllers or slices).

Logical or Functional Interfaces: Isolation or virtualization already exists in SDN in several other popular areas. The first one is the isolation between the different VMs where they are logically isolated from each other but may run in the same tenant or physical resources. Tenants are isolated from each other where each tenant in a cloud datacenter represents a different company that should be completely isolated from other tenants. Each tenant can have one or more VMs based on demand or requirement. For scalability and load balancing issues, controller tasks can be divided into different functional or logical slices. In this case, we want the different slices to communicate with each other. However, again such interaction should be conducted through well-defined interfaces and data or control from each slice should not interfere with those of the others.

Controller Distributed Architecture Interfaces: A single controller is not a realistic approach for production networks.

Consequently a cluster of controllers should exist to support each other and coordinate tasks' distribution. However, current OpenFlow standard does not allow inter-domain information exchange between the different controllers. Nonetheless, there are many use cases that justify the need for those different controllers to communicate with each other. Needless to say, that such communication between the different controllers should be performed through well-defined interfaces. Different controllers should not interfere with each other or violate others' security regulations. The exchange interface should consequently orchestrate such communication.

A virtualization layer such as FlowVisor (Sherwood et al 2009 [19]) is proposed to isolate the controller or a cluster of controllers from the underlying physical and networking elements. This logical layer intercepts all messages between controllers and switches. From the virtual world perspective, FlowVisor acts as VMWare or Virtual Box that logically isolates operating system from physical components. Consequently, different operating systems can work on the same machine or physical resources.

An adaptor should exist in the data plane between forwarding functions (network edge) and processing functions (network core) (Kind et al 2012[20], Raghavan et al 2012[10]).

Design patterns

The controller is the central control and management module in SDN. Even in controller distributed architecture, each switch is expected to communicate security with only one controller or controller slice. In design pattern a singleton pattern is proposed in such cases to enforce communicating with only one single instance. A singleton pattern should mediate communication between controller and switches from one side and controller and applications from the other side. From software design patterns, a singleton class or pattern is needed when we want a class to be available to the whole application and also that the whole application should have one and only one instance. This seems to be the same usage profile of the controller.

Design patterns' assessment

The usage of design patterns in software programs showed a mature level of experience from the software development team. In order to evaluate the usage of design patterns in SDN, we conducted several experiments using Java controllers (e.g. Floodlight, Beacon, FlowVisor, IRIS, and Maestro). We used tools that can perform automatic design patterns' detection (e.g. Pattern4, PDE, Pinot, etc.). Preliminary results showed few instances of usage of design patterns in those SDN or controller programs.

OpenFlow Software version

SDN separates and isolates control from data planes. Control plane (i.e. the controller) is completely a software application. OpenFlow protocol is developed to control communication between the controller and the data plane or the switches. However, we argue that OpenFlow focused on defining how controller should communicate with the switches from a networking or communication perspective. OpenFlow should be extended to specify how controller should communicate with the switches from a software perspective. This is since controller is completely a software and switches include also embedded software elements. In this section, we will focus from a high level perspective on what should OpenFlow software version (OpenFlowS) should include.

In SDN, controllers add flows to switches. A separate software module should exist to orchestrate and handle communication between controller and switches. Part of the functions that this orchestrator or adaptor module should handle is checking preconditions or constraints before allowing controller to add, delete or update a flow rule. This module should be responsible for tracking all network state related attributes. For example, through the flow-addition process, inconsistency can occur where the network state is changing while the controller is adding the flow rule based on expired information. This is usually referred to as "race condition". Proactive controllers may eventually notice and fix such inconsistencies. However, this will add complexity and overhead over the controller and the network (Peresini et al 2013 [14]).

From a software perspective, controller and switches are two modules or packages that need to interact with each other and provide services to each other. In this specific architecture, in most cases the controller is the client and the switches play the software service provider role. For example, the controller needs to CRUD (i.e. Create, Read, Update and Delete) rules from switch flow table(s). Those should be represented by public services in the switch. A switch should include a software interface with those four as public services. The inputs that controller will provide when calling those services depend on the specific method and will include flow rule attributes as well as other possible attributes related to programming or design (e.g. rule ID, group, etc.).

OpenFlowSadaptorshould typically include abstract classes or interfaces. Those should regulate how controller and switches interact with each other. For example, they should include the, high level, public services that should be provided along with their headers and signatures. They should also provide pre-conditions and post-conditions for those public services.

The organization behind SDN and OpenFlow (Open Networking Foundation, ONF) in a report in 2012, identified four major goals for SDN. Those four goals are related to software design and principles:

SDN aims at decoupling switch management and control plane from data plane. Coupling and cohesion are widely popular software design metrics upon which good design quality is evaluated. Internal modules should be highly related and interconnected to each other from functional perspectives (i.e. high cohesion). On the other hand, different modules should be lightly coupled with each other and only through well-defined interfaces.

A centralized controller to manage one to many switches. Different switches are controlled by same functionalities. They are only different in low level details that should be handled by switches themselves. High level functions that are similar should be centralized in one location. Low level details are distributed across the different switches.

We described this goal from two perspectives. The first one is related to policies in which global policies should be centralized, location independent and express the general design goals of the network. However, each switch or security control can have their own rules that are location dependent and that interpret high level policies from a small local scope perspective.

The other perspective is related to modules' isolation through the different adaptors. Communication through the different modules in such case can be formalized. One of the ambitious goals of SDN is to achieve automating different policy activities: Implementation, enforcement, configuration, orchestration, etc. A modular design is a key toward achieving such goal.

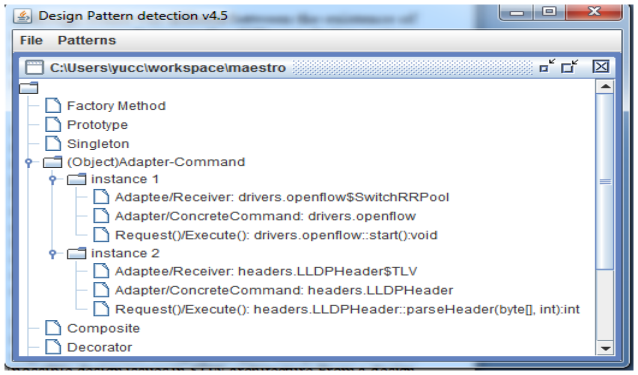
SDN should support the ability to manage network behavior through well- defined interfaces and modules. In other words, the "what part" should be separated from the "how part" and well-defined interface should exist and isolate them from each other. Although control in SDN can add or delete switch flow rules in the flow tables, however, this should not mean to allow controller to tamper internal switch architecture.

Using well-defined interfaces between controller and switches makes it possible for one controller to interact smoothly with switches from different vendors. A software open adaptor with well-defined interface should be designed between controller and different switches to shield both of them from each other and to formalize communication between them.

Deign Patterns Evaluation

In order to assess the level of using design patterns in SDN, we evaluated three open source controllers written in Java: Floodlight (www.projectfloodlight.org/floodlight), IRIS (openirisproject.tumblr.com) and Maestro (code.google.com/p/maestro-platform). The goal is to evaluate which such relatively large software applications are developed based on mature design perspectives. We should acknowledge however the debate on whether the usage of design patterns indicate a mature good design or not.

Evaluating Floodlight with Pattern4 design pattern detection tool, results showed only using one design pattern (Singleton) with 53 instances. No other design pattern is shown to be used in Floodlight.



C:\Users\yucc\workspace\maestro
Factory Method
Prototype
Singleton
(Object)Adapter-Command
instance 1
Adaptee/Receiver: drivers.openflow$SwitchRRPool
Adapter/ConcreteCommand: drivers.openflow
Request()/Execute(): drivers.openflow::start():void
instance 2
Adaptee/Receiver: headers.LLDPHeader$TLV
Adapter/ConcreteCommand: headers.LLDPHeader
Request()/Execute(): headers.LLDPHeader::parseHeader(byte[], int):int
Composite
Decorator

Fig. 2. Design pattern detection tool: Pattern4

Table 1 shows design patterns' instances in Maestro based on Pattern4 tool.

TABLE I. PATTERN4 ON MAESTRO CONTROLLER

pattern/class	Singleton	Adapter	State/Strategy	Visitor
	1	2	3	7

Table 2 shows design patterns' instances in IRIS based on two design pattern detection tools; Pattern4 and Web Of Patterns (WOP). Results showed the wide variation between the two tools in detecting the occurrence of design patterns usage.

TABLE II. PATTERN4 VS WOP ON IRIS CONTROLLER

abstract factory	bridge	Proxy	Template Method	State /Strategy	Decorator	Adapter	Singleton	Pattern Tool
0	0	2	3	23	1	28	20	**pattern4**
2	1874	62	96	0	0	478	0	**WebOfPatterns (WOP)**

TABLE III. PATTERN4 VS WOP ON OPENFLOWJ CONTROLLER

abstract factory	bridge	Prototype	Proxy	Template Method	State /Strategy	Decorator	Adapter	Singleton	Pattern Tool
0	0	4	0	0	13	0	5	22	**pattern4**
0	0	0	0	0	0	0	12	0	**WebOfPatterns (WOP)**

TABLE IV. PATTERN4 ON OPENFLOWJAVA-MASTER AND LACP-MASTER CONTROLLERS

Controller	Singleton	State/Strategy
openflowjava-master	6	4
lacp-master	12	0

Although results was different between the existence of design patterns between the different design pattern detection tools, however, results showed that such relatively large and important software applications are not considering the usage of design patterns.

IV. CONCLUSION

SDN is an emerging architecture for designing networks and distributing their functionalities. This new architecture took into consideration some of the problems and challenges in traditional networks. Two features in SDN can be considered as core and differentiate SDN from traditional networks. The first one is splitting control from data plane and moving it from switches to a new software program called the controller. The second one is in making this controller open as a vendor independent and to be programmed and extended by developers and users.

SDN is about programmable networks and consequently it makes sense to use some of the mature software experience such as design principles and patterns to solve possible design issues in SDN architecture.From a design perspective, those can make the network design easier to use, reuse, update, maintain and interact with. Software in SDN has a major role and this role is expected to continuously grow. From a software design perspective, SDN architecture is not mature enough. This explains why recent implementations of open controller should as Opendaylight extend the architecture to include new abstraction layers such as: Service Abstraction Layer (SAL) and REST API in two different locations of the architecture.

SDN focuses on the networking perspective when control is isolated from data. From a software perspective, there are many components in the SDN architecture that are not cohesive and that should be decoupled from each other. We showed also that there are already research proposals and progresses toward that goal. This will continue to evolve and we presented here a picture of the possible abstraction layers that SDN architecture may end up having. Those abstraction layers are necessary to allow different components that are not cohesive and that should be decoupled to communicate with each other through well-defined interfaces. This will allow those different modules to be used and evolve aside from each other.

REFERENCES

[1] 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.Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14.

[2] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, David Walker: Abstractions for network update. SIGCOMM 2012: 323-334

[3] M. Casado, T. Koponen, S. Shenker, A. Tootoonchian, "Fabric: A Retrospective on Evolving SDN," in Proc. of the first workshop on Hot to pics in software defined networks, Helsinki, Finland, Aug. 2012, pp. 8 5-90.

[4] Heng Pan, Hongtao Guan, Junjie Liu, Wanfu Ding, Chengyong Lin, GaogangXie: The FlowAdapter: enable flexible multi-table processing on legacy hardware. HotSDN 2013: 85-90

[5] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker, Optimizing the “One Big Switch” Abstraction in Software-Defined Networks CoNEXT’13, December 9–12, 2013, Santa Barbara, California, USA.

[6] M. Moshref, M. Yu, A. Sharma, and R. Govindan, “VCRIB: Virtualized rule management in the cloud,” in NSDI, Apr 2013

[7] Scott Shenker, Martin Casado, TeemuKoponen, and Nick McKeown, The future of networking and the past of protocols, A presentation, June, 2011

[8] AkiyoshiSugiki: An integrated management framework for virtual machines, switches, and their SDNs. ICON 2013: 1-6

[9] G. Danilewicz, M. Dziuba, J. Kleban, M. Michalski, R. Rajewski, M. Grief: Project ALIEN - abstraction layer for devices non-OpenFlow SDN networks. Telecommunication Review September 2014

[10] BarathRaghavan, Martin Casado, TeemuKoponen, Sylvia Ratnasamy, Ali Ghodsi, Scott Shenker: Software-defined internet architecture: decoupling architecture from infrastructure. HotNets 2012: 43-48

[11] Alan Shalloway and James Trott, Design Patterns Explained, A new perspective on object oriented design, second edition, Addison-Wesley, 2005.

[12] Eli Dart , Lauren Rotman , Brian Tierney , Mary Hester , Jason Zurawski, The Science DMZ: a network design pattern for data-intensive science, Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, November 17-21, 2013, Denver, Colorado [doi>10.1145/2503210.2503245]

[13] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” ACM SIGCOMM Computer Communication Review, vol. 42, no. 4, pp. 467–472, 2012, 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2013.

[14] Peter Peresíni, MaciejKuzniar, DejanKostic: OpenFlow Needs You! A Call for a Discussion about a Cleaner OpenFlow API. EWSDN 2013: 44-49

[15] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, “SIMPLE-fyingMiddlebox Policy Enforcement Using SDN.” ACM SIGCOMM, August 2013

[16] Phillip Porras , Seungwon Shin , Vinod Yegneswaran , Martin Fong , Mabry Tyson , GuofeiGu, A Security Enforcement Kernel for OpenFlow Networks, HOTSDN 2012.

[17] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks,” in Proceedings of the 2013 ACM conference on Computer and communications security, ser. CCS ’13. ACM, 2013

[18] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul, “FlowTags: Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions,” in Proceedings of the second workshop on Hot topics in software defined networks. ACM, 2013.

[19] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. OpenFlow Switch Consortium, Tech. Rep, 2009.

[20] Kind, M., Westphal, F.J., Gladisch, A., Topp, S.: SplitArchitecture: applying the software defined networking concept to carrier networks. In: IEEE World Telecommunications Congress (WTC) (2012)

[21] Paul Smith, Alberto E. Schaeffer Filho, David Hutchison, AndreasMauthe: Management patterns: SDN-enabled network resilience management. NOMS 2014:1-9

[22] Alsmadi, Izzat, Dianxiang Xu, Security of Software Defined Networks: A Survey, Computer and Security Journal, Elsevier, Volume 53, 09/2015.