**Software Defined Networking using Frenetic**

*Gunjan Sharma*

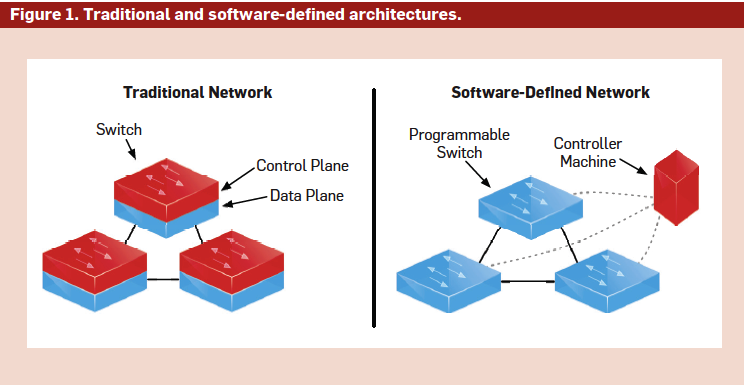
**Abstract**

Over the last couple of years Software Defined Networking has become the hottest buzzword – not just in enterprise networking but also among service providers- with research networks, large data centers hosting and cloud providers as some of the earlier adapters. SDN has received a lot of attention in recent years as a means of addressing some of the long-standing challenges in networking.

An emerging concept that promises improved network performance, flexibility and manageability, SDN has the potential to revolutionize networking and unlocking a wave of innovation for application services. To better understand the benefits of SDN, it is important to look how it fits in the broader perspective and ask what problems it will solve and whom it will help. We intend to find answers to many more questions mentioned above by understanding not only theoretical but also the practical concepts behind SDN. Since it has been standardized, we will also know more about the protocol OpenFlow and the programming languages used to program network equipment’s.

Let us begin by quoting the definition of Software Defined Networks. “Software-defined networking (SDN) is an approach to computer networking that allows network administrators to manage network services through abstraction of lower-level functionality [10].” Some also define it, as “SDN is a new network architecture that decouples the software that controls a network from the devices that implement it.” Software Defined Networking (SDN) provides an abstract layer over the physical network layer. This abstract layer acts as a controller and allows network administrators to have a programmable control of network traffic.

The concept is based on split archi­tecture, which separates forwarding functions from control functions. This decoupling removes some of the com­plexity from network management, providing operators with greater flexi­bility to make changes [2].



[3] Figure 1 contrasts the architectures of traditional networks and SDN. In SDN, one or more controller machines execute a general-purpose program that responds to events such as changes in network topology, connections initiated by end hosts, shifts in traffic load, or messages from other controllers, by computing a collection of packet-forwarding rules. The controllers then push these rules to the switches, which implement the required functionality efficiently using specialized hardware. Because SDN does not specify how controllers are implemented, it can be used to implement a variety of network algorithms, including simple ones such as shortest-path routing, and more sophisticated ones such as traffic engineering.

**Background**

SDN fills the gap made by traditional networking approaches that over time became too complex, closed and proprietary. The main cause of a network’s limitation is that it is build using switches, routers, and other devices that are complex in nature as they implement many distributed protocols and use closed and proprietary interfaces. This makes network administrator’s job very hard to implement something new. SDN helps to physically separate the control plane from the forwarding plane and a single control plane controls several forwarding devices.

OpenFlow is an attempt to unify the various standards and proprietary that existed in network’s hardware like switches, routers and other devices from various vendors. OpenFlow is an outcome of research at Stanford University and University of California at Berkeley for six years. OpenFlow protocol helps to manage and direct traffic among the network devices like switches and routers. Today OpenFlow is being adopted by leading vendors to help enabling researchers and network administrators to try new things.

Using OpenFlow, researchers have been able to configure and manage resources across campus networks. The most well-known controller platform is NOX.

“*OpenFlow also enabled flexible access control, Web server load balancing, energy efficient networking and seamless virtual-machine migration* [9] *.*”

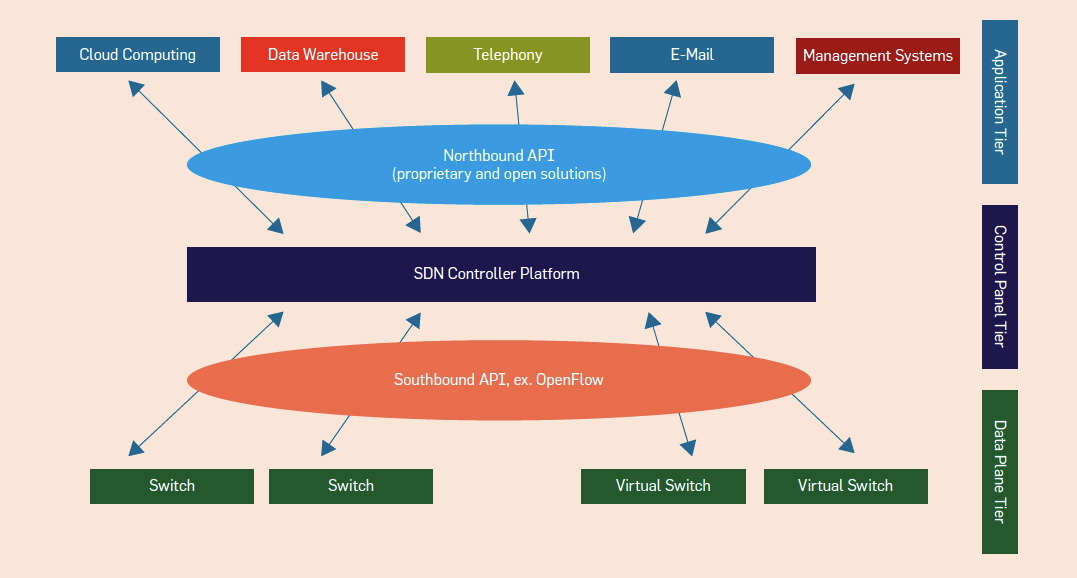
Today new languages like Frenetic are coming up to overcome barriers created by OpenFlow. The barriers made by OpenFlow are that it is not easy to get certain tasks done. First, decoupling various network tasks like routing, access control and traffic monitoring in separate small tasks is not possible using OpenFlow. Packet handling rules put by one module in place often overlap rules put by other modules. Second, OpenFlow does not provide a high level abstracted view of switch hardware. Third, not all packets are processed at the controller layer; some are still processed on switches. This created two layers of controlling packets, which in turn increases complexity. Fourth, certain programs can be erroneous [9]. Frenetic programming language for networks provides ways to overcome the limitations created by OpenFlow.

**Discussion**

**1) Software Defined Networks**

Software Defined Networking (SDN) has changed the way networks are handled now. It has two main features. Firstly, SDN provides an abstract layer over the physical network layer thereby separating the control plane (which decides how to handle the traffic) from the data plane (which forwards traffic according to decisions from the control plane). This abstract layer acts as a controller and allows network administrators to have a programmable control of network traffic. Secondly, SDN consolidates the control plane, so that one software control program controls multiple data-plane devices [11]. The ultimate goal of SDN is to “provide open user-controlled management of the forwarding hardware of a network element.” SDN operates on the idea of centralizing control-plane intelligence, but keeping the data plane separate.

**Architecture of a SDN [1]**



1) Control Plane: The control plane/controller presents an abstract view of the complete network infrastructure, enabling the administrator to apply custom policies/protocols across the network hardware. The network operating system (NOX) controller is the most widely deployed controller [4].

2) Northbound Application Interfaces: The “northbound" application programming interfaces (APIs) represent the software interfaces between the software modules of the controller platform and the SDN applications running atop the network platform. These APIs expose universal network abstraction data models and functionality for use by network applications. The “northbound APIs” are open source-based [4].

3) East-West Protocols: In the case of a multi-controller-based architecture, the East-West interface protocol manages interactions between the various controllers [4].

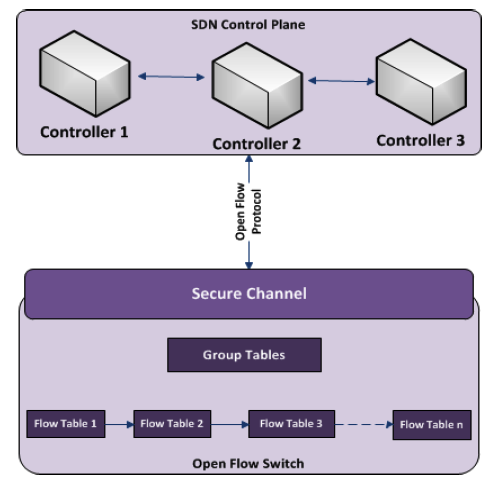
4) Data Plane and Southbound Protocols: The data plane represents the forwarding hardware in the SDN network architecture. Because the controller needs to communicate with the network infrastructure, it requires certain protocols to control and manage the interface between various pieces of network equipment. The most popular “southbound protocol” is the OpenFlow protocol [4]. The following section explains OpenFlow and its architecture

The advantages of using SDN architecture include that network can be controlled programmatically as it is decoupled from forwarding functions. Network wide traffic can be dynamically administered and more centrally managed. Most of all, it provides simplified network design and operation because instructions are provided by SDN controllers instead of proprietary-based specific vendor devices [12].

**2) OpenFlow**

OpenFlow is the protocol used for managing the southbound interface of the generalized SDN architecture. It is the first standard interface defined to facilitate interaction between the control and data planes of the SDN architecture. OpenFlow provides software-based access to the flow tables that instruct switches and routers how to direct network traffic. Using these flow tables, administrators can quickly change network layout and traffic flow. In addition, the OpenFlow protocol provides a basic set of management tools, which can be used to control features such as topology changes and packet filtering.

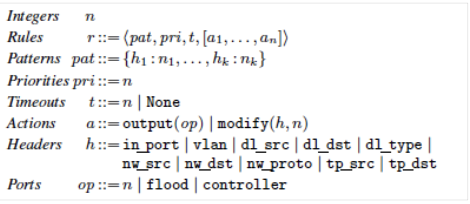
**OpenFlow Architecture [4]**



The OpenFlow specification is controlled and defined by the nonprofit open network foundation (ONF), which is led by a board of directors from seven companies that own and operate some of the largest networks in the world (Deutsche Telekom, Facebook, Google, Microsoft, Verizon, Yahoo, and NTT). Most of the networking hardware vendors such as HP, IBM, and CISCO offer switches and routers that use the OpenFlow protocol [13]. OpenFlow shares much common ground with the architectures proposed by ForCES and SoftRouter; however, the difference lies in inserting the concept of flows and leveraging the existence of flow tables in commercial switches

OpenFlow protocol is the most integral part for implementing SDN. OpenFlow protocol helps to manage and direct traffic among the network devices like switches and routers. The OpenFlow specification defines a standard collection of features switches must provide, as well as an interface controllers can use to communicate with switches: instructions for installing and deleting forwarding rules, and notifications about flows, topology, and traffic statistics. An OpenFlow switch maintains a forwarding table that contains a list of prioritized rules. Each rule has a pattern that describes a set of packets and actions that describe transformations on packets. When a packet arrives at a switch, the switch finds a rule whose pattern matches the packet headers and applies the associated actions. If multiple rules match, the switch applies the actions of the highest priority rule, while if no rules match, the switch encapsulates the packet in an OpenFlow message and sends it to the controllers. The controllers can either process the packet directly, or send messages back to the switch instructing it to install or delete rules in its forwarding table. The maximum size of a table is determined by hardware constraints, but most switches have space for at least several thousand rules.

In figure below, OpenFlow syntax is shown [9].

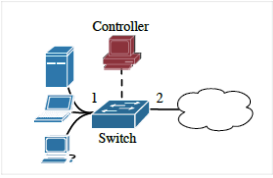


An OpenFlow network consists of a centralized controller, which manages a distributed collection of switches. Now the decision about how to handle traffic can happen both in the centralized controller and the switches themselves. Since the controller takes more time to process the packets, the controller handles only new or unexpected packets and the switch is configured to handle rest of the traffic.

Role of a switch: When the packet reaches the switch, the switch will first select a rule from its flow table. A flow table is used to keep a set of rules that specify how packets should be processed. If rule exists on how to handle the packet then the exact-match rule will be used. Exact-match rule has pattern matching every header in the field. It may also be the case that an exact-match rule is not there, so a wildcard rule with highest priority is chosen. After updating the byte and packet counter associated with the rule, the action listed on the rule is applied to the packet [9].

Role of a controller: In case of new and unexpected packets, there are no matching rules in the flow table. So the controller handles the processing of the packet in this case [9].

In figure below [9], a simple topology of a controller and switch is shown when packets are being passed from point 1 to point 2.



NOX controller is the most widely used controller for OpenFlow. NOX is a simple operating system that enables to write basic methods for networks. However, an analysis of OpenFlow/NOX shows that this system has several limitations and programming language called Frenetic aims at bridging this gap. There are other controllers like Onix, Beason and Nettle for OpenFlow as well, but all suffer similar limitations [9].

Four problems with OpenFlow/NOX:

1. “NOX Programs do not compose [9].” It only allows setup rules for monitoring web traffic by increasing the byte and packet counters associated with those rules. The problem is that these rules may be overlapping and so will not be efficient in routing, access control and traffic monitoring. So composing NOX programs requires great deal of effort to make sure different rules are not conflicting.

2. No higher layer of abstraction is present so there is a great deal of low level programming. For example, if we need to monitor all incoming web traffic except from a specific location. To do so we will have to write two low-level overlapping rules that work well together. This method tends to get cumbersome and error-prone [9].

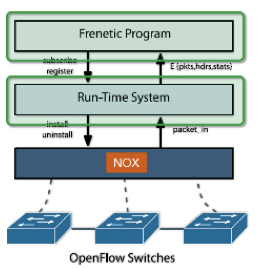
3. Since the controller and the switches form two-layer architecture and decisions about traffic flow take place in both layers, it might happen so that for asynchronous tasks it gets difficult to read and reason different rules. The controller program must specify communication patterns between two programs and deal with subtle concurrency issues.

4. Lastly, “programs are susceptible to subtle network race conditions [9].” When a new packet is received generally it is send to the controller and a rule is made on switch on how to handle those kinds of packets. But if before the switch is setup, another packet comes in, then even that packet will go to controller and a new rule might be setup. This can lead to certain packets failing to reach the destination. The programming language Frenetic aims at solving these problems of OpenFlow/NOX.

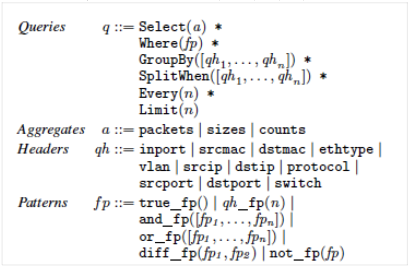
**3) Frenetic**

Frenetic is a language that gives programmers a high level view of OpenFlow networks. Programmers can use the network query language provide by Frenetic framework to get more information about traffic flow. Frenetic also provides network policy management library, which enables programmers to read network state. In figure below, the Frenetic architecture is displayed which shows how the high level language handles run-time system and governs the low level rules on OpenFlow switches.

**Frenetic Architecture [8]**



In figure below, the simple querying syntax used by Frenetic language is shown. According to the developers of Frenetic platform, “the key challenge is to find a balance between expressiveness, simplicity and control over cost [9].” Nevertheless this language provides the much-needed abstract view of run-time network traffic flows.



Frenetic solution to the four problems with OpenFlow/NOX:

1. This simple example shows how monitoring web traffic is easier with the help of Frenetic, which was cited as the first problem of OpenFlow/NOX earlier in this paper [9].

**def web\_query():**

**return \**

**(Select (sizes) \* Where (inport\_fp(2) & srcport\_fp(80))) \* Every (30))**

The first statement indicates that this is a definition of a method named web\_query. The second statement suggests that this method has a return type of stream of integers. The third statement is a simple SQL type query; it selects all packets arriving on physical port 2 and from TCP source port 80. It sums the sizes of all such packets every 30 seconds and returns the stream of integers. The results of this type of query can be used for traffic analysis, for monitoring security and for decisions regarding forwarding policy. Using the following command we can simply print the results to a printer.

**def web\_stats():**

**web\_query() >> Print()**

2. Unlike NOX, Frenetic supports the high level abstraction. The following example makes it clearer [9].

**def host\_query():**

**return (Select (sizes) \* Where (inport\_fp(2)) \* GroupBy ([dstmac]) \* Every(60))**

The above query summarizes the total volume of traffic arriving on physical port 2, grouped by destination host, every 60 seconds. This query runs in a frequency of 60 second and can be easily merged with a web\_query, which runs on the frequency of 30 seconds. Using NOX the same task will be tedious.

**def all\_stats():**

**Merge (host\_query(), web\_query()) >> Print()**

Using Frenetic each query can “see every packet” in the network. Thus, installing one query in the run-time does not silently stop any other queries from seeing packets.

3. It supports single tier programming so it sees every packet in the network. This helps remove many of the two tiered model complexities.

4. Race-free semantics is supported in Frenetic. The following example shows how a switch can be made to do learning [9].

**def learning\_query():**

**return (Select (packets) \* where (true\_fp()) \* GroupBy([srmac]) \* SPlitWhen ([inport]) \* Limit (1))**

**def connection\_printer():**

**learning\_query() >> Print()**

This query enables learning by returning one packet for each distinct source MAC, unless the port associated with the source MAC changes. Whereas for a NOX program it takes long to generate and install a rule to suppress packets 2, 3, 4 with the same source MAC, by then these packets might already have arrived at the switch or the controller and get processed again by controller [9].

Hence, the key advantages of Frenetic include Declarative design that appeals more to programmers. Modular Design so only limited network-wide effects. The single-tier programming concept supports see-every-packet abstraction. It is also race-free semantics and the query language is designed in a cost control manner [8].

**Conclusion**

The discussion in this paper helps to understand Software Defined Networks (SDN). SDN provides an abstract layer to physically separate the control plane from the forwarding plane and a single control plane controls several forwarding devices. OpenFlow protocol is the most integral part for implementing SDN. OpenFlow protocol helps to manage and direct traffic among the network devices like switches and routers. NOX controller is the most widely used controller for OpenFlow. NOX is a simple operating system that enables to write basic methods for networks. However, an analysis of OpenFlow/NOX shows that this system has several limitations and programming language called Frenetic aims to bridge this gap.

Frenetic helps to monitor web traffic with ease. Unlike NOX, Frenetic supports high-level abstraction. Frenetic supports single tier programming so it sees every packet in the network. This helps remove many of the two tiered model complexities. Frenetic enables learning by returning one packet for each distinct source MAC, unless the port associated with the MAC changes.

Currently people are working to extend Frenetic in several directions. One thread of work is developing security applications for performing authentication and access control, and for ensuring isolation between logical networks that share a common physical infrastructure. They are also designing a new run-time system that generates rules from the registered subscribers and forwarding rules eagerly. They plan to compare the tradeoffs between different rule generation strategies empirically.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**References:**

1. Software-Defined Networking – “Novel architecture allows programmers to quickly reconfigure network resource usage” by Keith Kirkpatrick.
2. Software-defined networking – “The service provider perspective” by Ericsson Review (The communications technology journal since 1924)
3. Abstractions for Software- Defined Networks – “New abstractions are critical for achieving SDN goals.” by Martin Casado, Nate Foster and Arjun Guha.
4. Software-Defined Networking – “State of the Art” by Manar Jammala1, Taranpreet Singha, Abdallah Shamia, RasoolAsalb, and Yiming Lic.
5. Software Defined Networking by David Meyer.
6. Languages for Software-Defined Networks by Nate Foster, Michael J.Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich,Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alee Story and David Walker.
7. OpenFlow » What is OpenFlow? (n.d.). Retrieved from

<http://archive.openflow.org/wp/learnmore/>

1. Foster, N., Harrison, R., Freedman, M. J., Monsanto, (2011). Frenetic slides: Frenetic: A High-Level Language for OpenFlow Networks.
2. Foster, N., Harrison, R., Freedman, M. J., Monsanto, (2011). Frenetic: A Network Programming Language. Sigplan Notices. doi:10.1145/2034574.2034812
3. <http://en.wikipedia.org/wiki/Software-defined_networking>
4. Ferguson, A. D., Guha, A., Liang, C., Fonseca, Networking: An API for Application Control of SDNs.
5. Software-Defined Networking (SDN) Definition - Open Networking Foundation. (n.d.). Retrieved from <https://www.opennetworking.org/sdn-resources/sdn-definition>
6. IBM, Software-Defined Networking: A New Paradigm for Virtual, Dynamic, Flexible Networking, October 2012

**Book reference:**

1. SDN : software defined networks by Nadeau, Thomas D.

**Web reference:**

1. <http://searchsdn.techtarget.com/tip/Network-programming-languages-Frenetic-and-Pyretic-simplify-OpenFlow>
2. <https://www.opennetworking.org/sdn-resources/sdn-definition>
3. *Frenetic*: A Programming Language for Software Defined Networks http://www.frenetic-lang.org/