01 - Operations and Management Overview  
  
Welcome to the final third of the course. In the first section, you reviewed the basic building blocks of the internet, and in the second section, you learned how networks deal with large amounts of network traffic. In the final section, you'll learn how network operators manage their networks. More importantly, these topics will introduce you to the forefront of networking research. >> That's right. We're going to cover software defined networking, traffic engineering, and network security. >> Let's get started.

02 - Network Management Overview  
  
Welcome to the third course in CS 6250, where we will be discussing network operations and management. This segment in the course has three lessons. The first lesson is focused on software-defined networking, and its role in making network operations and management easier. The second module covers traffic engineering, which is the process by which network operators reconfigure the network to balance traffic demands across the network. The third lesson covers network security. We will start with a lesson on Software Defined Networking. But before we jump into the details, I'd like to motivate a little bit why. In particular I plan to tell you about the role of network operators in running the network. So what is network management? Network management is the process of configuring the network to achieve a variety of tasks. Network configuration achieves a variety of tasks including balancing traffic load across the network, achieving various security goals, and satisfying business relationships that may exist between the network that's being configured. And neighboring networks such as the network's upstream Internet service provider. A key aspect to network management is configuring the network. Unfortunately, if the network is not configured correctly, many things can go wrong. Configuration mistakes can lead to problems, such as persistent oscillation, whereby, routers can't agree on a route to a destination. Loops, where packets get stuck in between two or more routers and never actually make it to the destination. Partitions, where by the network is split into two or more segments that are not connected, and black holes where packets reach a router that does not know what to do with the packet and drops it, as opposed to sending on to it's ultimate destination.

03 - Why is Configuration Hard  
  
So why is configuration hard to get right? First, it's difficult to define what we mean by correct behaviour in the first place. Second, the interactions between multiple routing protocols can lead to unpredictability. Furthermore, each autonomous system on the internet is independently configured. And the interaction between the policies of these autonomous systems can lead to unintended, or unwanted behaviour. The third reason that configuration is hard, is that operators simply make mistakes. Configuration is difficult, and network policies are very complex. Furthermore, Network configuration has historically been distributed across hundreds, or more, network devices. Across the network where each device is configured with vendor-specific low-level configuration. We'll see in the first part of this course how Software Defined Networking or SDN changes this, by centralizing the network's configuration in a logically centralized controller. At a very high level, Software Defined Networking provides exactly the primitives that operators need to run the network better. In particular, SDN provides operators three things. The first, is network-wide views of both topology and traffic. The second, is the ability to satisfy network level objectives such as those that we talked about before including load balance, security, and other high level goals. The third thing, that software defined networking provides that network operators need, is direct control. In particular, rather than requiring network operators to configure each device individually with indirect configuration, SDN allows an operator to write a control program that directly affects the data plane. So rather than having to configure each device individually and guess or infer what might happen, software-defined networking allows a network operator to express network level objectives and direct control from A logically centralized controller. So to make network operations easier, routers should forward packets since router hardware is specialized to forward traffic at very high rates. They should collect measurements such as traffic statistics and topology information. But on the other hand, there's no reason that a router should have to compute routes. Although, conventionally, routing has operated as a distributed computation of forwarding tables, the computation doesn't inherently need to run on the routers. Rather, the computation could be logically centralized, and controlled, from a centralized control program. This logical centralization is the fundamental tenant of SDN. So a simple way of summing up Software Defined Networking is simply to remove routing from the routers, and perform that routing computation at a logically centralized controller. Now of course, SDN has evolved to incorporate, a much broader range of controls than simply routing decisions, and we'll talk about the range of control that SDN controllers enable in today's networks, throughout this lesson.

04 - Software Defined Networking  
  
Let's start with a brief overview of Software Defined Networking, or SDN. We'll first start by defining SDN, and in particular we'll talk about what is a Software Defined Network, then well talk about what are the advantages of SDN over a conventional network architecture. We'll overview the history of SDN, the infrastructure that supports it, in particular how SDNs are designed and built and the applications of SDN. Specifically, what they can be used for and how they can be used to simplify various network management tasks. Perhaps the best way to understand what an SDN is Is to compare it to the behavior of today's networks. Today's networks have two functions. The first is the Data Plane, whose task it is to forward packets to their ultimate destination. But in order for the Data Plane to work, we also need a way of computing the state that each of these routers has that allows the routers to make the right decision in forwarding traffic to the destination. The state that lives in each of these routers that allows the routers to make these decisions about how to forward packets are called routing tables. It's the job of the network's Control Plane to compute these routing tables. In conventional networks, the Control and Data Plane both run on the routers that are distributed across the network. In an SDN, the Control Plane runs in a logically centralized controller. Additionally, the controller typically controls multiple routers across the network and often, the control program exerts control over all the routers in the network, thus facilitating network-wide control. These two characteristics are the defining features of a Software Defined Network. The separation of data and control allows a network operator to build a network with commodity devices, where the control, resides in a separate control program. This re-factoring allows us to move from a network where devices are vertically integrated making it very tough to innovate to a network where the devices have open interfaces that can be controlled by software. Thus, allowing for much more rapid innovation. Let's survey a brief history of SDN. Previous to 2004, configuration was distributed, leading to buggy and unpredictable behavior. Around 2004, we had the idea to control the network from a logically centralized high level program. That logically centralized controller focused on the border gateway protocol, and was called the routing control platform, or RCP. In 2005, researchers generalized the notion of the RCP for different planes. The decision plane which computed the forwarding state for devices in the network, the Data Plane, which forwarded traffic based on decisions made by the decision plane. And the dissemination and discovery planes, which provide the decision plane the information that it needs to compute the forwarding state, which ultimately gets pushed to the data plane. Around 2008, these concepts effectively hit the mainstream, through a product called OpenFlow. OpenFlow's intellectual roots are with the RCP and 4D. But OpenFlow has made practical when merchant silicon vendors open their APIs, so that switch chipsets could be controlled from software. So suddenly there was an emergence of cheap switches that were build based on open chip sets that could be controlled from software. This development effectively allowed us to decouple the control plane and the data plane in commodity switching hardware.

05 - Advantages of SDN  
  
SDN has many advantages over conventional networks. It's easier to coordinate behavior among a network of devices. The behavior of the network is easier to evolve, and it's also easier to reason about. These characteristics are all rooted in the fact that the control plan is separate from the data plan. Having a separate control plane or control program allows us to provide conventional cs techniques to old networking problems. So as before it was incredibly difficult to reason about or debug a network's behavior if the network behavior is now controlled by a logically centralized control program. We can use techniques from programming languages or software engineering to help us reason about the behavior of the network. As far as SDN's infrastructure is concerned, the Control Plane is typically a software program written in a high level language, such as Python or C, on the other hand the Data Plane. Is typically programmable hardware that's controlled by the control plane. The controller effects the forwarding state that's in the switch using control commands. Open flow is one standard that defines a set of control commands by which the controller can control the behavior of one or more switches. SDN has many applications including data centers, wide area backbone networks, enterprise networks, internet exchange points or IXPs, and home networks. Later modules in this course will explore how software defined networks. Can solve network management problems in some of these areas. In this course we will focus in particular on the first three applications.

06 - Control Plane Operations  
  
So as a quick quiz, which of the following are examples of control plane operations? Computing a forwarding path, that satisfies some high-level policy such as an access control policy, computing a shortest path routing tree, rate-limiting traffic so that the overall sending rate doesn't exceed a certain throughput Load balancing traffic based on a hash of the packet source IP address, or authenticating a user's device based on its MAC address. Please check all options that apply.

07 - Control Plane Operations  
  
The job of the Control Plane is to compute state that ultimately ends up in the data plane. So computing a forwarding path that satisfies a high-level policy is something that the Control Plane would do. The Control Plane can also compute shortest path routing trees. And it might make decisions about whether or not a user's device should be allowed to send traffic or not based on that device's MAC address. Rate-limiting is something that is typically done in the data plane, and the load-balancing example that we have listed here. Is such that a router or a switch would make decisions in the data plane based on a hash of the source IP address. So all of the decisions are being made at forwarding time, not by a centralized high-level program.

08 - Control and Data Planes  
  
Let's quickly review the difference between the Control plane and the Data plane. The control plane is the logic that controls forwarding behavior. Examples of control plane functions include routing protocols as well as logic for configuring network middle boxes. Now, a routing protocol might compute shortest paths or a topology. But ultimately, the results of such computations must be installed in switches that actually do the forwarding. The forwarding table themselves and specifically the actions associated with forwarding traffic according to the Control plane logic is what constitutes the data plane. So examples of data plane function include forwarding packets at the IP layer, and doing things like switching at layer two. So to reiterate, routing protocol functions that compute the paths are control plane functions. Where as, the act of actually taking a packet, on an input port, and forwarding it, to an output port, is a data plane function. So why is separating the data and control planes a good idea? The first reason, is independent evolution and development. Thus, software control of the network, can involve independently of the network hardware. The second reason that separating data and control plan is a good idea is the opportunity to control the network behavior from a high-level software program. Controlling the network from a high-level program in theory allows network operators to debug and check network behavior more easily. Then in the status quo, where network behavior is determined by the distributed low level configuration across hundreds of switches and routers. The separation of data and control provides opportunities for better network management and data centers by facilitating such network tasks as virtual machine migration to adapt to fluctuating network demands. In Routing, the separation of data and control provides more control over decision logic. In Enterprise networks, SDN provides the ability to write security applications such as applications that manage network access control. In Research networks, the separation of data and control. Effectively allows to virtualize the network, so that, research networks and experimental protocols can co-exist with production networks on the same, underlying network hardware.

09 - Reasons for Separating Data and Control  
  
So as a quiz, what are some of the reasons for separating the control and data planes? Eliminating a single point of failure? Ability to scale to much larger networks? Independent evolution of the data and control plane? Separating vendor hardware from control logic? Or ease of reasoning about network behavior? Please check all options that apply.

10 - Reasons for Separating Data and Control  
  
Separating the data and control plane can allow for independent evolution of the data and control plane. Separating vendor hardware from the logic that controls the behavior of the network, and the potential to more easily reason about network behavior since the behavior is now controlled from a single logically-centralized control program. While it's possible that separating the control plane from the data plane Could result in architectures that are more fault tolerant or more scalable. The separation of data and control plain does not inherently make the network more fault tolerant or more scalable. Therefore, neither of the first two options apply.

11 - Example Data Centers  
  
One example, where SDN can provide huge wins, is in the data center. A data center, typically consists of many racks of servers. And any particular cluster might have, as many as 20,000 servers. Assuming that each one of these servers can run about 200 virtual machines. That's 400,000 virtual machines in a cluster. A significant problem is provisioning or migrating these virtual machines in response to varying traffic loads. SDN solves this problem by programming the switch state from a central database. So supposing, I have two virtual machines within the data center that needs to communicate with one another. The forwarding state in the switches, in the data center ensures that traffic is forwarded correctly. If we need to provision additional virtual machines. Or migrate a virtual machine from one server to another in the data center, the state in these switches must be updated. Updating the state in this fashion is much easier to do, from a central controller or a central database, facilitating. This type of Virtual Machine Migration in the data center is one of the early killer apps of software-defined networking. This type of migration is also made easier by the fact that the servers are addressed with Layer two Addressing. And the entire data center Looks like a flat layer two topology. What this means, is that a server can be migrated from one portion of the data center to another without requiring the virtual machine to obtain new addresses. All that needs to happen for forwarding to work ,is the state of these switches. Needs to be updated. The task of updating switch date in this fashion is very easy to do, when the control and data plans are separate.

12 - Managing Data Centers  
  
Let's have another quiz on data centers. So how does the control data plan separation make managing data centers easier? The ability to monitor and control routes from a central point of control. The ability to migrate virtual machines without renumbering host addresses. A requirement for fewer switches or making load balance automatic. Please again check all that apply.

13 - Managing Data Centers  
  
So as we discussed, control data plane separation can make it easier to manage the data center by monitoring and controlling routes from a central point and allowing virtual machines to be migrated without renumbering host addresses. The control data plane separation does not inherently make it possible to build a data center with few switches nor does it automatically balance load.

14 - Challenges  
  
As another example where a control and data plan separation comes in handy, let's look at the security of internet backbones. Where filtering attack traffic is a regular network management task. Suppose that an attacker is sending lots of traffic towards a victim. In this case a measurement system might detect the attack, identify the entry point, and a controller such as the RCP might install what is called a null route to ensure that no more traffic reaches the victim from the attacker. Two fundamental challenges with SDN are scalability and consistency. In an SDN a single control element might be responsible for many forwarding elements. So control element might be responsible for hundreds to thousands of switches. Of course, for redundancy and reliability, typically we want to replicate the controller. So while the controller is logically centralized, physically there may be many replicas. And, in such a deployment scenario we need to ensure that different controller replicas see the same view of the network so that they make consistent decisions when they're installing state in the data plane. A final challenge that's also worth mentioning is security or robustness. In particular, we want to make sure that the network continues to function correctly in the event that a controller replica fails or is compromised.

15 - Coping With Scalability  
  
So as a brief quiz or thought question, let's think about some approaches for coping with the scalabilty associated with the control and data plane separation. One could, for example, eliminate redundant data structures in the controller. Or only perform control operations for a limited number of network operations, such as only performing control operations for routing decisions. One might send all traffic through the controller to minimize forwarding decisions that routers and switches need to make. One could cache forwarding decisions in the switches, or run multiple controllers.

16 - Coping With Scalabilty  
  
Eliminating redundant data structures can help save memory in the control program running at the controller. Only performing ,a fixed number of network management operations such as routing, can insure that the controller doesn't have to do too much, thereby improving scalability. Caching forwarding decisions, that the control plain has already made ,in the switches, can ensure ,that not to much traffic, is redirected to the controller, and running multiple controllers can distribute the load of the control plane across multiple replicas, sending all traffic to the controller only increases the controller load, and would not help with scale ability.

17 - Different SDN Controllers  
  
Now that we have a better understanding, of, the benefits of separating the data and control plane, let's now have a look, at the many different options for SDN controllers. There are a number of different SDN controllers that exist, including NOX, Ryu, Floodlight, Pyretic, Frenetic, Procera, Route Flow, Trema, and the list goes on. In this lesson, we will explore the merits of these three controllers. And when we get to the lesson on Programming SDN, we will take a close look at Frenetic and Pyretic. Let's now jump in and take a look at these three controllers.

18 - NOX Overview  
  
Nox was a first generation open flow controller. It is open source, stable, and widely used. There are two flavors of Nox. Classic Nox, and the New Nox. Classic Nox was written in C++ and Python and is no longer supported. The new Nox is C++ only. The Code base is fast, clean and well supported. More information about Nox is available at noxrepo.org. In a Nox network, there may be a set of switches and various network-attached servers. The controller maintains a network view and the controller may also run several applications that operate on that network view. The basic abstraction that NOX supports is a switch control abstraction where open flow is the prevailing protocol. Control is defined at the granularity of flows which are defined. By a ten-tuple in the original OpenFlow specification. So depending on whether a particular packet matches a subset of values specified as a flow rule, the controller may make different decisions for packets that belong to different parts of flow space. Or packets that match different subsets, of the fields defined by a flow. A flow is defined by the header or the 10-tuple which I distoluted to. A counter which maintains statistics and actions that should be performed. On packets that match this particular flow definition. Actions might include forwarding the packet. Dropping it or sending it to the controller. When a switch recives a packet, it updates it's counters for counting packets that belong to that flow and applies the correspondence actions for that flow, which include forwarding, dropping or sending to a controller. The basic programatic interface. For the [UNKNOWN] controller is based on events. The controller knows how to process different types of events, such as a switch, join or leave a packet in or packet received event should the switch redirect packet to controller as well as various statistics. The controller also keeps tracks of a network view. Which includes a view of the underlying network topology, and it also speaks a control protocol to the switches in the network. That control protocol effectively allows the controller to update the state in the network switches. The Nox controller implements the open flow protocol. Nox has implemented in C++ And it supports opaflow 1.0, a fork of nox called CPQD supports versions 1.1, 1.2 and 1.3. The programming is event based and a programmer can write an application by writing even handlers for the nox controller. NOX provides good performance but requires you to understand and be comfortable with the facilities and semantics of low level open flow commands. Later in this module, we will explore controllers based on pyretic and frenetic that do not have this characteristic. NOX also requires the programmer to write the control application in C++, which can be slow for development and debugging. To address the shortcomings that are associated with development in C++, Pox was developed. Pox is widely used, maintained, and supported. It's also easy to use, and easy to read and write the control programs. Of course, as might come with implementing a controller in python, the performance of Pox is not as good as the performance of Nox.

19 - When to Use Pox  
  
So as a quick quiz, when might you use Pox? In a class project? In a large internet data center? Or in a university research project? Please check all that apply.

20 - When to Use Pox  
  
You might use Pox in a class project, or in a university research project where there's a need to quickly prototype, and evaluate a brand new control application. Pox is less applicable in a large internet data center, because it does not perform as well as other controllers.

21 - Ryu, Floodlight, Nox, and Pox  
  
Other controllers include Ryu, which is an open source Python controller. Ryu supports OpenFlow 1.0, 1.2, and 1.3, as well as the Nicira extensions. It also works with Open Stack. The support for the later versions of OpenFlow. And the integration with the open stack, are advantages over other SDN controllers. Because Ryu is implemented in Python, it still does not perform as well other SDN controllers, such as nox. Another popular SDN controller is floodlight. Floodlight is written in Java. It supports Overflow 1.0, and is a fork from the early beacon controller. Floodlight is maintained by big switch networks. Advantages include good documentation, integration with the rest API, and good performance. Unfortunately it also has a fairly steep learning curve. So you should use floodlight is you already know Java, if you need production level performance and support. And you will use the rest api to interact with the controller. So we can compare these two controllers, with the two controllers that we already discussed, Nox and Pox. We have controllers in three different languages, Python, Java, and C++. We have controllers that support later versions of open-flow, and support open stack. And we have controllers that provide better performance, as well as controllers that are easier to use for rapid prototyping. All of these controllers are still relatively hard to use, because they entail interacting directly with open flow, flow table modifications. Which operate at a very low level of matching on flows and performing specific actions. As we'll see, it's possible to develop programming languages on top of these controllers, that make it much easier for a neetwork opeartor to reason about network behavior. Before we jump into higher level programming languages however, let's first see how we can use these existing control frameworks to customize network control.

22 - Customizing Control  
  
In this lesson, we will learn how to write control programs to customize network control. We will review the operation of a hub and a learning switch, then we will explore how to use the POX controller to create a simple MiniNet topology. And then we will explore how to customize the Pox controller to perform two types of network control. As a review, when a host sends a packet to a hub, the hub maintains no state About which output port a package should be forwarded to reach a particular destination. Therefore, the hub simply forwards the input package on every output port. In Pox, this code is fairly simple. When the controller starts. It adds a listener that listens for a connection up, which is a connection from a switch. When switch connects, it simply sense and open-flow, flow modification back to the switch it says flood all packets out every output port. The first function here involved creates the open-flow massage. And the second sends that message back to the switch. In contrast a learning switch maintains a switch table that's initially empty. But when a packet arrives on input port 1, the switch creates a table. That associates host A with output port 1, such that whenever a subsequent packet is destined for destination A, the switch knows to forward the packet via output port 1. I won't show you the full Python code here. But it's fairly simple, and you can go look at the Pox distribution to see the learning switch example. As before, when the first packet arrives at the switch, it is diverted to the controller, at this point, the controller maintains a hash table that maps the address to the out put port, when it sees that first packet >From Post A, it updates the address and port table. If the packet's a multicast packet,the controller makes a decision to flood that packet on all output ports. Likewise, if there's no table entry for the destination for that packet, the controller also instructs the switch to forward the packet on all output ports. If the source and destination address are the same, The controller instructs the switch to drop the packet. Otherwise, the controller installs the flow table entry corresponding to that destination address and output port. Installing that flow table entry in the switch prevents future packets for that flow from being redirected to the controller. Rather, all subsequent packets on that flow can be handled directly by the switch, since it now knows which output port to send a packet for that particular destination.

23 - Summary  
  
OpenFlow makes modifying forwarding behavior easy, because forwarding decisions are based on matches on the openflow 10-tuple. There to switching is simply a match on the destination Mac address which has a corresponding action of forwarding out of particular output port. If all of the fields are specified for forwarding out a particular output port then we have flow switching behavior. If all of the flow specifications are wild carded except for say the source MAC address to make a forwarding or drop decision Then we have a firewall. Constructing a firewall is as simple as building a hash table that stores key value pairs where the table maps a switch and source MAC address to a true or false value depending on whether traffic should be forwarded or dropped. The controller Might then only decide to forward traffic if the firewall entry maps to true. It is important to emphasize the performance implications of caching the decisions at the switch. So, packets only reach the controller if there's no flow table entry at the switch. If on the other hand, there is a float table entry at the switch, then the switch can simply forward the packets rather than sending them to the controller. So when a controller decides to take an action on a packet, it installs that action as a flow table entry in the switch, and that decision or flow table entry is cached until that flow table entry expires. In summary, customizing control is easy. We've explored how to use the POX controller to develop alternate control programs. And it's possible to turn a switch into a firewall in less than 40 lines of python code. We also explored the performance benefits of caching rules and decisions, to avoid sending too much traffic to the controller. As we know, forwarding performance in a switch is as fast but whenever we have to send traffic to the controller it slows things down. So whatever decisions we can cache in the switch will only serve to improve the performance of the network.