Table of contents

Introduction

In this lecture, we learn about Software Defined Networking (SDN). The need to separate the control plane from the data plane, coupled with increasing challenges that networks have been facing gradually led to the development of the SDN technology. We start with a brief overview of the stages that took place and eventually led to the development of the SDN technology. We learn about the architecture of the SDN controllers and we look into some example controllers. Finally, we learn about OpenDaylight, a popular and open source project for network programmability.

In the second lecture, we focus on data plane programmability, the P4 language (a high-level language for protocol independent packet processors). Also, we look at a P4 example application. The SDN paradigm has been used across multiple applications such as traffic engineering, security, and data center network applications. In this lecture, we look at how an SDN approach would work within an Internet Exchange Point.

Readings and Additional Resources

Readings:

- The Road to SDN: An Intellectual History of Programmable Networks
- Kurose-Ross, 7th Edition. Section 4.1.1 , Chapter 5.
- Software-Defined Networking: A Comprehensive Survey (Links to an external site.)
- ONOS: Towards an Open, Distributed SDN OS (Links to an external site.)
- P4: Programming Protocol-Independent Packet Processors (Links to an external site.)
- A Software Defined Internet Exchange

Optional Readings:

- SDN Controllers: Benchmarking & Performance Evaluation (Links to an external site.)
- SDN architecture

- P4 Language tutorial (Links to an external site.)
- An Industrial-Scale Software Defined Internet Exchange Point

What led us to SDN?

Software Defined Networking (SDN) arose as part of the process to make computer networks more programmable. Computer networks are very complex and especially difficult to manage for two main reasons:

- Diversity of equipment on the network
- Proprietary technologies for the equipment

Diversity of equipment

Computer networks have a wide range of equipment - from routers and switches to middleboxes such as firewalls, network address translators, server load balancers, and intrusion detection systems (IDSs). The network has to handle different software adhering to different protocols for each of these equipment. Even with a network management tool offering a central point of access, they still have to operate at a level of individual protocols, mechanisms and configuration interfaces, making network management very complex.

Proprietary Technologies

Equipment like routers and switches tend to run software that is closed and proprietary. This means that configuration interfaces vary between vendors. In fact, these interfaces could also differ between different products offered by the same vendor! This makes it harder for the network to manage all these devices centrally.

These characteristics of computer networks made them highly complex, slow to innovate, and drove up the costs of running a network. SDN offers new ways to redesign networks to make them more manageable! It employs a simple idea: separation of tasks. We've seen that our code becomes more modular and easy to manage when we divide it into smaller functions with focused tasks. Similarly, SDN divides the network into two planes: the control plane and the data plane. It uses this separation to simplify management and speed up innovation!

A Brief History of SDN: The Milestones

In this section, we provide an overview of the history of SDN, as a summary based on the paper "The Road to SDN: An Intellectual History of Programmable Networks"

The history of SDN can be divided into three phases:

1. **Active networks**
2. **Control and data plane separation**
3. **OpenFlow API and network operating systems**

Let's take a look at each phase.

1. **Active networks**

This phase took place from the mid-1990s to the early 2000s. During this time, the internet takeoff resulted in an increase in the applications and appeal of the internet. Researchers were keen on testing new ideas to improve network services. However, this required standardization of new protocols by the IETF (Internet Engineering Task Force), which was a slow and frustrating process. This tediousness led to the growth of active networks, which aimed at opening up network control. Active networking envisioned a programming interface (a network API) that exposed resources/network nodes and supported customization of functionalities for subsets of packets passing through the network nodes. This was the opposite of the popular belief in the internet community: simplicity of the network core was important to the success of the internet!

In the early 1990s, the networking approach was primarily via IP or ATM (Asynchronous Transfer Mode). Active networking became one of the first 'clean slate' approaches to network architecture. There were two types of programming models in active networking. These models differ based on where the code to execute at the nodes was carried.

- **Capsule model:** carried in-band in data packets
- **Programmable router/switch model:** established by out-of-band mechanisms.

Although the capsule model was most closely related to active networking, both models had some effect on the current state of SDNs. By carrying the code in data packets, capsules brought a new data-plane functionality across networks. They also used caching to make code distribution more efficient. Programmable routers made decision making a job for the network operator.

Technology push: The pushes that encouraged active networking were

- Reduction in computation cost. This enabled us to put more processing into the network.
- Advancement in programming languages. For languages like Java, the options of platform portability, code execution safety, and VM (virtual machine) technology to protect the active node in case of misbehaving programs.
- Advances in rapid code compilation and formal methods.
- Funding from agencies such as DARPA (U.S. Defense Advanced Research Projects Agency) for a collection promoted interoperability among projects. This was especially beneficial because there were no short-term use cases to alleviate the skepticism people had about the use of active networking.

Use pull: The use pulls for active networking were

- Network service provider frustration concerning the long timeline to develop and deploy new network services.
- Third-party interests to add value by implementing control at a more individualistic nature. This meant dynamically meeting the needs of specific applications or network conditions.
- Researchers' interest in having a network that would support large-scale experimentation.
- Unified control over middleboxes. We discussed the disadvantage of having diverse programming models that varied not only based on the type of middlebox (for example, firewalls, proxies, etc) but based on the vendor. Active networking envisioned unified control that could replace individually managing these boxes. This actually foreshadows the trends we see now in network functions virtualization – where we also attempt to provide a central unifying framework for networks with complex middlebox functions.

It is interesting to note that the use pulls for active networks in the mid-1990s are similar to those for SDN now! In addition to these use cases, active networks made three major contributions related to SDN:

- Programmable functions in the network to lower the barrier to innovation. Active networks were one of the first to introduce the idea of using programmable networks to overcome the slow speed of innovation in computer networking. While many early visions for SDN concentrated on increasing programmability of the control plane, active networks focused on the programmability of the data plane. This has continued to develop independently. Recently, this data-plane programmability has been gaining more traction due to the emerging network function virtualization (NFV) initiatives. In addition, the concept of isolating experimental traffic from normal traffic had emerged from active networking and is heavily used in OpenFlow and other SDN technologies.
- Network virtualization, and the ability to demultiplex to software programs based on packet headers. Active networking produced a framework that described a platform that would support experimentation with different programming models. This was the need that led to network virtualization.
- The vision of a unified architecture for middlebox orchestration. The last use-pull for SDN, i.e., unified control over middleboxes, was never fully realized in the era of active networking. While it did not directly influence NFV, some lessons from its research are useful while trying to implement a unified architecture now!

One of the biggest downfalls for active networking was that it was too ambitious! Since it required end users to write Java code, it was too far removed from the reality at that time, and hence was not trusted to be safe. Since active networking was more involved in redesigning the architecture of networks, not as much

emphasis was given to performance and security, which users were more concerned about. However, it is worthwhile to note that some efforts did aim to build high-performance active routers, and there were a few notable projects that did address the security of networks. Since there were no specific short-term problems that active networks solved, it was hard for them to see widespread deployment. The next efforts had a more focused scope and distinguished between control and data planes. This difference made it easier to focus on innovation in a specific plane and inflict widespread change.

2. **Control and data plane separation**

This phase lasted from around 2001 to 2007. During this time, there was a steady increase in traffic volumes and thus, network reliability, predictability and performance became more important. Network operators were looking for better network-management functions such as control over paths to deliver traffic (traffic engineering). Researchers started exploring short-term approaches that were deployable using existing protocols. They identified the challenge in network management lay in the way existing routers and switches tightly integrated the control and data planes. Once this was identified, efforts to separate the two began.

Technology push: The technology pushes that encouraged control and data plane separation were

- Higher link speeds in backbone networks led vendors to implement packet forwarding directly in the hardware, thus separating it from the control-plane software.
- Internet Service Providers (ISPs) found it hard to meet the increasing demands for greater reliability and new services (such as virtual private networks), and struggled to manage the increased size and scope of their networks.
- Servers had substantially more memory and processing resources than those deployed one-to-two years prior. This meant that a single server could store all routing states and compute all routing decisions for a large ISP network. This also enabled simple backup replication strategies, thus ensuring controller reliability.
- Open-source routing software lowered the barrier to creating prototype implementations of centralized routing controllers.

These pushes inspired two main innovations:

- Open interface between control and data planes
- Logically centralized control of the network

This phase was different from active networking in several ways:

- It focused on spurring innovation by and for network administrators rather than end users and researchers.
- It emphasized programmability in the control domain rather than the data domain.
- It worked towards network-wide visibility and control rather than device-level configurations.

Use pulls: Some use pulls for the separation of control and data planes were

- Selecting between network paths based on the current traffic load
- Minimizing disruptions during planned routing changes
- Redirecting/dropping suspected attack traffic
- Allowing customer networks more control over traffic flow
- Offering value-added services for virtual private network customers

Most work during this phase tried to manage routing within a single ISP, but there were some proposals about ways to enable flexible route control across many administrative domains. The attempt to separate the control and data planes resulted in a couple of concepts that were used in further SDN design:

- Logically centralized control using an open interface to the data plane.
- Distributed state management.

Initially, many people thought separating the control and data planes was a bad idea, since there was no clear idea as to how these networks would operate if a controller failed. There was also some skepticism about moving away from a simple network where all have a common view of the network state to one where the router only had a local view of the outcome of route-selection. However, this concept of separation of planes

helped researchers think clearly about distributed state management. Several projects exploring a clean-slate architecture commenced and laid the foundation for OpenFlow API.

3. **OpenFlow API and network operating systems**

This phase took place from around 2007 to 2010. OpenFlow was born out of the interest in the idea of network experimentation at scale (by researchers and funding agencies). It was able to balance the vision of fully programmable networks and the practicality of ensuring real-world deployment. OpenFlow built on the existing hardware and enabled more functions than earlier route controllers. Although this dependency on hardware limited its flexibility, it enabled immediate deployment.

The basic working of an OpenFlow switch is as follows. Each switch contains a table of packet-handling rules. Each rule has a pattern, list of actions, set of counters and a priority. When an OpenFlow switch receives a packet, it determines the highest priority matching rule, performs the action associated with it and increments the counter.

Technology push: OpenFlow was adopted in industry, unlike its predecessors. This could be due to:

- Before OpenFlow, switch chipset vendors had already started to allow programmers to control some forwarding behaviors.
- This allowed more companies to build switches without having to design and fabricate their own data plane.
- Early OpenFlow versions built on technology that the switches already supported. This meant that enabling OpenFlow initially was as simple as performing a firmware upgrade!

Use pulls:

- OpenFlow was started to meet the need of conducting large-scale experimentation on network architectures. In the late 2000s, OpenFlow testbeds were deployed across many college campuses to show its capability on single-campus networks and wide area backbone networks over multiple campuses.
- OpenFlow was useful in data-center networks – there was a need to manage network traffic at large scales.
- Companies started investing more in programmers to write control programs, and less in proprietary switches that could not support new features easily.
- This allowed many smaller players to become competitive in the market by supporting capabilities like OpenFlow.

Some key effects that OpenFlow had were:

- Generalizing network devices and functions.
- The vision of a network operating system.
- Distributed state management techniques.

Why Separate the Data Plane from the Control Plane?

Why separate the control plane from the data plane? We know that SDN differs from traditional approaches by separating the control and data planes. The control plane contains the logic that controls the forwarding behavior of routers such as routing protocols and network middlebox configurations. The data plane performs the actual forwarding as dictated by the control plane. For example, IP forwarding and Layer-2 switching are functions of the data plane. The reasons we separate the two are:

1. **Independent evolution and development**

In the traditional approach, routers are responsible for both routing and forwarding functionalities. This meant that a change to either of the functions would require an upgrade of hardware. In this new approach, routers only focus on forwarding. Thus, innovation in this design can proceed independently of other routing considerations. Similarly, improvement in routing algorithms can take place without affecting any of the existing routers. By limiting the interplay between these two functions, we can develop them more easily.

2. **Control from a high-level software program**

In SDN, we use software to compute the forwarding tables. Thus, we can easily use higher-order programs to control the routers' behavior. The decoupling of functions makes debugging and checking the behavior of the network easier. Separation of the control and data planes supports the independent evolution and development of both. Thus, the software aspect of the network can evolve independent of the hardware aspect. Since both control and forwarding behavior are separate, this enables us to use higher-level software programs for control. This makes it easier to debug and check the network's behavior.

In addition, this separation leads to opportunities in different areas:

- **Data centers.** Consider large data centers with thousands of servers and VMs. Management of such large network is not easy. SDN helps to make network management easier.
- **Routing.** The interdomain routing protocol used today, BGP, constrains routes. There are limited controls over inbound and outbound traffic. There is a set procedure that needs to be followed for route selection. Additionally, it is hard to make routing decisions using multiple criteria. With SDN, it is easier to update the router's state, and SDN can provide more control over path selection.
- **Enterprise networks.** SDN can improve the security applications for enterprise networks. For example, using SDN it is easier to protect a network from volumetric attacks such as DDoS, if we drop the attack traffic at strategic locations of the network.
- **Research networks.** SDN allows research networks to coexist with production networks.

Control Plane and Data Plane Separation

Two important functions of the network layer are:

1. **Forwarding**

Forwarding is one of the most common, yet important functions of the network layer. When a router receives a packet at its input link, it must determine which output link that packet should be sent through. This process is called forwarding. It could also entail blocking a packet from exiting the router, if it is suspected to have been sent by a malicious router. It could also duplicate the packet and send it along multiple output links. Since forwarding is a local function for routers, it usually takes place in nanoseconds and is implemented in the hardware itself. Forwarding is a function of the data plane. So, a router looks at the header of an incoming packet and consults the forwarding table, to determine the outgoing link to send the packet to.

2. **Routing**

Routing involves determining the path from the sender to the receiver across the network. Routers rely on routing algorithms for this purpose. It is an end-to-end process for networks. It usually takes place in seconds and is implemented in software. Routing is a function of the control plane. In the traditional approach, the routing algorithms (control plane) and forwarding function (data plane) are closely coupled. The router runs and participates in the routing algorithms. From there it is able to construct the forwarding table, which it consults for the forwarding function.

In the SDN approach, on the other hand, there is a remote controller that computes and distributes the forwarding tables to be used by every router. This controller is physically separate from the router. It could be located in some remote data center, managed by the ISP or some other third party. We have a separation of the functionalities. The routers are solely responsible for forwarding, and the remote controllers are solely responsible for computing and distributing the forwarding tables. The controller is implemented in software, and therefore we say the network is software-defined. These software implementations are also increasingly open and publicly available, which speeds up innovation in the field.

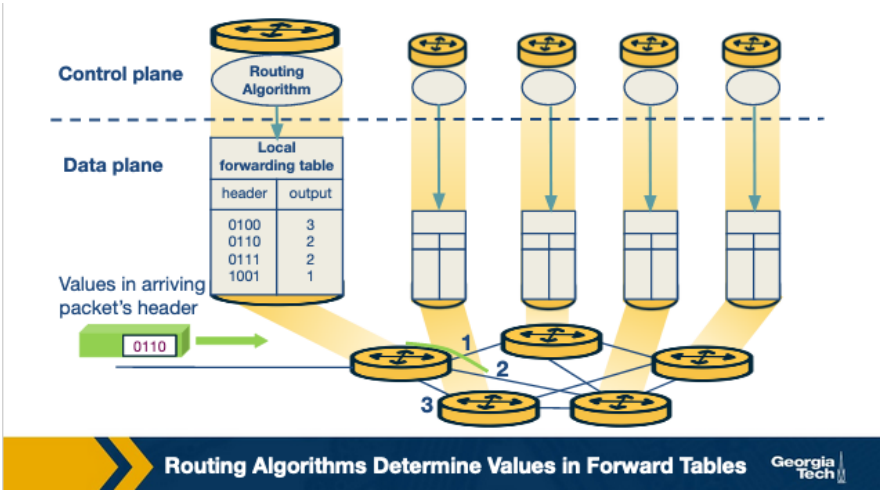

Figure 1: image

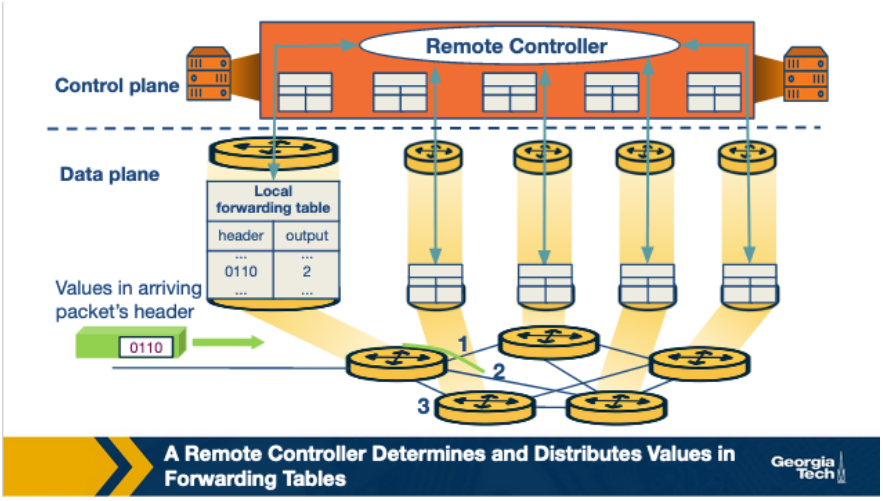

Figure 2: image

The SDN Architecture

In the figure below we see the main components of an SDN network:

- **SDN-controlled network elements**: The SDN-controlled network elements, sometimes called the infrastructure layer, is responsible for the forwarding of traffic in a network based on the rules computed by the SDN control plane.
- **SDN controller**: The SDN controller is a logically centralized entity that acts as an interface between the network elements and the network-control applications.
- **Network-control applications**: The network-control applications are programs that manage the underlying network by collecting information about the network elements with the help of SDN controller.

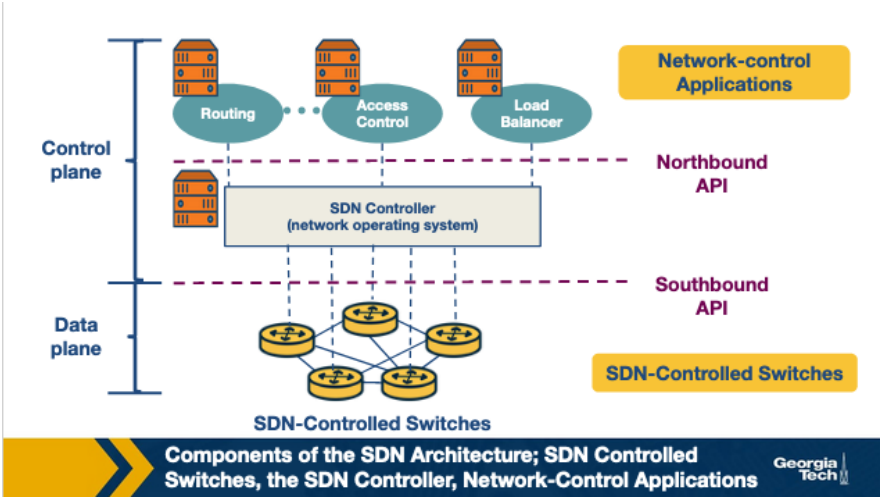

Figure 3: image

Let us now take a look at the four defining features in an SDN architecture:

1. **Flow-based forwarding:** The rules for forwarding packets in the SDN-controlled switches can be computed based on any number of header field values in various layers such as the transport layer, network layer and link layer. This differs from the traditional approach where only the destination IP address determines the forwarding of a packet. For example, OpenFlow allows up to 11 header field values to be considered.
2. **Separation of data plane and control plane:** The SDN-controlled switches operate on the data plane and they only execute the rules in the flow tables. Those rules are computed, installed, and managed by software that runs on separate servers.
3. **Network control functions:** The SDN control plane, (running on multiple servers for increased performance and availability) consists of two components: the controller and the network applications. The controller maintains up-to-date network state information about the network devices and elements (for example, hosts, switches, links) and provides it to the network-control applications. This information, in turn, is used by the applications to monitor and control the network devices.
4. **A programmable network:** The network-control applications act as the "brain" of the SDN control plane by managing the network. Example applications can include network management, traffic engineering, security, automation, analytics, etc. For example, we can have an application that determines the end-to-end path between sources and destinations in the network using Dijkstra's algorithm.

The SDN Controller Architecture

The SDN controller is a part of the SDN control plane and acts as an interface between the network elements and the network-control applications. An SDN controller can be broadly split into three layers:

- Communication layer: communicating between the controller and the network elements
- Network-wide state-management layer: stores information of network-state
- Interface to the network-control application layer: communicating between controller and applications

Let's look at each layer in detail starting from the bottom:

1. **Communication Layer:** This layer consists of a protocol through which the SDN controller and the network-controlled elements communicate. Using this protocol, the devices send locally observed events to the SDN controller providing the controller with a current view of the network state. For example, these events can be a new device joining the network, heartbeat indicating the device is up, etc. The communication between SDN controller and the controlled devices is known as the "southbound" interface. OpenFlow is an example of this protocol, which is broadly used by SDN controllers today.
2. **Network-wide state-management layer:** This layer is about the network state that is maintained by the controller. The network state includes any information about the state of the hosts, links, switches and other controlled elements in the network. It also includes copies of the flow tables of the switches. Network-state information is needed by the SDN control plane to configure the flow tables.
3. **The interface to the network-control application layer:** This layer is also known as the controller's "northbound" interface, which the SDN controller uses to interact with network-control applications. Network-control applications can read/write the network state and flow tables in the controller's state-management layer. The SDN controller can notify applications of changes in the network state, based on the event notifications sent by the SDN-controlled devices. The applications can then take appropriate actions based on the event. A REST interface is an example of a northbound API. The SDN controller, although viewed as a monolithic service by external devices and applications, is implemented by distributed servers to achieve fault tolerance, high availability and efficiency. Despite the issues of synchronization across servers, many modern controllers such as OpenDayLight and ONOS have solved it and prefer distributed controllers to provide highly scalable services.

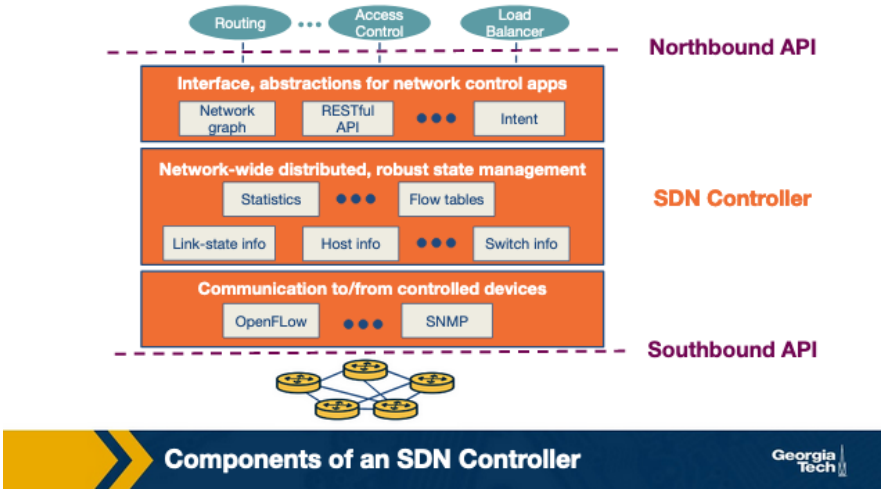

Figure 4: image

Revisiting the Motivation for SDN

In this topic, we are talking about the motivation that led to SDN. As IP networks grew in adoption worldwide, there were a few challenges that became more and more pronounced, such as:

1. Handling the ever growing complexity and dynamic nature of networks: The implementation of network policies required changes right down to each individual network device, which were often carried out by vendor-specific commands and required manual configurations. This was a heavy upkeep for operators. Traditional IP networks are quite far away from achieving automatic response mechanisms to dynamic network environment changes.
2. Tightly coupled architecture: The traditional IP networks consist of a control plane (handles network traffic) and a data plane (forwards traffic based on the control plane's decisions) that are bundled together. They are contained inside networking devices, and are thus not flexible to work on. This is evidenced by the fact that any new protocol update takes as long as 10 years, because of the way these changes need to percolate down to every networking device that is a part of the IP network.

Software Defined Networking. This networking paradigm is an attempt to overcome limitations of the legacy IP networking paradigm. It starts by separating out the control logic (in the control plane) from the data plane. With this separation, the network switches simply perform the task of forwarding, and the control logic is purely implemented in a logically centralized controller (or a network OS), thereby making it possible for innovation to occur in areas of network reconfiguration and policy enforcement. Despite the centralized nature of control logic, in practice, production-level SDNs need a physically distributed control plane to achieve performance, reliability and scalability.

The separation of control and data plane is achieved by using a programming interface between the SDN controller and the switches. The SDN controller controls the data plane elements via the API. An example of such an API is OpenFlow. A switch in OpenFlow has one or more tables for packet handling rules. Each rule matches a subset of network traffic and performs actions such as dropping, forwarding, modifying etc. An OpenFlow switch can be instructed by the controller to behave like a firewall, switch, router, or even perform other roles like load balancer, traffic shaper, etc.

As opposed to traditional IP networks, SDN principles allow for a separation of concerns introduced between the definition of networking policies, their implementation in hardware, and the forwarding of traffic. It is this separation that allows for networking control problems to be viewed as tractable pieces, allowing for newer networking abstractions and simplifying networking management, allowing innovation.

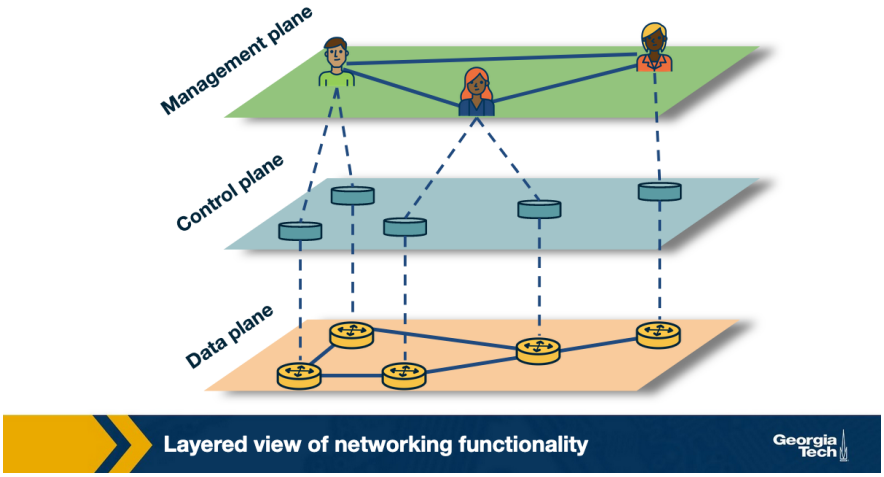

Figure 5: image

Traditionally viewed, computer networks have three planes of functionality, which are all abstract logical concepts:

Data plane: These are functions and processes that forward data in the form of packets or frames.

Control plane: These refer to functions and processes that determine which path to use by using protocols to populate forwarding tables of data plane elements.

Management plane: These are services that are used to monitor and configure the control functionality, e.g. SNMP-based tools.

In short, say if a network policy is defined in the management plane, the control plane enforces the policy and the data plane executes the policy by forwarding the data accordingly.

SDN Advantages

What are the main differences between the traditional approach (conventional networks) and the new SDN paradigm? Since the two approaches have contrasting differences, what are the advantages to using the SDN technology?

Conventional networks: We saw earlier that these networks come with a tightly coupled data and control plane, thereby making the networking components physically embedded. As a result, to add a new networking feature, one has to go through the process of modifying all control plane devices - e.g. installing new firmware / hardware upgrades. To avoid this, traditionally, a new specialized equipment was introduced (known as middlebox) through which concepts and features such as load balancers, intrusion detection systems, firewalls, etc. were introduced. Since these middleboxes are required to be carefully placed in the network topology, it is much harder to later change or reconfigure them.

Software-defined networks: Since SDN decouples the control plane from the physical networking devices, it isolates itself as an external entity (SDN controller). With this, middlebox services can be viewed as a SDN controller application. This approach has several advantages:

1. *Shared abstractions:* These middlebox services (or network functionalities) can be programmed easily now that the abstractions provided by the control platform and network programming languages can be shared.
2. *Consistency of same network information:* All network applications have the same global network information view, leading to consistent policy decisions while reusing control plane modules
3. *Locality of functionality placement:* Previously, the location of middleboxes was a strategic decision and big constraint. However, in this model, the middlebox applications can take actions from anywhere in the network.
4. *Simpler integration:* Integrations of networking applications are smoother. For example, load balancing and routing applications can be combined sequentially.

The SDN Landscape

In this topic we are looking at an overview of the SDN-landscape. The landscape of the SDN architecture can be decomposed into layers as shown in the figure below.

Each layer performs its own functions through different technologies. The figure above presents three perspectives of the SDN landscape: (a) a plane-oriented view, (b) the SDN layers, and (c) a system design perspective. Next, for each layer, we are providing an overview of the technologies that have been developed. Also, for some representative technologies we are referencing links to actively maintained tutorials.

1. **Infrastructure:** Similar to traditional networks, an SDN infrastructure consists of networking equipment (routers, switches and other middlebox hardware). What is now different is that these physical networking equipment are merely forwarding elements that do a simple forwarding task, and any logic to operate them is directed from the centralized control system. Popular examples of such infrastructure equipment include OpenFlow (software) switches such as SwitchLight, Open vSwitch, Pica8, etc. For further details

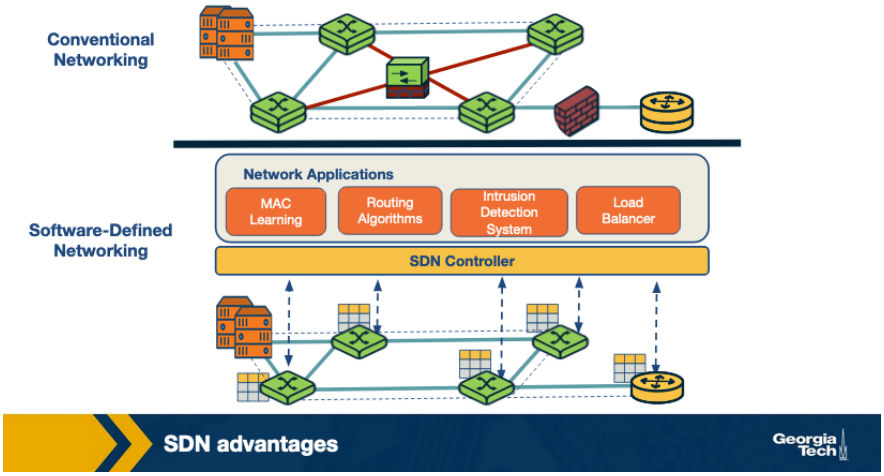

Figure 6: image

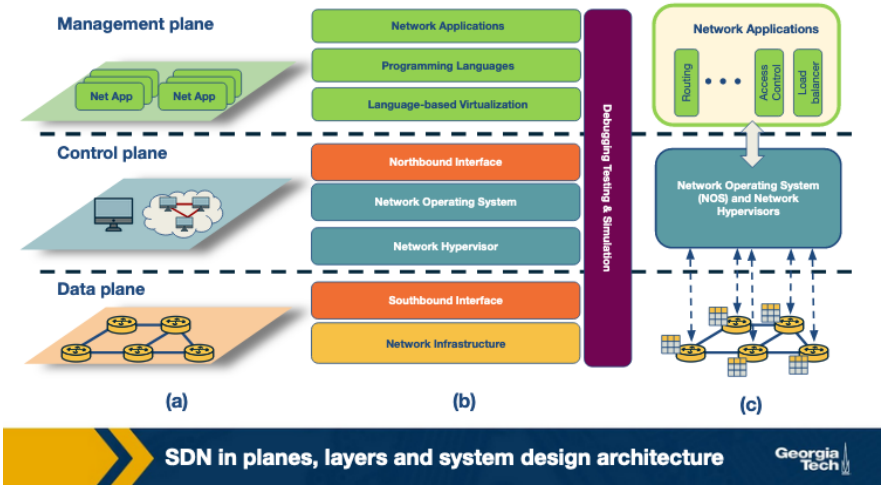

Figure 7: image

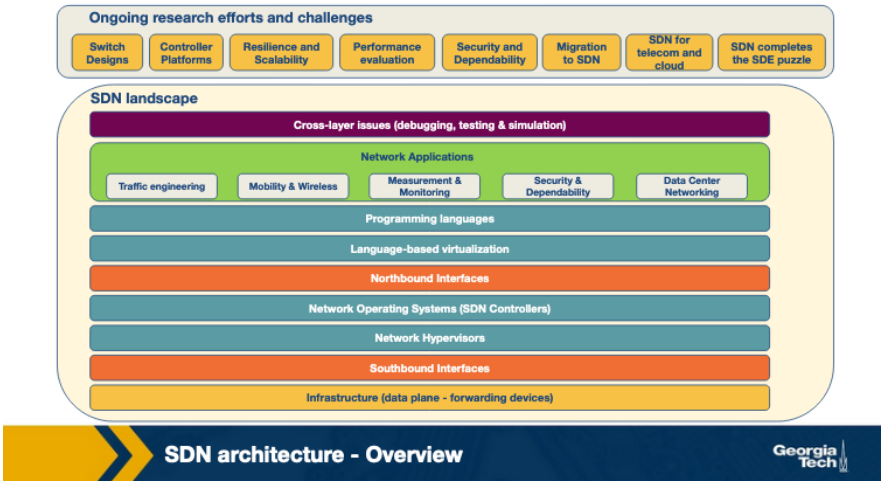

Figure 8: image

and hands on tutorial please look through these links: OpenFlow: https://github.com/mininet/openflow-tutorial/wiki (Links to an external site.)

2. **Southbound interfaces:** These are interfaces that act as connecting bridges between connecting and forwarding elements, and since they sit in between control and data plane, they play a crucial role in separating control and data plane functionality. These APIs are tightly coupled with the forwarding elements of the underlying physical or virtual infrastructure. The most popular implementation of Southbound APIs for SDNs is OpenFlow, however there are other APIs proposed such as ForCES, OVSDB, POF, OpFlex, OpenState, etc. For more reading and hands on for OVSDB: http://docs.openvswitch.org/en/latest/ref/ovsdb.7/ (Links to an external site.)
3. **Network virtualization:** For a complete virtualization of the network, the network infrastructure needs to provide support for arbitrary network topologies and addressing schemes, similar to the computing layer. Existing virtualization constructs such as VLAN, NAT and MLPS are able to provide full network virtualization, however these technologies are connected by a box-by-box basis configuration and there is no unifying abstraction that can be leveraged to configure these in a global manner, thereby making current network provisioning tasks as long as months and years. New advancements in SDN network virtualization such as VxLAN, NVGRE, FlowVisor, FlowN, NVP are promising.
4. **Network operating systems:** The promise of SDN is to ease network management and solve networking problems by using a logically centralized controller by way of a network operating system (NOS). The value of a NOS is in providing abstractions, essential services and common APIs to developers. For example, while programming a network policy, if a developer doesn't need to worry about low-level details about data distribution among routing elements, that is an abstraction. Such systems propel more innovation by reducing inherent complexity of creating new network protocols and network applications. Some popular NOSs are OpenDayLight, OpenContrail, Onix, Beacon and HP VAN SDN. For more details and tutorials for OpenDayLight, please follow this link: https://www.opendaylight.org/technical-community/getting-started-for-developers/tutorials (Links to an external site.)
5. **Northbound interfaces:** The two core abstractions of an SDN ecosystem are the Southbound and Northbound interfaces. We have already seen Southbound interfaces, and that it already has a widely acceptable norm (OpenFlow). Compared to that, a standard for Northbound interface is still an open problem, as are its use cases. What is relatively clear is that Northbound interfaces are supposed to be a mostly software ecosystem, as opposed to the Southbound interfaces. Another key requirement is the abstraction that guarantees programming language and controller independence. Some popular examples are Floodlight, Trema, NOX, Onix and SFNet. For a tutorial to get a more hands on experience on Floodlight: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343514/Tutorials (Links to an external site.)
6. **Language-based virtualization:** An important characteristic of virtualization is the ability to express modularity and allowing different levels of abstraction. For example, using virtualization we can view a single physical device in different ways. This takes away the complexity away from application developers without compromising on security which is inherently guaranteed. Some popular examples of programming languages that support virtualization are Pyretic, libNetVirt, AutoSlice, RadioVisor, OpenVirteX, etc.
7. **Network programming languages:** Network programmability can be achieved using low-level or high-level programming languages. Using low-level languages, it is difficult to write modular code, reuse it and it generally leads to more error-prone development. High level programming languages in SDNs provide abstractions, make development more modular, code more reusable in control plane, do away with device specific and low-level configurations, and generally allow faster development. Some examples of network programming languages in SDNs are Pyretic, Frenetic, Merlin, Nettle, Procera, FML, etc. For a tutorial on Frenetic programming language: http://frenetic-lang.github.io/tutorials/Introduction/ (Links to an external site.) Pyretic: https://github.com/frenetic-lang/pyretic/wiki (Links to an external site.)
8. **Network applications:** These are the functionalities that implement the control plane logic and translate to commands in the data plane. SDNs can be deployed on traditional networks, and can find itself in home area networks, data centers, IXPs etc. Due to this, there is a wide variety of network applications such as routing, load balancing, security enforcement, end-to-end QoS enforcement, power consumption reduction, network virtualization, mobility management, etc. Some well known solutions

are Hedera, Aster*x, OSP, OpenQoS, Pronto, Plug-N-Serve, SIMPLE, FAMS, FlowSense, OpenTCP, NetGraph, FortNOX, FlowNAC, VAVE, etc.

SDN Infrastructure Layer

In the previous topic, we talked about the landscape of an SDN infrastructure. In this topic we are zooming into the infrastructure layer. The SDN infrastructure composes of networking equipment (routers, switches and appliance hardware) performing simple forwarding tasks. The physical devices do not have embedded intelligence or control, as the network intelligence is now delegated to a logically centralized control system - the Network Operating System (NOS). An important difference in these networks is that they are built on top of open and standard interfaces that ensure configuration and communication compatibility and interoperability among different control plane and data plane devices. As opposed to traditional networks that use proprietary and closed interfaces, these networks are able to dynamically program heterogeneous network devices as forwarding devices.

In the SDN architecture, a data plane device is a hardware or software entity that forwards packets, while a controller is a software stack running on commodity hardware. A model derived from OpenFlow is currently the most widely accepted design of SDN data plane devices. It is based on a pipeline of flow tables where each entry of a flow table has three parts: a) a matching rule, b) actions to be executed on matching packets, and c) counters that keep statistics of matching packets. Other SDN-enabled forwarding device specifications include Protocol-Oblivious Forwarding (POF) and Negotiable Datapath Models (NDMs).

In an OpenFlow device, when a packet arrives, the lookup process starts in the first table and ends either with a match in one of the tables of the pipeline or with a miss (when no rule is found for that packet). Some possible actions for the packet include: - Forward the packet to outgoing port - Encapsulate the packet and forward it to controller - Drop the packet - Send the packet to normal processing pipeline - Send the packet to next flow table

SDN Southbound Interfaces

In the previous topic, we talked about the landscape of an SDN infrastructure. In this topic, we are zooming into the Southbound Interfaces. The Southbound interfaces or APIs are the separating medium between the control plane and data plane functionality.

From a legacy standpoint, development of a new switch typically takes up to two years for commercialization. Added to that is the upgrade cycles and time required to software development for the new product. Since the southbound APIs represent one of the major barriers for introduction and acceptance of any new networking technology, API proposals like OpenFlow have received good reception. These standards promote interoperability and deployment of vendor-agnostic devices. This has already been achieved by the OpenFlow-enabled equipments from different vendors.

Currently, OpenFlow is the most widely accepted southbound standard for SDNs. It provides specification to implement OpenFlow-enabled forwarding devices, and for the communication channel between data and control plane devices. There are three information sources provided by OpenFlow protocol:

1. Event-based messages that are sent by forwarding devices to controller when there is a link or port change
2. Flow statistics are generated by forwarding devices and collected by controller
3. Packet messages are sent by forwarding devices to controller when they do not know what to do with a new incoming flow

These three channels are key to provide flow-level info to the Network Operating System (NOS).

Despite OpenFlow being the most popular southbound interface for SDN, there are others API proposals such as ForCES, OVSDB, POF, OpFlex, OpenState, etc. In case of ForCES, it provides a more flexible approach

to traditional network management without changing the current architecture of the network, i.e, it does not need a logically centralized controller. The control and data planes are separated but potentially can also be kept in the same network element. OVSDB is another southbound API that acts complementary to OpenFlow or Open vSwitch. It allows the control elements to create multiple vSwitch instances, set QoS policies on interfaces, attach interfaces to the switches, configure tunnel interfaces on the OpenFlow data paths, manage queues and collect statistics.

SDN Controllers: Centralized vs Distributed

As we've seen earlier, the biggest drawback of traditional networks is that they are configured using low-level, device-specific instruction sets and run mostly proprietary network operating systems. This challenges the notion of device-agnostic developments and abstraction, which are key ideas to solve networking problems. SDN offers these by means of a logically centralized control. A controller is a critical element in an SDN architecture as it is the key supporting piece for control logic (applications) to generate network configuration based on the policies defined by the network operator. There is a broad range of architectural choices when it comes to controllers and control platforms.

Core controller functions: Some base network service functions are what we consider the essential functionality all controllers should provide. Functions such as topology, statistics, notifications, device management, along with shortest path forwarding and security mechanisms are essentials network control functionalities that network applications may use in building its logic. For example, security mechanisms are critical components to provide basic isolation and security enforcements between services and applications. For instance, high priority services' rules should always take precedence over rules created by applications with low priority.

SDN Controllers can be categorized based on many aspects. In this topic we will categorize them based on centralized or distributed architecture.

Centralized controllers: In this architecture, we typically see a single entity that manages all forwarding devices in the network, which is a single point of failure and may have scaling issues. Also, a single controller may not be enough to handle a large number of data plane elements. Some enterprise class networks and data centers use such architectures, such as Maestro, Beacon, NOX-MT. They use multi-threaded designs to explore parallelism of multi-core computer architectures. For example, Beacon can deal with more than 12 million flows per second by using large sized computing nodes of cloud providers. Other single controller architectures such as Trema, Ryu NOS, etc. target specific environments such as data centers, cloud infras. Controllers such as Rosemary offer specific functionality and guarantees security and isolation of applications by using a container based architecture called micro-NOS.

Distributed controllers: Unlike single controller architectures that cannot scale in practice, a distributed network operating system (controller) can be scaled to meet the requirements of potentially any environment - small or large networks. Distribution can occur in two ways: it can be a centralized cluster of nodes or physically distributed set of elements. Typically, a cloud provider that runs across multiple data centers interconnected by a WAN may require a hybrid approach to distribution - clusters of controllers inside each data center and distributed controller nodes in different sites. Properties of distributed controllers:

- Weak consistency semantics
- Fault tolerance

An example Controller: ONOS

As we saw in the previous topic there are different types of SDN controllers. In this topic, we are talking about an example distributed controller.

ONOS (Open Networking Operating System) is a distributed SDN control platform. It aims to provide a global view of the network to the applications, scale-out performance and fault tolerance. The prototype was built based on Floodlight, an open-source single-instance SDN controller. The below figure depicts the high level architecture.

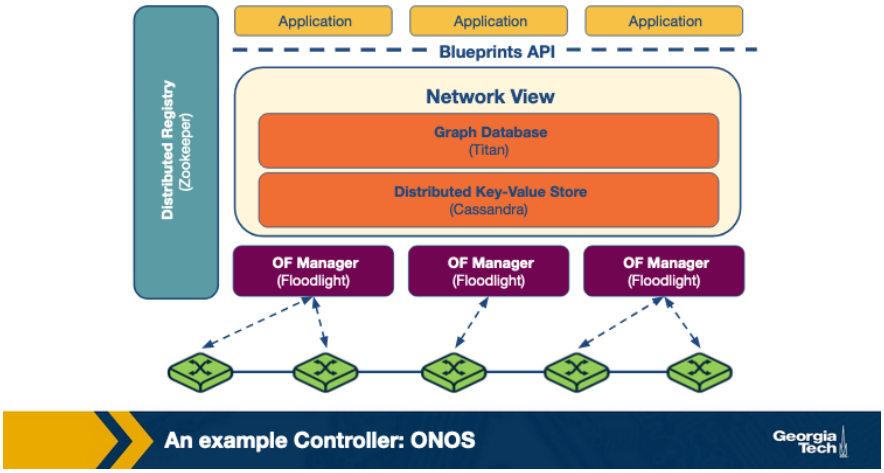

Figure 9: image

Owing to the distributed architecture of ONOS, there are several ONOS instances running in a cluster. The management and sharing of the network state across these instances is achieved by maintaining a global network view. This view is built by using the network topology and state information (port, link and host information, etc) that is discovered by each instance.

To make forwarding and policy decisions, the applications consume information from the view and then update these decisions back to the view. The corresponding OpenFlow managers receive the changes the applications make to the view, and the appropriate switches are programmed.

Titan, a graph database and a distributed key value store Cassandra is used to implement the view. The applications interact with the network view using the Blueprints graph API.

The distributed architecture of ONOS offers scale-out performance and fault tolerance. Each ONOS instance serves as the master OpenFlow controller for a group of switches. The propagation of state changes between a switch and the network view is handled solely by the master instance of that switch. The workload can be distributed by adding more instances to the ONOS cluster in case the data plane increases in capacity or the demand in the control plane goes up.

To achieve fault tolerance, ONOS redistributes the work of a failed instance to other remaining instances. Each switch in the network connects to multiple ONOS instances with only one instance acting as its master. Each ONOS instance acts as a master for a subset of switches. Upon failure of an ONOS instance, an election is held on a consensus basis to choose a master for each of the switches that were controlled by the failed instance. For each switch, a master is selected among the remaining instances with which the switch had established connection. At the end of election for all switches, each switch would have at most one new master instance.

Zoopkeeper is used to maintain the mastership between the switch and the controller.

Programming the Data Plane: The Motivation

In this topic, we are talking about the need to offer programmability on the data plane and we are introducing P4 which is a language that was developed for this purpose.

P4 (Programming Protocol-independent Packet Processors) is a high-level programming language to configure switches which works in conjunction with SDN control protocols. The popular vendor-agnostic OpenFlow interface, which enables the control plane to manage devices from different vendors, started with a simple rule table to match packets based on a dozen header fields. However, this specification has grown over the years to include multiple stages of the rule tables with increasing number of header fields to allow better exposure of a switch's functionalities to the controller.

Thus, to manage the demand for increasing number of header fields, a need arises for an extensible, flexible approach to parse packets and match header fields while also exposing an open interface to the controllers to leverage these capabilities.

P4 is used to configure the switch programmatically and acts as a general interface between the switches and the controller with its main aim of allowing the controller to define how the switches operate. The below figure explains the relationship between P4 and existing APIs such as OpenFlow, which targets to populate forwarding rules in fixed function switches:

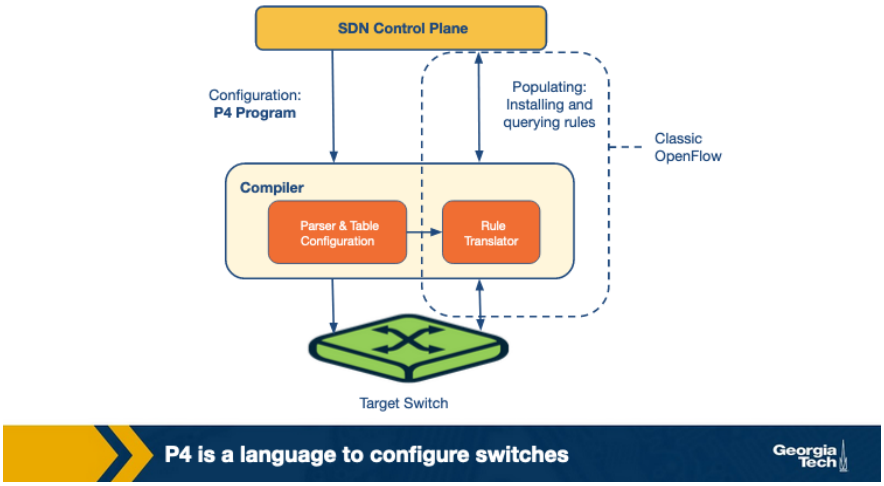

Figure 10: image

The following are the primary goals of P4:

- Reconfigurability: The way parsing and processing of packets takes place in the switches should be modifiable by the controller.
- Protocol independence: To enable the switches to be independent of any particular protocol, the controller defines a packet parser and a set of tables mapping matches and their actions. The packet parser extracts the header fields which are then passed on to the match+action tables to be processed.
- Target independence: The packet processing programs should be programmed independent of the underlying target devices. These generalized programs written in P4 should be converted into target-dependent programs by a compiler which are then used to configure the switch.

Programming the Data Plane: P4's Forwarding Model

In this topic, we are talking in more detail about P4 and the forwarding model that this approach proposes. The switches using P4 use a programmable parser and a set of match+action tables to forward packets. The tables can be accessed in multiple stages in a series or parallel manner. This contrasts with OpenFlow, which supports only fixed parsers based on predetermined header fields and only a series combination of match+actions tables.

The P4 model allows generalization of packet processing across various forwarding devices such as routers, load balancers, etc., using multiple technologies such as fixed function switches, NPUs, etc. This generalization

allows the design of a common language to write packet processing programs that are independent of the underlying devices. A compiler then maps these programs to different forwarding devices.

The following are the two main operations of the P4 forwarding model:

1. **Configure:** These sets of operations are used to program the parser. They specify the header fields to be processed in each match+action stage and also define the order of these stages.
2. **Populate:** The entries in the match+action tables specified during configuration may be altered using the populate operations. It allows addition and deletion of the entries in the tables.

In short, configuration determines the packet processing and the supported protocols in a switch whereas population decides the policies to be applied to the packets.

SDN Applications: Overview

In this topic, we are looking at an overview of the application areas of SDN.

1. **Traffic Engineering**

This is one of the major areas of interest for SDN applications with main focus on optimizing the traffic flow so as to minimize power consumption, judiciously use network resources, perform load balancing, etc. With the help of optimization algorithms and monitoring the data plane load via southbound interfaces, the power consumption can be reduced drastically while still maintaining the desired goals of performance. ElasticTree is one such application which identifies and shut downs specific links and devices depending on the traffic load. Load balancing applications such as Plug-n-Serve and Aster*x achieve scalability by creating rules based on wildcard patterns which enables handling of large numbers of requests from a particular group. Another use case of SDN applications is to automate the management of router configuration to reduce the growth in routing tables due to duplication of data. Large scale service providers also use SDN for traffic optimization to scale dynamically, e.g. ALTO VPN enables dynamic provisioning of VPNs in cloud infrastructure.

2. **Mobility and Wireless**

The existing wireless networks face various challenges in its control plane including management of the limited spectrum, allocation of radio resources and load-balancing. The deployment and management of various wireless networks (WLANS, cellular networks) is made easier using SDN. SDN-based wireless networks offer a variety of features including on-demand virtual access points (VAPs), usage of spectrum dynamically, sharing of wireless infrastructure, etc. OpenRadio, which is considered as the OpenFlow for wireless, enables decoupling of the wireless protocols from the underlying hardware by providing an abstraction layer. Light virtual access points (LVAPs) offer an improved way of managing wireless networks by using a one-to-one mapping between LVAPs and clients. The Odin framework leverages LVAPs and the applications built on it provide features such as mobility management, channel selection algorithms, etc. In contrast to traditional wireless networks, a user can move between APs without any visible lag as the mobility manager can automatically move the client LVAP to a different AP.

3. **Measurement and Monitoring**

The first class of applications in this domain aims to add features to other networking services. For example, new functions can be added easily to measurement systems such as BISmark in an SDN-based broadband connection, which enables the system to respond to change in network conditions. A second class of these applications aim to improve the existing features of SDNs using OpenFlow such as reducing the load on the control plane arising from collection of data plane statistics using various sampling and estimation techniques. OpenSketch is a southbound API that offers flexibility for network measurements. OpenSample and PayLess are examples of monitoring frameworks.

4. **Security and Dependability**

The applications in this area focus majorly on improving the security of networks. One approach of using SDN to enhance security is to impose security policies on the entry point to the network. Another approach

is to use programmable devices to enforce security policies on a wider network. DDoS detection, an SDN application identifies and mitigates DDoS flooding attacks by leveraging the timely information collected from the network. Furthermore, SDN has also been used to detect any anomalies in the traffic, to randomly mutate the IP addresses of hosts to fake dynamic IPs to the attackers (OF-RHM), and monitoring the cloud infrastructures (CloudWatcher). With regards to improving the security of SDN itself, there have been simple approaches like rule prioritizations for applications. However, there's still significant room for research and improvement in this area.

5. **Data Center Networking**

Data Center networking can be revolutionized by the use of SDN which aims to offer services such as live migration of networks, troubleshooting, real-time monitoring of networks among various other features. SDN applications can also help detect anomalous behavior in data centers by defining different models and building application signatures from observing the information collected from network devices in the data center. Any deviation from the signature history can be identified and appropriate measures can be taken. SDN also helps in performing dynamic reconfigurations of virtual networks involved in a live virtual network migration, which is an important feature of virtual networks in cloud. LIME is one such SDN application which aims to provide live migration and FlowDiff is an application which detects abnormalities.

SDN Application Example: A Software Defined Internet Exchange

In a previous topic, we talked about the Internet Exchange Points (IXPs) and their importance in today's Internet ecosystem. In this topic we are looking at how the SDN technology could be applied to improve the operation of an IXP. The SDN technology has many applications. In this topic we are only looking into an example SDN application for IXPs. The routing of packets across the Internet is currently handled through the popular Border Gateway Protocol (BGP). However, BGP has limitations which makes Internet routing unreliable and difficult to manage. The two main limitations are:

1. Routing only on destination IP prefix - The routing is decided based on the destination prefix IP of the incoming packet. There's no flexibility to customize rules for example based on the traffic application or the source/destination network.
2. Networks have little control over end-to-end paths - Networks can only select paths advertised by direct neighbors. Networks cannot directly control preferred paths but instead have to rely on indirect mechanisms such as "AS Path prepending".

Using SDN, researchers have proposed to address the above BGP limitations. SDN can perform multiple actions on the traffic by matching over various header fields, not only by matching on the destination prefix.

We have talked about IXPs at previous lecture, but as a reminder, an Internet Exchange Point (IXP) is a physical location that facilitates interconnection between networks so that they can exchange traffic and BGP routes. In the context of the IXPs, researchers have proposed an SDN based architecture, called SDX. SDX was proposed to implement multiple applications including:

- Application specific peering - Custom peering rules can be installed for certain applications, such as high-bandwidth video applications like Netflix or YouTube which constitute a significant amount of traffic volume.
- Traffic engineering - Controlling the inbound traffic based on source IP or port numbers by setting forwarding rules.
- Traffic load balancing - The destination IP address can be rewritten based on any field in the packet header to balance the load.
- Traffic redirection through middleboxes - Targeted subsets of traffic can be redirected to middleboxes.

SDX Architecture

Let's look into the proposed SDX architecture. In a traditional IXP the participant ASes connect their BGP-speaking border router to a shared layer-two network and a BGP route server. The layer-2 network is used for forwarding packets (data plane) and the BGP route server is used for exchanging routing information

(control plane). In the SDX architecture, each AS the illusion of its own virtual SDN switch that connects its border router to every other participant AS. For example, AS A has a virtual switch connecting to the virtual switches of ASes B and C.

Each AS can define forwarding policies as if it is the only participant at the SDX, without influencing how other participants forward packets on their own virtual switches. Each AS can have its own SDN applications for dropping, modifying, or forwarding their traffic. The policies can also be different based on the direction of the traffic (inbound or outbound). An inbound policy is applied on the traffic coming from other SDX participant on a virtual switch. An outbound policy is applied to traffic from the participant's virtual switch port towards other participants. The SDX is responsible to combine the policies from multiple participants into a single policy for the physical switch.

To write policies SDX uses the Pyretic language to match header fields of the packets and to express actions on the packets. Let's consider the example of application-specific peering, and let's see how a participant network expresses example policies:

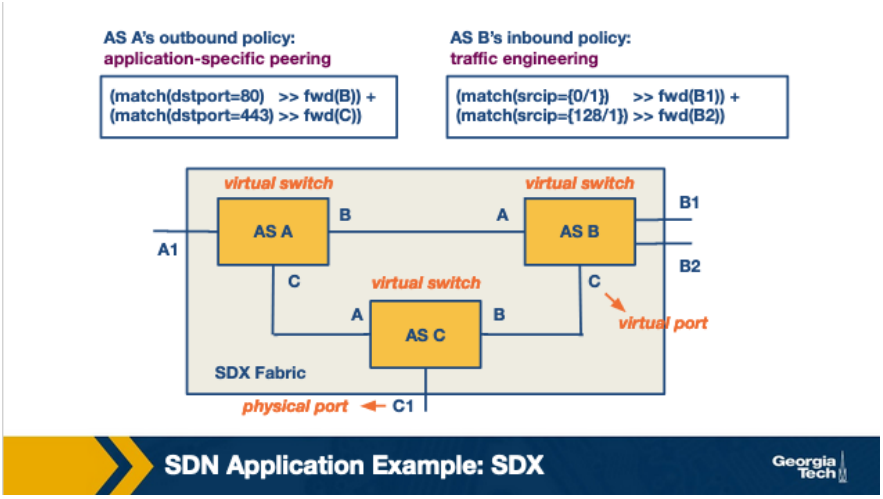

Figure 11: image

According to AS A's outbound policy, the HTTP traffic with destination port 80 is forwarded to AS B and HTTPS traffic with destination port 443 is forwarded to AS C.

This is expressed using the match statement:

```
(match(dstport = 80) >> fwd(B)) +
(match(dstport = 443) >> fwd(C))
```

- The match statement filters and return the packets that match the specified port numbers.
- The sequential operator "»" then forwards the returned packets to the next function.
- The fwd() function modifies the location (next destination) of the packet to the location of the corresponding switch.
- The parallel operator "+" applies each given policy to the packets and returns the combined output. If neither of the two policies matches, the packet is dropped.

SDN Applications: Wide Area Traffic Delivery

In this section, we'll look at a few applications of SDN, specifically SDX, in the domain of wide area traffic delivery.

1. **Application specific peering**

ISPs prefer dedicated ASes to handle the high volume of traffic flowing from high bandwidth applications such as YouTube, Netflix. This can be achieved by identifying a particular application's traffic using packet classifiers and directing the traffic in a different path. However this involves configuring additional and appropriate rules in the edge routers of the ISP. This overhead can be eliminated by configuring custom rules for flows matching a certain criteria at the SDX.

2. **Inbound traffic engineering** An SDN enabled switch can be installed with forwarding rules based on the source IP address and source port of the packets, thereby enabling an AS to control how the traffic enters its network. This is in contrast with BGP which performs routing based solely on the destination address of a packet. Although there are workarounds such as using AS path prepending and selective advertisements to control the inbound traffic using BGP, they come with certain limitations. An AS's local preference takes a higher priority for the outgoing traffic and the selective advertisements can lead to pollution of the global routing tables.

3. **Wide-area server load balancing** The existing approach of load balancing across multiple servers of a service involves a client's local DNS server issuing a request to the service's DNS server. As a response, the service DNS returns the IP address of a server such that it balances the load in its system. This involves DNS caching which can lead to slower responses in case of a failure. A more efficient approach to load balancing can be achieved with the help of SDX, as it supports modification of the packet headers. A single anycast IP can be assigned to a service, and the destination IP addresses of packets can be modified at the exchange point to the desired backend server based on the request load.

4. **Redirection through middle boxes** SDX can be used to address the challenges in existing approaches to using middleboxes (firewalls, load balancers, etc). The placement of middleboxes are usually targeted at important junctions, such as the boundary of the enterprise networks with their upstream ISPs. To avoid the high expenses involved in placing middleboxes at every location in case of geographically large ISPs, the traffic is directed through a fixed set of middleboxes by the ISPs. This is done by manipulating routing protocols such as internal BGP to essentially hijack a subset of traffic and sending it to a middlebox. This approach could result in unnecessary additional traffic being redirected, and is also limited by the fixed set of middleboxes. To overcome these issues, an SDX can identify and redirect the desired traffic through a sequence of middleboxes.