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. In addition to virtualizing the network functions businesses have gained agility through technologies such as Domain Name Services (DNS) and Virtual Local Area Network (VLAN) tagging.

However, these solutions only partially address the underlying issues and can be difficult to ‘lift and shift’ without causing service disruptions. The physical network also introduces design requirements that the virtual environment must adhere to. This is visible in scenarios such as (1) overlapping address ranges; (2) mixture of certain protocols (e.g. IPv4 and IPv6); and (3) multi-tenant isolation.

Consider the scenario where Contoso transitions their private infrastructure to a public cloud. It is likely that certain assumptions will be made in the device’s network configuration, such as the static address of internal DNS servers. If these values need to be configuration values need to be updated during the live migration that introduces downtime risk.

It can be difficult to innovate on virtualized functions, as they often require replacement of a think layer. For instance, adding a security filter requires invasive changes to both the control and packet forwarding planes. These innovation barriers can be broken by decomposing each aspect of the system into dedicated and replaceable components.

## Software Defined Networking (2014)

Software Defined Networking expose abstractions for the (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.

An administrator needs to be able to state, ‘a malicious user is blocked after N operations in M seconds.’ They should be able to state this once in their desired language and let the system hide any differences between the manufactures. Within each plane there are multiple layers to provide the appropriate abstractions for the specific use cases. These layers include:

* Network Applications: Firewalls, load balancers, security services, etc.
* Programming Languages: High-level languages (e.g. Java or Python) that can respond to events and interact with the control plane’s configuration.
* Northbound Interface: Abstracts the programming interface with the controller.
* Controller: The configurable system which enforces policy and holds the holistic state of the network.
* Network Hypervisor: Enables multiple virtualized networking devices to reside on the same physical hardware.
* Southbound Interface: Abstraction between the controller’s policy and the network devices implementation.
* Network Infrastructure: The device drivers that are responsible for receiving and forwarding packets.

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

The de facto protocol used by software defined networks is called OpenFlow (OFP). Within OFP network traffic is partitioned into *flows*. A flow can have an arbitrary level of granularity such as a simple web request or all messages destined for a database cluster.

An *OpenFlow Switch* has one or more *flow tables* with each entry in the table mapping *match criteria* with zero or more *actions.* A group table can also exist to associate match criteria with a collection of related flows. The *OpenFlow Channel* is responsible for distributing policy notifications from the *OpenFlow* *Controller* to the *OpenFlow Switch*. This distribution takes place across the *OpenFlow Protocol* (OFP), which standardizes the messages used to describe the policy operations.

The authors decomposed network virtualization into clear separation of responsibilities. Their list of layers included: Infrastructure Provider; Virtual Network Provider; Virtual Network Operator; Service Provider; and Virtual Network User/End User. The literatures accounting appeared to be the most detailed assessment that was discovered.

The paper concludes with a breakdown of concrete advantages gained through the adoption of NVF and SDN. Some of the high-level categories included (1) multi-tenant scenarios and increased capacity utilization; (2) encouraging innovation and improved agility; (3) network segmentation and security isolation; (4) traffic shaping and enforced Quality of Service (QoS); (5) Networking as a Service (NaaS) and related elastic scenarios; (6) improved cost management; and (7) increased transparency through consistent metrics.

## Software Defined Networking with OpenFlow (2013)

While OpenFlow is an important aspect of software defined networking, the concrete implement has only been described in abstract. Azodolmolky’s e-book fills those gaps with scenario specific examples written in Python.

### 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 can execute any arbitrary code as a mechanism to perform a virtually unlimited number of scenarios.

For instance, the action can duplicate traffic to create a multi-casting system or leverage the built-in flow metrics for resource quotas. A novel idea for handling out-of-order events was proposed using barrier and queue constructs. Since the configuration is dynamic, actions can be registered to send from the queue only after the parent record is received. This can remove certain error conditions within service to service communication.

The agreed with Jammal et. al that pure OpenFlow compatible devices to not perform onboard controls and must rely solely on the controller. An analogy exists between the controller being the network operating system, like the kernel of a computer operating system.

### Chapter 2: Implementing the OpenFlow Switch

Due to the complexity residing within the controller, it is relatively straight forward to build Application-Specific Integrated Circuits (ASIC) that perform OpenFlow switching. These devices tend to be relatively cheap and provide hardware level performance. This has led to many vendors creating OpenFlow compatible solutions.

For many scenarios it is easier for students and researchers to run software defined networks within software. One of the most commonly used simulators is called Mininet. It can run large topologies using only the local hardware available on commodity laptops. This is accomplished through Linux’s Network Namespaces, as a mechanism to mimic the behavior of virtual device drivers.

The chapter concluded with a reprint of the Mininet Walkthrough Tutorial, that asks the reader to configure three simulated hosts and packet sniffing through Wireshark. After completing the guide, it should be clear to the reader how Mininet topologies work and how to configure them in more advanced scenarios.

### 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.

There are two policy delivery models denoted as the ‘reactive’ and ‘proactive’ control models. Under a *reactive* control model, an Open Switch needs to contact the controller at the beginning of a flow. The controller will reply with the desired policy and the switch can continue forwarding messages. *Proactive* control models broadcast the policy to each network function and relies on local caching. This gives a tradeoff between centralized versus decentralized design.

Depending on the granularity of the flow there can be significant performance penalties. However, unlike traditional packet switching, OpenFlow operates on a higher construct of flows. Like Jammal et. al description, flow entries can be grouped into aggregates for additional management simplicity. For instance, a policy might state ‘that all web servers can communicate with the backend data store.’ Therefore, if the OpenFlow Switch is managing 10 or 1000 web servers, the difference in this regard is negligible.

The chapter concluded with example comparisons between open source controllers NOX (C++), POX (Python), NodeFlow (JavaScript), and Floodlight (Java).

## 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.