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 resource provisioning and elastic scenarios. While these virtualized technologies addressed challenges related to the deployment of network functions, they 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 network capabilities (e.g. new multicast protocols or security filters) to be added at the appropriate level without rewriting the entire stack. If new features only require 100s of lines of code then it (1) promotes innovation; (2) makes broad adoption of programmable networking economical; and (3) improves efficiency through highly dynamic real-time configurations updates.

# Literature Review

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

Active networks are not a novel idea and have existed for some time. However, their configuration has predominately relied on vendor-specific proprietary languages. Within an enterprise environment, it is commonplace to leverage heterogeneous 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 a 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 instead of physical 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. Businesses have also gained agility and utilization density 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 can also introduce 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) implementing multi-tenant isolation.

Consider the scenario where Contoso transitions its 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 updated during the live migration that introduces downtime risk.

It can be difficult to innovate on virtualized functions, as changes can touch thick monolithic services. For instance, adding a security filter could require 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. Afterwards, it might be possible to only modify the control plane.

## Software Defined Networking (2014)

Software Defined Networking exposes 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 might define a policy that ‘*malicious users are 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 different 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 to the controller.
* Controller: The Network Operating System (NOS) enforces policy and manages 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 handle 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 to 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 paper concludes with a breakdown of concrete advantages gained through the adoption of NVF and SDN. These benefits span (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 the 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 multicasting solution or leverage the built-in flow metrics for resource quotas. A novel idea for handling out-of-order events was proposed using a barrier and queue construct. When the out-of-order event arrives, the system can (1) divert it into a queue; (2) register an action to dequeue when the dependent action is present. This can remove certain error conditions within service to service communication.

### Chapter 2: Implementing the OpenFlow Switch

Due to the complexity residing within the controller not the switch, it is straight forward to build Application-Specific Integrated Circuits (ASIC) that perform OpenFlow switching. These devices tend to be relatively cheap and provide hardware performance. This has encouraged many vendors to produce OpenFlow compatible devices.

For many scenarios, it is easier for students and researchers to run software defined networks with simple simulators. 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. The guide is detailed enough to provide the reader with confidence to implement more advanced scenarios afterwards.

### Chapter 3: The OpenFlow Controller

The controller implements (1) the interface to interact with the network switches; and (2) expose 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 the 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 rely 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 called flows. Flows 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 policy size 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 the Flow Tables 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 a reference to the same mutable data structure. This enables scenarios where one module manipulates the source information and another the destinations.

## Software-Defined Networking: A Comprehensive Survey (2014)

Several IEEE members contributed to the ‘Comprehensive Survey,’ an extensive document that examines every aspect of software defined networking. It should be required reading as it covers (1) the historical sequence of events that lead to the necessity of SDN; (2) decomposition of the infrastructure into layers; (3) describes the implementation variations of each layer; and (4) provides a roadmap for future research.

Most of the content can be obtained through the previously discussed literatures. However, this survey contains significantly more depth in each area. Another important feature of this effort is the categorization of 579 related publications. This enables readers to find very specialized materials for any aspect of software-defined networking.

One of the analogies that resonated was a comparison between networking and computer memory. Modern software does not generally concern itself about memory allocations, it calls into the operating system and then a continuous block appears. That virtual address space might reside across noncontinuous physical pages, but that is an irrelevant detail to the application.

To meet the needs of modern micro-services, the same capability is required from the network. For instance, a video streaming service should be able to reserve bandwidth capacity from the Network Operating System (NOS). It is solely the responsibility of the NOS to find the relevant resources and present them as a logical view. Perhaps this requires setting up forwarding devices in multiple datacenters. Despite the arbitrary level of complexity, these scenarios need to be challenges for the NOS not the video service.

# Conclusions

Modern systems need to be distributed, fault tolerant, and highly adaptive to changes of incoming request volumes. To efficiently enable these scenarios systems need to exist to allow for extensibility of the underlying networking infrastructure.

Previous efforts have attempted to virtualized traditional network functions, which improved some aspects of the system design. For instance, the time to purchase and configure a physical switch might be measured in weeks. In contrast, a virtual switch can be provisioned in minutes. While it is possible to easily create *copies* of these network functions, it is difficult to make *revisions*. Consider the challenges of introducing new protocols or security enhancements to an environment, due to the developers needing to modify thick layers of functionality.

Software-defined networking addresses these challenges by exposing abstractions and clear separation of duties. As seen in the POX reference implementation, entire avenues of innovation are now possible to businesses of all sizes. If existing hardware switches are insufficient, then businesses can even produce ASIC implementations. These capabilities were previously limited to only the largest enterprises.

These innovations should be expressible with general purpose languages, such as Java and Python. This further lowers the barrier to entry and allows development teams to incorporate internal systems within flow decisions. Perhaps a legacy system does not handle out of order events, the development team could introduce flow barriers and queue the messages until its dependencies are processed. It is unlikely that specialized business logic could be expressed through a generic product. Yet, writing a few hundred lines of glue code to integrate existing product libraries could be relatively trivial.