Section 1: Week 1: Software Defined Networks

Nate Bachmeier

TIM-7010: Computer Networking and Mobile Computing

May 19, 2019

North Central University

# Software Defined Networking

Traditional networks are built as ‘thick closed systems’ and are intended for deployments that are statically provisioned. The monolithic design of these Network Functions (NF) (e.g. routers, load balancers, protocols, etc.) limits innovation, as it is non-trivial to replace an individual component within the system (Lopez, Caraguary, Villalba, & Lopez, 2015). This introduces complexities for organizations as they move toward dynamic systems and agile methodologies.

To improve on these scenarios the notion of Virtual Network Functions (VNF) transitioned traditional network functions onto hypervisors. This allowed for dynamic provisioning and elastic scenarios. While these virtualized technologies addressed challenges related to deployment of network functions, these virtual functions are cloned after their physical monolithic predecessors (Jammal, Signh, Shami, Asal, & Li, 2014).

Modern software design decomposes monolithic systems into (1) reusable modules; (2) higher level abstractions; (3) and removes shared responsibility by decoupling components. Software Defined Networking (SDN) is an approach to bring these patterns and practices to NF and VNF topologies.

SDN exposes a clear separation of duty between (1) the forwarding plane; (2) the network state; and (3) the control plane (Eissa, Bozed, & Younis, 2019). This enables a network application (e.g. new multi-cast protocols or security filters) to be added at the appropriate level without rebuilding the entire stack. If new features require 100s of lines of code versus 100s of 1000s, then it (1) promotes innovation; (2) gives rise to the broad adoption of programable networking; and (3) improves efficiency through highly dynamic real time configurations.

# Literature Review

## Trends on Virtualization with SDN and NFV (2014)

Active networks are not a novel idea and have been around for some time. However, their configuration has predominately relied on vendor specific proprietary languages. Within an enterprise environment it is common place to leverage heterogenous network vendors. This has introduced challenges as multiple implementations are needed to fully address shared scenarios.

Introducing new protocols and networking technologies can require significant delays between formation to implementation. These delays represent risk to projects and prolong the time until benefits from the decisions are received.

Server virtualization has gained mainstream adoption and is readily available across many enterprise environments. Just as the time to provision a virtual server has declined from weeks to minutes, so should the time required to provision a virtual network function. This transition from physical to virtual network functions has improved environment agility and reduced CAPEX/OPEX costs.

Improvements have also come through technologies such as Domain Name Services (DNS) and Virtual Local Area Network (VLAN) tagging. However, these only partially solve the address as it can be difficult to ‘lift and shift’ legacy systems without disruptions. The physical network also introduces design requirements that the virtual environment must adhere to. This is visible in scenarios such as overlapping address ranges and the mixture of certain protocols (e.g. IPv4 and IPv6).

These virtualized functions are blocked from innovation as it is difficult to replace thick layers. The remainder of the article describes how SDN breaks these barriers by decomposing thick layers into dedicated and replaceable components.

## Software Defined Networking (2014)

The authors explain that SDN expose abstractions of (1) the forwarding plane; (2) the network state; and (3) the control plane. This allows for each subsystem to be decoupled and therefore independently extended or replaced. Traditional systems were clunky all or nothing. Within an SDN there are distinct multiple layers, specifically (top to bottom):

* Network Applications: Firewalls, load balancers, security services, etc.
* Programming Languages: High Level Languages (e.g. Java or Python) can customize the configuration of the control plane.
* Northbound Interface: Abstract the programming interface to the controller (avoiding vendor specific code)
* Controller: Configurable system that expresses paths and policy decisions
* Network Hypervisor: Layer for hosting virtualized networking devices (think device drivers).
* Southbound Interface: Abstraction between the controller’s policy and the network devices implementation.
* Network Infrastructure: The physical networking devices.

## Software Defined Networks: State of the Art and Research Challenges (2014)

The article begins with an overview of the benefits from using SDN and OpenFlow. This is followed with additional details of the architecture like Eissa's description, and not repeated here.

Next definitions are provided for the entities involved in traffic management.

* A *flow* is a partition of network traffic.
* An *OpenFlow Switch* holds one or more `flow tables` and `group tables` to associate which actions need to be performed.
* An *OpenFlow Channel* is an interface between the switch and controller for transferring policy.
* An *OpenFlow Controller* maintains all protocols and policy information.
* The *OpenFlow Protocol (OFP)* are the algorithms used by the switch to make forwarding decisions based on the controller policy.

They also decompose network virtualization into multiple distinct responsibilities.

* Infrastructure Provider
* Virtual Network Provider
* Virtual Network Operator
* Service Provider
* Virtual Network User/End User

This is followed with a breakdown of concrete advantages of NVF and SDN.

* Multi-Tenant and Higher Utilization
* Encouraging Innovation and Improve Agility
* Network Segmentation and Security Isolation
* Traffic Shaping and Ensuring QoS
* Elastic and dynmamic networking configuration (e.g. Networking as a Service NaaS).
* Reduce costs and "go green"
* Increase transparency through consistent metrics

## Software Defined Networking with OpenFlow (2013)

In this e-book, Azodolmolky covers the implementation of SDN through OpenFlow. he continues with Python based examples for implementing different aspects of the SDN system.

### Chapter 1: Introducing OpenFlow.

The separation between policy and forwarding devices is implemented as `FlowTables` (FT). Each entry within the table maps `matching criteria` to zero or more `actions` that need be performed on the `flow`. These actions are limited to the developers imagination, such as conditionally dropping or duplicating the traffic. Performance metrics are also exposed and can be used for quota based systems and similar multi-tenant scenarios.

Pure OpenFlow compatible devices do not perform onboard controls and rely completely on the `controller` to provide forwarding rules. The controller can be thought of as the `Network Operating System`, and like a computer's operating system -- makes all the decisions around access and resource sharing.

The chapter concludes with a multi-page enumeration of the various features that are expressed within the OpenFlow messages. For instance, messages can be passed to add or remove routes. More complicated policies can be expressed such as `barriers` and `queuing` to enforce order of operations.

### Chapter 2: Implementing the OpenFlow Switch

Many vendors have created OpenFlow compatible switches as their is minimal complexity. Because the logic is contained within the controller, it is possible to use simple ASIC hardware based systems. This provides very fast implementations that are cheap to maintain.

`Mininet` is a software-based solution that can be used to simulate large scale software defined networks using minimal hardware (e.g. single laptop). A tutorial then followed for using `Wireshark` and `Mininet` to simulate and record traffic between 3 hosts.

### Chapter 3: The OpenFlow Controller

The controller implements (1) the interface to interact with network switches; and (2) provide the programmable API for network applications.

Under a `reactive control model`, a OpenFlow Switch needs to contact the controller each time the flow begins. The controller will then reply with the desired policy and the converation will continue for that session. `Proactive control models` instead broadcast the policy to each of the NF which must locally cache them. This gives a tradeoff between `centralized versus decentralized` design.

Note: Traditional networks are `packet switching networks` which means that an individual packet is self-contained. OpenFlow based systems operate on a higher construct called Flows. Since a flow spans multiple packets this results in easier management of the session. It is also possible to group multiple related flows into `aggregate flows`, such as (1) all traffic between two physical machines; or (2) all web server traffic.

The chapter concludes with example controllers such as NOX (C++), POX (Python), NodeFlow (JavaScript), Floodlight (Java), and OpenDayLight.

## POX (2017)

“POX started life as an OpenFlow controller, but can now also function as an OpenFlow switch, and be useful for writing networking software in general (Murphy, 2017).” The Python based source code is freely available on GitHub and contains a respectable level of comments. By reviewing an example implementation, it becomes possible to fill in the missing gaps of the literature.

Users of this controller create the service process and pass a list of third-party modules. A module is responsible for providing the custom behaviors of the system (e.g. transforming a packet or auditing an action). Each module must register for the events they wish to receive and implement a simple callback interface.

As events occur within the controller’s context, it will query for event subscribers and invoke their callback operations. If the event has multiple subscriptions, they are (1) called in a priority order and (2) passed the same reference mutable data structure. This enables scenarios where one module manipulates the source information and another the destinations.

## Mininet Walkthrough (2019)

Researchers after need the ability to simulate network topologies in a consistent and repeatable manner. Across the literature review multiple resources strongly encouraged the use of *mininet*. According to their product documentation, it leverages Linux Network Namespaces as a mechanism to segment a local process into a virtual network device.