1 - SDN Intro - lang\_en\_vs4  
  
1 00:00:00,180 --> 00:00:01,740 Now that you know that network management 2 00:00:01,740 --> 00:00:04,480 is a tough, complicated process, we're going to 3 00:00:04,480 --> 00:00:05,260 take a look at one of the 4 00:00:05,260 --> 00:00:08,300 most recent advancements in networking, software defined networking. 5 00:00:08,300 --> 00:00:10,430 &gt;&gt; Along the way, you're going to complete two 6 00:00:10,430 --> 00:00:13,060 exciting projects on Mininet. In the the first, you'll write 7 00:00:13,060 --> 00:00:15,660 you're own virtual switch. In the second, you'll use a 8 00:00:15,660 --> 00:00:17,750 programming language designed for software-defined 9 00:00:17,750 --> 00:00:19,030 networking to create a firewall.

10 - Network Virtualization - lang\_en\_vs5  
  
1 00:00:00,240 --> 00:00:02,900 Let's now talk about an application of software 2 00:00:02,900 --> 00:00:06,420 defined networking. Which is network virtualization. So we'll 3 00:00:06,420 --> 00:00:09,190 talk first about what network virtualization is, then 4 00:00:09,190 --> 00:00:11,870 we'll talk about how it's implemented. And then 5 00:00:11,870 --> 00:00:14,800 we'll talk about some examples and applications, such 6 00:00:14,800 --> 00:00:18,510 as Mininet. So network virtualization is simply an 7 00:00:18,510 --> 00:00:21,970 abstraction of the physical network. Where multiple logical 8 00:00:21,970 --> 00:00:25,420 networks can be run on the same underlying 9 00:00:25,420 --> 00:00:28,750 shared physical substrate. For example, a logical network 10 00:00:28,750 --> 00:00:32,369 might map a particular network topology onto the 11 00:00:32,369 --> 00:00:35,260 underlying physical topology. And there might be multiple 12 00:00:35,260 --> 00:00:37,790 logical networks, that map onto the same physical 13 00:00:37,790 --> 00:00:41,870 topology. And these logical networks might actually share 14 00:00:41,870 --> 00:00:45,220 nodes and links in the underlying physical typology. 15 00:00:45,220 --> 00:00:48,290 But each logical network has its own view 16 00:00:48,290 --> 00:00:50,590 as if it were running its own private version 17 00:00:50,590 --> 00:00:52,360 of the network. Now you can see from 18 00:00:52,360 --> 00:00:56,090 this picture, that the nodes in the physical network 19 00:00:56,090 --> 00:00:58,430 need to be shared or sliced. So the 20 00:00:58,430 --> 00:01:01,490 nodes in the physical typology might be virtual machines. 21 00:01:01,490 --> 00:01:05,670 Similarly, a single link in the logical topology 22 00:01:05,670 --> 00:01:09,410 might map to multiple links in the physical topology. 23 00:01:09,410 --> 00:01:12,510 The mechanism to achieve these virtual links is 24 00:01:12,510 --> 00:01:15,720 typically through tunneling. So a packet that's destined from 25 00:01:15,720 --> 00:01:18,370 A to B in the logical topology, might 26 00:01:18,370 --> 00:01:21,540 be encapsulated, in a packet that's destined for 27 00:01:21,540 --> 00:01:25,180 node X first, before the packet is decapsulated 28 00:01:25,180 --> 00:01:27,770 and ultimately sent to B. It may also 29 00:01:27,770 --> 00:01:31,560 be easy to understand virtual networking as an 30 00:01:31,560 --> 00:01:34,238 analogy to virtual machines, which you may be 31 00:01:34,238 --> 00:01:37,140 familiar with already. So in a virtual machine 32 00:01:37,140 --> 00:01:40,850 environment, we have virtual machines where a hypervisor, 33 00:01:40,850 --> 00:01:43,510 arbitrates access to the underlying physical 34 00:01:43,510 --> 00:01:46,520 resources. Providing to each virtual machine the 35 00:01:46,520 --> 00:01:48,250 illusion that it's operating on its 36 00:01:48,250 --> 00:01:53,040 own dedicated version of the hardware. Similarly, 37 00:01:53,040 --> 00:01:56,200 with virtual networking, a network hypervisor 38 00:01:56,200 --> 00:01:59,550 of sorts arbitrates access to the underlying 39 00:01:59,550 --> 00:02:03,350 physical network, to multiple virtual networks. Providing 40 00:02:03,350 --> 00:02:06,040 the illusion that each virtual network actually 41 00:02:06,040 --> 00:02:08,180 has its own dedicated physical network.

11 - Why Use Network Virtualization - lang\_en\_vs3  
  
1 00:00:00,330 --> 00:00:02,710 One of the main motivations for the rise 2 00:00:02,710 --> 00:00:05,290 of virtual networking was the" ossification" of the 3 00:00:05,290 --> 00:00:08,600 internet architecture. In particular because the internet protocol 4 00:00:08,600 --> 00:00:12,060 was so pervasive, it made it very difficult 5 00:00:12,060 --> 00:00:14,840 to make fundamental changes to the way the 6 00:00:14,840 --> 00:00:18,210 underlying internet architecture operated. There was a lot 7 00:00:18,210 --> 00:00:22,110 of work on overlay networks in the 2000's 8 00:00:22,110 --> 00:00:25,470 but one size fits all network architectures were very 9 00:00:25,470 --> 00:00:28,960 difficult to deploy. So rather than try 10 00:00:28,960 --> 00:00:34,260 to replace existing network architectures, network virtualization 11 00:00:34,260 --> 00:00:38,310 was intended to allow for easier evolution. 12 00:00:38,310 --> 00:00:42,460 In other words, network virtualization enables evolution because 13 00:00:42,460 --> 00:00:45,660 we didn't have to pick a winner for a replacement for IP. We could 14 00:00:45,660 --> 00:00:49,360 instead let multiple architectures exist in parallel. 15 00:00:49,360 --> 00:00:51,400 Now, this was sort of a green field 16 00:00:51,400 --> 00:00:57,990 view of why virtual networking was potentially a good idea. In practice, network 17 00:00:57,990 --> 00:01:01,675 virtualization has really taken off in multi 18 00:01:01,675 --> 00:01:04,580 -tenant data centers where there may be 19 00:01:04,580 --> 00:01:07,980 multiple tenants or applications running on 20 00:01:07,980 --> 00:01:10,010 a shared cluster of servers. Well know 21 00:01:10,010 --> 00:01:13,540 examples of this include Amazon's EC2, Rack 22 00:01:13,540 --> 00:01:16,490 Space, and things like Google App Engine. 23 00:01:16,490 --> 00:01:19,380 Large service providers such as Google, Yahoo 24 00:01:19,380 --> 00:01:22,780 and so forth also use network virtualization 25 00:01:22,780 --> 00:01:25,460 to adjust the resources that are devoted 26 00:01:25,460 --> 00:01:27,570 to any particular service at a given time.

12 - Network Virtualization Quiz - lang\_en\_vs4  
  
1 00:00:00,470 --> 00:00:04,640 So what are the motivations for network virtualization or virtual networks 2 00:00:04,640 --> 00:00:08,070 that we've discussed? Easier troubleshooting. Facilitating 3 00:00:08,070 --> 00:00:10,710 research and evolution by allowing coexistence 4 00:00:10,710 --> 00:00:13,530 of production networks with experimental ones. 5 00:00:13,530 --> 00:00:16,370 Better forwarding performance. And adjusting the 6 00:00:16,370 --> 00:00:20,130 resources of the network as demands change. Please check all that apply.

13 - Network Virtualization Quiz Solution - lang\_en\_vs4  
  
1 00:00:00,290 --> 00:00:03,080 As we discussed, virtual networks can facilitate research 2 00:00:03,080 --> 00:00:06,980 in evolution by allowing experimental networks to coexist 3 00:00:06,980 --> 00:00:10,020 with production networks. Because the networks are virtual, 4 00:00:10,020 --> 00:00:12,730 they can be scaled up and down adjusting the 5 00:00:12,730 --> 00:00:15,410 resources that are devoted to any one particular 6 00:00:15,410 --> 00:00:18,980 service as demands change. We discuss this in the 7 00:00:18,980 --> 00:00:21,730 context of production networks, such as Google and 8 00:00:21,730 --> 00:00:25,410 Yahoo. Virtual networks are not inherently easier to troubleshoot, 9 00:00:25,410 --> 00:00:27,170 nor do they necessarily provide better 10 00:00:27,170 --> 00:00:29,950 forwarding performance. In fact, forwarding performance may 11 00:00:29,950 --> 00:00:34,130 be worse, due to the additional level of indirection that has been added.

14 - Network Virtualization Uses SDN - lang\_en\_vs4  
  
1 00:00:00,250 --> 00:00:01,400 Some of the promised benefits of 2 00:00:01,400 --> 00:00:05,120 Network Virtualization are more rapid innovation since 3 00:00:05,120 --> 00:00:08,560 innovation can proceed at the rate at 4 00:00:08,560 --> 00:00:11,980 which software evolves. Rather on hardware development 5 00:00:11,980 --> 00:00:14,240 cycles, allowing for new forms of network 6 00:00:14,240 --> 00:00:17,880 control, and potentially simplifying programming. It is 7 00:00:17,880 --> 00:00:20,400 important to make a distinction between Network 8 00:00:20,400 --> 00:00:25,390 Virtualization and Software-Defined Networking. Network Virtualization is 9 00:00:25,390 --> 00:00:29,160 arguably one of the first killer applications for 10 00:00:29,160 --> 00:00:31,820 SDN. And in some sense, SDN is a 11 00:00:31,820 --> 00:00:35,920 tool for implementing Network virtualization. But the two 12 00:00:35,920 --> 00:00:37,790 are not one in the same. Remember the 13 00:00:37,790 --> 00:00:40,930 defining tenant of SDN is the separation of 14 00:00:40,930 --> 00:00:43,940 the data and control plant, whereas the defining 15 00:00:43,940 --> 00:00:47,660 tenant of Network virtualization. Is to separate the 16 00:00:47,660 --> 00:00:50,520 underlying physical network from the logical networks that 17 00:00:50,520 --> 00:00:55,580 lie on top of it. So SDN can be used to simplify many aspects of 18 00:00:55,580 --> 00:00:59,180 Network virtualization. But it does not inherently 19 00:00:59,180 --> 00:01:03,020 obstruct the details of the underlying physical network.

15 - Characteristics of Network Virtualization Quiz - lang\_en\_vs4  
  
1 00:00:00,120 --> 00:00:01,435 So which of the following are 2 00:00:01,435 --> 00:00:05,390 characteristics of network virtualization, but not necessarily 3 00:00:05,390 --> 00:00:08,950 characteristics of SDN? Be careful, the distinction 4 00:00:08,950 --> 00:00:11,530 between SDN and virtual networks was as 5 00:00:11,530 --> 00:00:17,800 we discussed in the previous part of this lesson. Allowing multiple tenants to 6 00:00:17,800 --> 00:00:21,930 share underlying physical infrastructure. Controlling behavior from 7 00:00:21,930 --> 00:00:25,440 a logically centralized controller. Separating logical and 8 00:00:25,440 --> 00:00:29,520 physical networks. Or separating data and control planes. 9 00:00:29,520 --> 00:00:31,510 Again, please feel free to check all that apply.

16 - Characteristics of Network Virtualization Quiz Solution - lang\_en\_vs4  
  
1 00:00:00,570 --> 00:00:03,830 Network virtualization can allow multiple tenants to share the 2 00:00:03,830 --> 00:00:07,814 underlying physical infrastructure. And it also separates logical and 3 00:00:07,814 --> 00:00:11,680 physical networks. The other two options are defining characteristics 4 00:00:11,680 --> 00:00:15,050 of software defined networking, but not of network virtualization.

17 - Design Goals for Network Virtualization - lang\_en\_vs4  
  
1 00:00:00,350 --> 00:00:03,350 So virtual networks have various design goals. It 2 00:00:03,350 --> 00:00:06,790 should be flexible, able to support different topologies, 3 00:00:06,790 --> 00:00:10,720 routing and forwarding architectures and independent configurations. They 4 00:00:10,720 --> 00:00:13,290 should be manageable, in other words, they should separate 5 00:00:13,290 --> 00:00:15,260 the policy that a network operator is trying 6 00:00:15,260 --> 00:00:18,330 to specify from the mechanisms of how those policies 7 00:00:18,330 --> 00:00:21,910 are implemented. They should be scalable, maximizing the 8 00:00:21,910 --> 00:00:25,840 number or coexisting virtual networks. They should be secure 9 00:00:25,840 --> 00:00:27,970 by isolating the different logical networks from 10 00:00:27,970 --> 00:00:31,050 one another. They should be programmable and 11 00:00:31,050 --> 00:00:33,380 they should be heterogenious in the sense 12 00:00:33,380 --> 00:00:36,930 that they should support different technologies. So virtual 13 00:00:36,930 --> 00:00:41,050 networks have two components, nodes and edges. 14 00:00:41,050 --> 00:00:44,560 The physical nodes themselves must be virtualized. One 15 00:00:44,560 --> 00:00:46,630 possible way virtualizing a node is a 16 00:00:46,630 --> 00:00:51,310 virtual machine. A more lightweight way of virtualizing 17 00:00:51,310 --> 00:00:56,210 a node is using a virtual environment such as a VServer or a Jail. 18 00:00:56,210 --> 00:00:59,245 The hypervisor or whatever technology is enabling 19 00:00:59,245 --> 00:01:02,670 the virtual environment can effectively slice the 20 00:01:02,670 --> 00:01:06,210 underlying physical hardware to provide the allusion 21 00:01:06,210 --> 00:01:09,390 of multiple guest nodes or multiple virtual 22 00:01:09,390 --> 00:01:13,910 nodes. Examples of node virtualization include, virtual 23 00:01:13,910 --> 00:01:16,890 machine environment such as Xen or VMware or 24 00:01:16,890 --> 00:01:20,540 what's called OS level virtualization or virtual environments, 25 00:01:20,540 --> 00:01:23,580 such as Linux Vservers. Now, in a virtual network, 26 00:01:23,580 --> 00:01:26,590 we need to connect these virtual machines. Each 27 00:01:26,590 --> 00:01:30,060 virtual machine or virtual environment has its own view 28 00:01:30,060 --> 00:01:32,250 of the network stack. And we may want 29 00:01:32,250 --> 00:01:35,330 to provide the appearance that these nodes are connected 30 00:01:35,330 --> 00:01:38,890 to one another over a Layer two topology, 31 00:01:38,890 --> 00:01:42,380 even if they are in fact separated by multiple 32 00:01:42,380 --> 00:01:49,210 IP hops. One possible way of doing that is to encapsulate the Ethernet packet as 33 00:01:49,210 --> 00:01:54,740 it leaves the VM on the left in an IP packet. The IP packet can then be destined 34 00:01:54,740 --> 00:01:56,650 for the IP address of the machine on 35 00:01:56,650 --> 00:01:59,870 the right, and when the packet arrives at this 36 00:01:59,870 --> 00:02:03,920 machine, the host can decapsulate the packet and 37 00:02:03,920 --> 00:02:07,800 pass the original Ethernet packet to the VM or 38 00:02:07,800 --> 00:02:10,820 the virtual environment that's residing on that physical 39 00:02:10,820 --> 00:02:13,450 node. Each of one these physical hosts, may 40 00:02:13,450 --> 00:02:16,695 in fact, host multiple virtual machines or virtual 41 00:02:16,695 --> 00:02:19,030 environments, which creates the need for a virtual 42 00:02:19,030 --> 00:02:22,020 switch that resides on a physical host. This 43 00:02:22,020 --> 00:02:26,090 virtual switch provides the function of networking virtual 44 00:02:26,090 --> 00:02:30,570 machines together over a virtual layer two topology. 45 00:02:30,570 --> 00:02:32,850 The Linux bridge is an example of a software 46 00:02:32,850 --> 00:02:35,620 switch that can perform this type of function. 47 00:02:35,620 --> 00:02:40,160 Open Vswitch is another example of software that performs 48 00:02:40,160 --> 00:02:42,940 this type of glue function. You can see more 49 00:02:42,940 --> 00:02:47,430 information about Open Vswitch on the URL provided here.

18 - Virtualization in Mininet - lang\_en\_vs4  
  
1 00:00:00,490 --> 00:00:02,560 The mini net tool we have been using 2 00:00:02,560 --> 00:00:06,040 in the course, is actually a example of, network 3 00:00:06,040 --> 00:00:10,080 visualization, we are in fact running a, an 4 00:00:10,080 --> 00:00:13,940 entire virtual network, on your laptop. When you start 5 00:00:13,940 --> 00:00:16,950 mini net using the MN script, each host 6 00:00:16,950 --> 00:00:20,695 in the virtual network, is a bash process. With 7 00:00:20,695 --> 00:00:23,270 it's own network name space. A network name 8 00:00:23,270 --> 00:00:26,490 space is kind of like a virtual machine except 9 00:00:26,490 --> 00:00:28,664 it's a lot more lightweight. It's infact 10 00:00:28,664 --> 00:00:31,350 called OS Level Virtualization. So, each one of 11 00:00:31,350 --> 00:00:34,000 these virtual nodes, has its own view of 12 00:00:34,000 --> 00:00:36,580 the network stack as shown here with these 13 00:00:36,580 --> 00:00:40,280 interfaces. But it has a shared filesystem and 14 00:00:40,280 --> 00:00:43,470 it's not, in fact, running it's own independent 15 00:00:43,470 --> 00:00:48,180 virtual machine. The root namespace manages the communication 16 00:00:48,180 --> 00:00:51,650 between these distinct virtual nodes, as well as 17 00:00:51,650 --> 00:00:56,140 the switch that connects these nodes in the topology that you set up. Virtual 18 00:00:56,140 --> 00:01:00,220 ethernet pairs are assigned two name spaces. 19 00:01:00,220 --> 00:01:03,610 For example, S1 eth1 is assigned to an 20 00:01:03,610 --> 00:01:09,870 interface in H2's network name space. And S1 eth2 is assigned to a network 21 00:01:09,870 --> 00:01:12,910 name space in H3's virtual network name 22 00:01:12,910 --> 00:01:17,090 space. The open flow switch effectively performs forwarding 23 00:01:17,090 --> 00:01:19,760 between the interfaces in the root name 24 00:01:19,760 --> 00:01:22,650 space. But because the interfaces are paired, we 25 00:01:22,650 --> 00:01:26,680 get the illusion, of sending traffic, between h2 26 00:01:26,680 --> 00:01:30,320 and h3. When we make modifications, to the 27 00:01:30,320 --> 00:01:33,230 open flow switch, via the controller we're infact 28 00:01:33,230 --> 00:01:35,830 doing that in the root name space. In 29 00:01:35,830 --> 00:01:40,020 summary, virtual networks facilitate flexible, agile deployment, by 30 00:01:40,020 --> 00:01:42,840 enabling rapid innovation at the pace of software. 31 00:01:42,840 --> 00:01:46,140 Vender independence, and scale. We talked about the distinction 32 00:01:46,140 --> 00:01:49,670 between SDN's and virtual networks, as well as various 33 00:01:49,670 --> 00:01:54,130 technologies that enable virtual networks, such as virtual machines 34 00:01:54,130 --> 00:01:57,750 for creating virtual nodes, and tunneling for creating virtual links.

19 - SDN Programming Difficulty - lang\_en\_vs4  
  
1 00:00:00,260 --> 00:00:02,050 In this lesson we'll talk about the why 2 00:00:02,050 --> 00:00:06,510 and how of programming SDNs. Unfortunately programming OpenFlow is 3 00:00:06,510 --> 00:00:09,310 not easy. It offers only a very low level 4 00:00:09,310 --> 00:00:13,280 of abstraction in the form of match action rules. 5 00:00:13,280 --> 00:00:15,710 The controller only sees events that switches don't know 6 00:00:15,710 --> 00:00:18,320 how to handle. And there can be race conditions 7 00:00:18,320 --> 00:00:21,270 if switch level rules are not installed properly, as 8 00:00:21,270 --> 00:00:24,040 we've already seen in the lesson on consistent updates

2 - Updates in Software Defined Networks - lang\_en\_vs4  
  
1 00:00:00,470 --> 00:00:05,950 In this lesson we'll be exploring consistent updates in SDN's. 2 00:00:05,950 --> 00:00:11,000 As a reminder from last lesson, we looked at how to update switch flow table 3 00:00:11,000 --> 00:00:16,460 entries using open flow control commands from the control. The 4 00:00:16,460 --> 00:00:21,530 open flow API however does not provide specific guarantees 5 00:00:21,530 --> 00:00:26,120 about the Level of consistency that 6 00:00:26,120 --> 00:00:29,820 packets along an end-to-end path can experience. So 7 00:00:29,820 --> 00:00:34,620 for example, updates to multiple switches along a 8 00:00:34,620 --> 00:00:36,430 path in a network that occur at different 9 00:00:36,430 --> 00:00:39,360 times may result in problems such as forwarding 10 00:00:39,360 --> 00:00:42,580 loops. Additionally, if updates to the switches along 11 00:00:42,580 --> 00:00:44,850 an end-to-end path occur in the middle of 12 00:00:44,850 --> 00:00:47,680 a flow, Packets from the same flow may 13 00:00:47,680 --> 00:00:52,090 be subjected to different network states. These two problems 14 00:00:52,090 --> 00:00:55,310 are known as consistency problems. The first problem 15 00:00:55,310 --> 00:00:57,900 is known as a packet level consistency problem and 16 00:00:57,900 --> 00:00:59,770 the second problem is known as a flow level 17 00:00:59,770 --> 00:01:03,760 consistency problem. In this lesson, we will explore these 18 00:01:03,760 --> 00:01:05,980 problems in more detail and look at various 19 00:01:05,980 --> 00:01:10,530 approaches to guaranteeing consistent updates in SDNs. To think 20 00:01:10,530 --> 00:01:13,220 about consistency properly, we first need a notion of 21 00:01:13,220 --> 00:01:17,280 a high level programming model that sits on top 22 00:01:17,280 --> 00:01:21,070 of what we would call the southbound interface. 23 00:01:21,070 --> 00:01:23,750 We'll talk about how to write applications, that 24 00:01:23,750 --> 00:01:26,180 use the controller interface that we learned about 25 00:01:26,180 --> 00:01:28,800 in the last lesson. That can rely on 26 00:01:28,800 --> 00:01:32,680 a better notion of consistency, than existing controller 27 00:01:32,680 --> 00:01:35,380 platforms currently provide. Let's first think about how 28 00:01:35,380 --> 00:01:38,140 we want to program these applications. And what 29 00:01:38,140 --> 00:01:42,340 type of abstraction the applications would require from the 30 00:01:42,340 --> 00:01:43,690 underlying control interface.

20 - SDN Programming Interface - lang\_en\_vs4  
  
1 00:00:00,170 --> 00:00:04,370 The solution to this is to provide some kind of northbound API, which is a 2 00:00:04,370 --> 00:00:07,580 programming interface that allows applications and other kinds 3 00:00:07,580 --> 00:00:10,890 of orchestration systems to program the network. So 4 00:00:10,890 --> 00:00:13,170 where we have at the low-level the 5 00:00:13,170 --> 00:00:17,690 controller updating state in the switch using OpenFlow 6 00:00:17,690 --> 00:00:21,090 flow modification rules we may have applications or 7 00:00:21,090 --> 00:00:25,460 orchestration systems that need to perform more sophisticated 8 00:00:25,460 --> 00:00:29,310 tasks, such as path computation, loop avoidance, 9 00:00:29,310 --> 00:00:31,850 and so forth. But we need a higher-level 10 00:00:31,850 --> 00:00:35,390 programming interface that allows these applications to talk 11 00:00:35,390 --> 00:00:38,498 to the controller so the application isn't writing 12 00:00:38,498 --> 00:00:42,880 low-level OpenFlow rules, but rather is expressing what 13 00:00:42,880 --> 00:00:44,908 it wants to have happen in terms of 14 00:00:44,908 --> 00:00:48,130 higher-level behaviors without regard to such things as 15 00:00:48,130 --> 00:00:50,690 whether or not the rules are being installed 16 00:00:50,690 --> 00:00:53,740 in a consistent, and correct fashion. Various people 17 00:00:53,740 --> 00:00:56,940 may write these applications including network operators, service 18 00:00:56,940 --> 00:00:59,920 providers, researchers, and really anyone who wants to 19 00:00:59,920 --> 00:01:03,690 develop capabilities on top of OpenFlow. The benefits 20 00:01:03,690 --> 00:01:07,140 of such a northbound API are vendor independence, 21 00:01:07,140 --> 00:01:09,490 as well as the ability to quickly modify 22 00:01:09,490 --> 00:01:13,570 or customize control through various popular programming languages. 23 00:01:13,570 --> 00:01:15,730 The idea is that these applications might be 24 00:01:15,730 --> 00:01:18,090 written in high-level programming languages, such 25 00:01:18,090 --> 00:01:21,120 as Python, and wouldn't actually have to 26 00:01:21,120 --> 00:01:24,370 perform low-level switch modifications, but rather could 27 00:01:24,370 --> 00:01:27,968 express policies in terms of much higher-level 28 00:01:27,968 --> 00:01:32,210 abstractions. Examples of such applications include the 29 00:01:32,210 --> 00:01:35,130 implementation of a large virtual switch abstraction, 30 00:01:35,130 --> 00:01:38,370 security applications, and services that may need 31 00:01:38,370 --> 00:01:41,840 to integrate traffic processing with middle boxes. 32 00:01:41,840 --> 00:01:43,770 This programmatic interface is called the 33 00:01:43,770 --> 00:01:46,430 northbound API and currently there's no 34 00:01:46,430 --> 00:01:51,400 standard for the northbound API, as there is for the southbound API 35 00:01:51,400 --> 00:01:55,470 in OpenFlow. But we'll look at various APIs in programming languages that 36 00:01:55,470 --> 00:01:57,900 each compile to OpenFlow rules that 37 00:01:57,900 --> 00:02:00,090 are installed on switches across the network.

21 - Frenetic Language - lang\_en\_vs4  
  
1 00:00:00,110 --> 00:00:05,590 One example of a programming language that sits on top of such a north-bound API 2 00:00:05,590 --> 00:00:10,880 is Frenetic, which is a SQL-Like query language. For example, Frenetic would 3 00:00:10,880 --> 00:00:14,390 allow a programmer to count the number 4 00:00:14,390 --> 00:00:17,930 of bytes, grouped by destination Mac address 5 00:00:17,930 --> 00:00:21,870 and report the updates to these counters 6 00:00:21,870 --> 00:00:25,360 every 60 seconds. The group by statement. 7 00:00:25,360 --> 00:00:27,790 The group by statement allows a grouping 8 00:00:27,790 --> 00:00:31,570 of counts by the destination mac address. Where 9 00:00:31,570 --> 00:00:34,310 allows restrictions to only count trafic coming from 10 00:00:34,310 --> 00:00:36,520 a web server coming in on a particular 11 00:00:36,520 --> 00:00:40,590 port and every specifies that the results of 12 00:00:40,590 --> 00:00:42,960 this query should only be returned every 60 13 00:00:42,960 --> 00:00:46,415 seconds. More information about Frenetic is available at 14 00:00:46,415 --> 00:00:50,470 frenetic-lang.org. And in the course, we'll actually going 15 00:00:50,470 --> 00:00:54,760 to use a language called Pyretic that is based on the 16 00:00:54,760 --> 00:00:58,920 same underlying theory as Frenetic, except that it's embedded in Python.

22 - Overlapping Network Policies - lang\_en\_vs4  
  
1 00:00:00,690 --> 00:00:03,710 One issue with programming at this higher 2 00:00:03,710 --> 00:00:06,490 level of abstraction is that an operator might 3 00:00:06,490 --> 00:00:09,730 write multiple modules, each of which effects 4 00:00:09,730 --> 00:00:13,340 the same traffic. For example, an operator might 5 00:00:13,340 --> 00:00:16,490 write an application that monitors traffic. Another 6 00:00:16,490 --> 00:00:19,280 one that specifies how routing should take place, 7 00:00:19,280 --> 00:00:22,200 another that involves the specification of firewall 8 00:00:22,200 --> 00:00:25,960 rules And yet another that balances traffic load 9 00:00:25,960 --> 00:00:29,270 across the links in the network. Ultimately, 10 00:00:29,270 --> 00:00:32,049 all of these applications, or modules, must 11 00:00:32,049 --> 00:00:34,930 be combined into a single set of 12 00:00:34,930 --> 00:00:37,220 open flow rules that together achieve the network 13 00:00:37,220 --> 00:00:40,320 operator's overall goal. For this, we need 14 00:00:40,320 --> 00:00:43,870 composition operators, or ways that specify How these 15 00:00:43,870 --> 00:00:48,150 individual modules should be combined or composed 16 00:00:48,150 --> 00:00:51,030 into a single coherent application. Let's now talk 17 00:00:51,030 --> 00:00:54,050 about two different ways to compose policies.

23 - Composing Network Policies with Pyretic - lang\_en\_vs4  
  
1 00:00:00,220 --> 00:00:02,000 One way of composing policies is to 2 00:00:02,000 --> 00:00:07,100 perform both operations simultaneously. For example, one might 3 00:00:07,100 --> 00:00:09,910 want to forward traffic but also count how 4 00:00:09,910 --> 00:00:11,910 much traffic is being forwarded. Both of those 5 00:00:11,910 --> 00:00:14,670 operations can be performed in parallel. Another 6 00:00:14,670 --> 00:00:18,130 way of composing policies is in sequence. Sequential 7 00:00:18,130 --> 00:00:21,860 composition performs one operation then the next. For 8 00:00:21,860 --> 00:00:25,420 example, we might want to implement a firewall. 9 00:00:25,420 --> 00:00:27,680 And whatever traffic makes it though the firewall 10 00:00:27,680 --> 00:00:31,800 might then be subjected to the switching policy. 11 00:00:31,800 --> 00:00:35,100 One example of sequential composition, might be a 12 00:00:35,100 --> 00:00:38,930 load balancer. In this example, a policy might 13 00:00:38,930 --> 00:00:42,040 take some traffic coming from, half of the 14 00:00:42,040 --> 00:00:45,140 source IP addresses, and rewrite that to one 15 00:00:45,140 --> 00:00:47,220 server replica and take the other half of 16 00:00:47,220 --> 00:00:50,870 the traffic and rewrite it to the other replica. 17 00:00:50,870 --> 00:00:53,920 After the load balancer rewrites the destination 18 00:00:53,920 --> 00:00:57,000 IP address, we need a routing module to 19 00:00:57,000 --> 00:01:00,010 forward the traffic out the appropriate port on 20 00:01:00,010 --> 00:01:03,030 the switch. In this case, we've used sequential 21 00:01:03,030 --> 00:01:06,400 composition to first apply a load balance policy 22 00:01:06,400 --> 00:01:09,270 that rewrites the destination IP address based on 23 00:01:09,270 --> 00:01:11,200 the source IP address where the traffic is 24 00:01:11,200 --> 00:01:16,600 coming from and sequentially apply a routing policy 25 00:01:16,600 --> 00:01:19,300 that forwards the traffic out the appropriate 26 00:01:19,300 --> 00:01:23,290 port. Depending on the resulting destination IP 27 00:01:23,290 --> 00:01:28,670 address after that rewrite has taken place. Notice that we can use predicates to 28 00:01:28,670 --> 00:01:32,620 specify which traffic traverses which modules. Those 29 00:01:32,620 --> 00:01:37,260 predicates can apply specific actions based on 30 00:01:37,260 --> 00:01:41,940 things like the input port and the packet header fields. The ability to compose 31 00:01:41,940 --> 00:01:45,500 policies in this fashion allows each module 32 00:01:45,500 --> 00:01:49,860 to partially specify functionality without having to write 33 00:01:49,860 --> 00:01:52,260 the policy for the entire network. This 34 00:01:52,260 --> 00:01:56,000 leaves some flexibility so that one module can 35 00:01:56,000 --> 00:01:57,750 implement a small bit of the network 36 00:01:57,750 --> 00:02:02,470 function, leaving some functions for other modules. This 37 00:02:02,470 --> 00:02:06,940 also allows for module re-use, since a module need not be tied to a particular 38 00:02:06,940 --> 00:02:10,758 network setting. For example, in this particular 39 00:02:10,758 --> 00:02:12,880 example where we've applied that load balancer 40 00:02:12,880 --> 00:02:15,910 followed by routing, the load balancer spreads 41 00:02:15,910 --> 00:02:18,740 traffic across the replicas without regard to the 42 00:02:18,740 --> 00:02:22,370 underlying network paths that traffic takes once 43 00:02:22,370 --> 00:02:26,210 those destination IP addresses are rewritten. In summary, 44 00:02:26,210 --> 00:02:32,070 we've covered two concepts. One is the notion of a Northbound API, which sits on 45 00:02:32,070 --> 00:02:38,750 top of an SDN controller and provides and exposes higher level abstractions that 46 00:02:38,750 --> 00:02:44,060 allows the operator or programmer to write policies without regard to how open 47 00:02:44,060 --> 00:02:46,490 flow rules eventually get installed. We've 48 00:02:46,490 --> 00:02:49,700 also talked about two different composition operators. 49 00:02:49,700 --> 00:02:53,920 Parallel composition and sequential composition, which specify 50 00:02:53,920 --> 00:02:57,990 how individual simpler policies can be composed 51 00:02:57,990 --> 00:03:03,100 to implement more complex network applications, thus allowing different SDN 52 00:03:03,100 --> 00:03:07,550 control programs to independently perform tasks on the same traffic.

24 - Pyretic Language - lang\_en\_vs4  
  
1 00:00:00,210 --> 00:00:02,450 In this lesson we will look at Pyretic 2 00:00:02,450 --> 00:00:06,450 which is an SDN language, and run time that 3 00:00:06,450 --> 00:00:09,860 implements some of the composition operators that we 4 00:00:09,860 --> 00:00:12,250 discussed in the last lesson. The language is a 5 00:00:12,250 --> 00:00:14,820 way of expressing these high level policies, and 6 00:00:14,820 --> 00:00:18,520 the run time provides the function of compiling these 7 00:00:18,520 --> 00:00:21,970 policies. So, the OpenFlow rules that are eventually are 8 00:00:21,970 --> 00:00:25,380 installed on the switches. One key abstraction in Pyretic 9 00:00:25,380 --> 00:00:31,360 is the notion of located packets. The idea that we can apply a policy based on 10 00:00:31,360 --> 00:00:34,590 a packet and it's location, in a network. 11 00:00:34,590 --> 00:00:37,020 Such as the switch, at which that packet is 12 00:00:37,020 --> 00:00:39,540 located or the port on which that packet 13 00:00:39,540 --> 00:00:44,290 arrives. Pyretic offers several features. The first is, the 14 00:00:44,290 --> 00:00:47,630 ability to take as input a packet, and 15 00:00:47,630 --> 00:00:51,220 then return packets at different locations in the network. 16 00:00:51,220 --> 00:00:54,270 This effectively allows the implementation of network 17 00:00:54,270 --> 00:00:58,320 policy as a function, that simply takes packets 18 00:00:58,320 --> 00:01:01,120 and returns other packets at different locations. 19 00:01:01,120 --> 00:01:03,130 The second feature of Pyredic is the notion 20 00:01:03,130 --> 00:01:06,190 of Boolean predicates. Unlike open flow rules, 21 00:01:06,190 --> 00:01:09,590 which do not permit the expression of simple 22 00:01:09,590 --> 00:01:12,800 conjunctions such as and, and or, or negations 23 00:01:12,800 --> 00:01:16,310 like not. Pyredic allows the expressions of policies 24 00:01:16,310 --> 00:01:19,420 in terms of these predicates. Pyredic also provides 25 00:01:19,420 --> 00:01:22,080 the notion of virtual package header fields. Which 26 00:01:22,080 --> 00:01:26,810 allows the programmer to refer to packet locations 27 00:01:26,810 --> 00:01:29,120 and also to tag packets so that specific 28 00:01:29,120 --> 00:01:32,050 functions can be applied at different portions of 29 00:01:32,050 --> 00:01:35,900 the program. Pyretic also provides composition operators, such 30 00:01:35,900 --> 00:01:39,150 as parallel and sequential composition, which we discussed 31 00:01:39,150 --> 00:01:41,440 in the last lesson. The notion of network 32 00:01:41,440 --> 00:01:45,002 policy as a function contrasts with the 33 00:01:45,002 --> 00:01:47,850 Open Flow style of programming. In OpenFlow, 34 00:01:47,850 --> 00:01:50,750 policies are simply bit patterns. In other 35 00:01:50,750 --> 00:01:54,550 words, match statements for which matching packets are 36 00:01:54,550 --> 00:01:56,990 subject to a particular action. These types 37 00:01:56,990 --> 00:01:59,190 of policies can be particularly difficult to 38 00:01:59,190 --> 00:02:02,890 reason about. In contrast, in Pyretic, policies 39 00:02:02,890 --> 00:02:06,690 are functions that map packets to other packets. 40 00:02:06,690 --> 00:02:08,910 Some example functions in Pyretic include the 41 00:02:08,910 --> 00:02:12,450 identify function, which returns the original packet, 42 00:02:12,450 --> 00:02:15,090 none or drop, which returns the empty 43 00:02:15,090 --> 00:02:18,850 set, match which returns the identity if the 44 00:02:18,850 --> 00:02:26,430 field f matches the value v and returns none or drop otherwise. Mod, which 45 00:02:26,430 --> 00:02:32,020 returns the same packet with the field f set to v. Forward, which is simply 46 00:02:32,020 --> 00:02:36,100 syntactic sugar on mod. To say that, the 47 00:02:36,100 --> 00:02:38,050 output port field in the packet should be 48 00:02:38,050 --> 00:02:42,470 modified to the parameter specified and flood which 49 00:02:42,470 --> 00:02:44,860 returns one packet for each port on the 50 00:02:44,860 --> 00:02:47,620 network spanning tree. In open flow, packets either 51 00:02:47,620 --> 00:02:49,840 match on a rule, or they simply fall 52 00:02:49,840 --> 00:02:53,600 through to the next rule. So, or, not, 53 00:02:53,600 --> 00:02:57,180 Ect can be tough to reason about. In contrast, 54 00:02:57,180 --> 00:03:00,090 peretics match function outputs either the 55 00:03:00,090 --> 00:03:03,620 packet or nothing, depending on whether the 56 00:03:03,620 --> 00:03:06,710 predicate is satisfied. For example, we could 57 00:03:06,710 --> 00:03:08,890 apply a match statement that says match 58 00:03:08,890 --> 00:03:12,190 destination IP equals ten zero zero three. 59 00:03:12,190 --> 00:03:14,680 And this function would take packets as 60 00:03:14,680 --> 00:03:19,340 input and only return packets that satisfy 61 00:03:19,340 --> 00:03:22,380 this particular predicate. In addition to the 62 00:03:22,380 --> 00:03:26,430 standard packet header fields, Pyretic offers the notion 63 00:03:26,430 --> 00:03:28,880 of virtual packet header fields, which is a 64 00:03:28,880 --> 00:03:32,290 unified way of representing packet metadata. In Pyretic, 65 00:03:32,290 --> 00:03:35,010 the packet is nothing more than a dictionary that 66 00:03:35,010 --> 00:03:37,480 maps a field name such as the destination 67 00:03:37,480 --> 00:03:40,570 IP address to a value. Now, these field names 68 00:03:40,570 --> 00:03:43,570 could correspond to fields in an actual packet 69 00:03:43,570 --> 00:03:48,120 header. But they can also be virtual. For example, 70 00:03:48,120 --> 00:03:50,270 we could provide a match statement based 71 00:03:50,270 --> 00:03:53,070 on a switch, indicating that we only want 72 00:03:53,070 --> 00:03:58,870 to return packets that are located at a particular switch or on the input port, 73 00:03:58,870 --> 00:04:01,090 indicating that we only want c packets 74 00:04:01,090 --> 00:04:04,080 whose attributes match a particular input port. The 75 00:04:04,080 --> 00:04:07,650 match function matches on this packet meta-data 76 00:04:07,650 --> 00:04:10,525 and the mod function can modify this meta-data.

25 - Composing Network Policies with Pyretic - lang\_en\_vs4  
  
1 00:00:00,410 --> 00:00:03,690 Pyretic enables the notion of both sequential and 2 00:00:03,690 --> 00:00:07,670 parallel composition as we've discussed in previous lessons. 3 00:00:07,670 --> 00:00:10,350 For example, we could match all packets for 4 00:00:10,350 --> 00:00:13,760 a particular destination IP address and send them or 5 00:00:13,760 --> 00:00:16,400 forward them out a particular output port. The 6 00:00:16,400 --> 00:00:19,200 double greater than sign shown here Is the 7 00:00:19,200 --> 00:00:23,060 way of expressing sequential composition in Pyretic. Parallel 8 00:00:23,060 --> 00:00:26,350 composition allows two policies to be applied in parallel. 9 00:00:26,350 --> 00:00:28,380 In this example, we match on a 10 00:00:28,380 --> 00:00:32,340 particular destination IP address and subsequently forward the 11 00:00:32,340 --> 00:00:35,270 traffic out Output Port one. In Parallel, 12 00:00:35,270 --> 00:00:37,000 we apply a different set of policies that 13 00:00:37,000 --> 00:00:39,860 match on a different source IP address. 14 00:00:39,860 --> 00:00:42,830 An output the packets on a different output 15 00:00:42,830 --> 00:00:46,590 port. In Pyretic, the plus operator performs 16 00:00:46,590 --> 00:00:51,480 parallel composition of policies. Pyretic allows an operator, 17 00:00:51,480 --> 00:00:55,530 to construct queries, which allow the program 18 00:00:55,530 --> 00:00:58,380 to see packet streams. For example, the 19 00:00:58,380 --> 00:01:01,280 packets query allows the operator to see 20 00:01:01,280 --> 00:01:05,500 packets, arriving at a particular switch with 21 00:01:05,500 --> 00:01:08,520 a particular source MAC address. The one 22 00:01:08,520 --> 00:01:10,710 parameter here indicates that we only want 23 00:01:10,710 --> 00:01:13,530 to see the first packet that arrives 24 00:01:13,530 --> 00:01:17,230 with a unique source MAC address and switch. 25 00:01:17,230 --> 00:01:20,250 We can then register callbacks for these packet streams. That are 26 00:01:20,250 --> 00:01:23,600 invoked to handle each new packet that arrives for that query.

26 - Dynamic Policies in Pyretic - lang\_en\_vs4  
  
1 00:00:00,480 --> 00:00:03,790 Dynamic policies are policies who's forwarding behavior can 2 00:00:03,790 --> 00:00:06,980 change. They are represented as a time series 3 00:00:06,980 --> 00:00:09,870 of static policies. The current value of the 4 00:00:09,870 --> 00:00:13,520 policy at any time is self dot policy. A 5 00:00:13,520 --> 00:00:16,190 common programming idiom in Pyretic is to set 6 00:00:16,190 --> 00:00:18,940 a default policy, and then register a call 7 00:00:18,940 --> 00:00:21,510 back that updates that policy. In the assignment, 8 00:00:21,510 --> 00:00:25,530 you will create a similar topology that you created 9 00:00:25,530 --> 00:00:27,630 in the pox assignment, but we will now 10 00:00:27,630 --> 00:00:31,330 use pyretic to implement a simple switch and firewall. 11 00:00:31,330 --> 00:00:34,300 In pyretic every first packet with a new 12 00:00:34,300 --> 00:00:37,160 source mac address, at the switch is read by 13 00:00:37,160 --> 00:00:40,230 a query. The policy is updated with a 14 00:00:40,230 --> 00:00:44,440 new predicate, everytime a new mapping between a MAC 15 00:00:44,440 --> 00:00:46,740 address and an output port is learnt. In 16 00:00:46,740 --> 00:00:50,640 the assignment you also create a dynamic firewall policy, 17 00:00:50,640 --> 00:00:53,050 register a callback to check the rules 18 00:00:53,050 --> 00:00:55,920 and sequentially compose your firewall policy with 19 00:00:55,920 --> 00:00:58,110 a learning switch, thus provided as part 20 00:00:58,110 --> 00:01:02,470 of the pyritic distribution. In summary, pyretic allows 21 00:01:02,470 --> 00:01:06,120 operators to write complex network policies as 22 00:01:06,120 --> 00:01:09,690 functions. It allows an operator to express predicates 23 00:01:09,690 --> 00:01:12,940 on packets, including things such as and, 24 00:01:12,940 --> 00:01:16,340 or, and not. It provides the capability to 25 00:01:16,340 --> 00:01:21,300 specify and modify virtual packet headers as packet metadata and it 26 00:01:21,300 --> 00:01:26,252 provides ways to compose complex network policies from simpler independent ones.

27 - Pyretic Policy Quiz - lang\_en\_vs4  
  
1 00:00:00,580 --> 00:00:06,206 As a quiz, which of the following is the appropriate pyretic rule for 2 00:00:06,206 --> 00:00:12,220 sending traffic from source IP address 10.0.0.1 to destination 3 00:00:12,220 --> 00:00:17,846 at IP address 10.1.2.3, and traffic from source IP address 4 00:00:17,846 --> 00:00:23,480 10.0.0.2 to destination IP address 10.2.3.4?

28 - Pyretic Policy Quiz Solution - lang\_en\_vs4  
  
1 00:00:00,290 --> 00:00:02,620 The following policy matches the appropriate source 2 00:00:02,620 --> 00:00:06,340 IP addresses and forwards to the corresponding 3 00:00:06,340 --> 00:00:08,870 output destination IP address. Each of these 4 00:00:08,870 --> 00:00:12,010 matching and forwarding operations, can happen, in parallel.

3 - SDN Programming Introduction - lang\_en\_vs4  
  
1 00:00:00,340 --> 00:00:04,019 Let's consider a network of SDN switches, such 2 00:00:04,019 --> 00:00:07,250 as open flow switches and a controller that 3 00:00:07,250 --> 00:00:09,340 is controlling those switches, and let's assume that 4 00:00:09,340 --> 00:00:13,760 we would like to write a program using this 5 00:00:13,760 --> 00:00:16,560 interface. We can think about this programming as 6 00:00:16,560 --> 00:00:19,440 proceeding in three steps. The first is that the 7 00:00:19,440 --> 00:00:23,300 controller needs to read, or monitor network state, 8 00:00:23,300 --> 00:00:26,020 as well as various events that may be occurring 9 00:00:26,020 --> 00:00:29,160 in the network. These events may include failures, topology 10 00:00:29,160 --> 00:00:32,960 changes, security events, and so forth. The second step 11 00:00:32,960 --> 00:00:37,240 is to compute the policy based on the state 12 00:00:37,240 --> 00:00:39,700 that the controller sees from the network. This is 13 00:00:39,700 --> 00:00:41,860 effectively what we talked about last time, is the 14 00:00:41,860 --> 00:00:44,390 role of the decision plane. In deciding what the 15 00:00:44,390 --> 00:00:47,470 forwarding behavior of the network should be, in response 16 00:00:47,470 --> 00:00:51,430 to various states that it reads from the network switches. 17 00:00:51,430 --> 00:00:54,440 The third step is to write policy 18 00:00:54,440 --> 00:00:57,830 back to the switches by installing the appropriate 19 00:00:57,830 --> 00:01:01,260 flow table state into the switches. Consistency 20 00:01:01,260 --> 00:01:04,220 problems can arise in two steps. First, the 21 00:01:04,220 --> 00:01:07,300 controller may read state from the network 22 00:01:07,300 --> 00:01:10,680 switches at different times, resulting in an inconsistent 23 00:01:10,680 --> 00:01:13,570 view of the network-wide state, and second, the 24 00:01:13,570 --> 00:01:16,630 controller may be writing policy as traffic is 25 00:01:16,630 --> 00:01:19,268 actively flowing through the network. Which can 26 00:01:19,268 --> 00:01:21,779 disrupt packets along an n to n path, 27 00:01:21,779 --> 00:01:25,586 or packets that should be treated consistently because 28 00:01:25,586 --> 00:01:29,020 they're part of the same flow. Both reading 29 00:01:29,020 --> 00:01:32,010 and writing networks state can be challenging because 30 00:01:32,010 --> 00:01:35,890 open flow rules are simple match action predicates, 31 00:01:35,890 --> 00:01:38,710 so it can be very difficult to express 32 00:01:38,710 --> 00:01:41,750 complex logic with these rules. If we want 33 00:01:41,750 --> 00:01:47,100 to read state that requires multiple rules, expressing 34 00:01:47,100 --> 00:01:51,990 a policy that allows us to read such a state can be complicated 35 00:01:51,990 --> 00:01:57,120 without more sophisticated predicates. For example, let's suppose that 36 00:01:57,120 --> 00:02:01,110 when we are reading state, we'd like to see all web serving traffic except for 37 00:02:01,110 --> 00:02:03,810 source one, two, three, four. Simple match 38 00:02:03,810 --> 00:02:07,010 action rules do not allow us to express 39 00:02:07,010 --> 00:02:10,080 such exceptions. As a solution to this 40 00:02:10,080 --> 00:02:13,230 problem, we need a language primitive that 41 00:02:13,230 --> 00:02:16,140 allows us to express predicates. Here is 42 00:02:16,140 --> 00:02:19,180 a simple statement that has several predicates; 43 00:02:19,180 --> 00:02:25,995 such as, and and not. A runtime system can then translate these predicates into 44 00:02:25,995 --> 00:02:28,980 low-level open flow rules, ensuring that they 45 00:02:28,980 --> 00:02:32,220 are installed atomically and in the right order. 46 00:02:32,220 --> 00:02:34,540 Another problem that arises is that switches only 47 00:02:34,540 --> 00:02:37,930 have limited space for rules. It's simply not 48 00:02:37,930 --> 00:02:41,960 possible to install all possible rule patterns for 49 00:02:41,960 --> 00:02:44,780 every set of flows that we'd like to monitor. 50 00:02:45,990 --> 00:02:48,750 For example, if we'd like to count the 51 00:02:48,750 --> 00:02:52,180 number of bytes for every source IP address, and 52 00:02:52,180 --> 00:02:54,560 generate a histogram with the resulting traffic, we 53 00:02:54,560 --> 00:02:57,430 would potentially need a flow table entry for every 54 00:02:57,430 --> 00:03:00,060 possible source IP address. It's simply not 55 00:03:00,060 --> 00:03:03,660 possible to install all of these possible rules. 56 00:03:03,660 --> 00:03:05,650 The solution is to have the run 57 00:03:05,650 --> 00:03:10,600 time system dynamically unfold rules as traffic arrives. 58 00:03:10,600 --> 00:03:12,930 A programmer would specify something like a 59 00:03:12,930 --> 00:03:15,780 group by source IP address, and the run 60 00:03:15,780 --> 00:03:19,390 time system would dynamically add open flow rules 61 00:03:19,390 --> 00:03:23,340 to the switch as traffic arrives. Thereby guaranteeing 62 00:03:23,340 --> 00:03:27,890 that there are only rules in the switch that correspond to active traffic.

4 - Reading Network State - lang\_en\_vs4  
  
1 00:00:00,390 --> 00:00:03,040 Another problem that arises when reading state, 2 00:00:03,040 --> 00:00:07,390 is that ,extra unexpected events may introduce 3 00:00:07,390 --> 00:00:11,420 inconsistencies. A common programming idiom, is that, 4 00:00:11,420 --> 00:00:14,220 the first packet goes to the controller 5 00:00:14,220 --> 00:00:19,840 and once the controller figures out what policy to apply for that flow, the 6 00:00:19,840 --> 00:00:23,790 controller then installs rules in the switches, 7 00:00:23,790 --> 00:00:26,540 in the network, corresponding to that flow. 8 00:00:26,540 --> 00:00:29,390 What if more packets should arrive ,at the 9 00:00:29,390 --> 00:00:33,230 switch before the controller has a chance to install 10 00:00:33,230 --> 00:00:36,050 rules for that flow? At this point, multiple 11 00:00:36,050 --> 00:00:40,170 packets may reach the controller, but the application It 12 00:00:40,170 --> 00:00:42,270 is running on top of the controller, may 13 00:00:42,270 --> 00:00:45,600 not need or want to see these additional packets. 14 00:00:45,600 --> 00:00:48,940 So, the solution, is to have the programmer specify 15 00:00:48,940 --> 00:00:52,410 by a high level language a limit of one, 16 00:00:52,410 --> 00:00:55,320 indicating that the application should only see 17 00:00:55,320 --> 00:00:57,590 the first packet of the flow and that 18 00:00:57,590 --> 00:01:00,330 the subsequent packet should be suppressed. The 19 00:01:00,330 --> 00:01:03,690 runtime system then hides the extra events. So 20 00:01:03,690 --> 00:01:05,570 to remind you where we are, we 21 00:01:05,570 --> 00:01:09,240 talked about problems with consistency when reading state 22 00:01:09,240 --> 00:01:12,495 from the network, and we talked about ,three 23 00:01:12,495 --> 00:01:17,790 approaches to, helping guarantee consistency when reading state. 24 00:01:17,790 --> 00:01:21,270 Predicates, rule unfolding and suppression. And let's now talk 25 00:01:21,270 --> 00:01:25,090 about primitives that can help maintain consistency, when writing State.

5 - Writing Network Policy - lang\_en\_vs4  
  
1 00:00:00,140 --> 00:00:02,100 There are many reasons that a controller 2 00:00:02,100 --> 00:00:04,950 might want to write policy, to change the 3 00:00:04,950 --> 00:00:09,420 state and the network switches, including maintenance, unexpected 4 00:00:09,420 --> 00:00:12,260 failure, and traffic engineering. Any of these network 5 00:00:12,260 --> 00:00:15,090 tasks involve or require updating state in the 6 00:00:15,090 --> 00:00:18,490 network switches, and when that state transition happens, 7 00:00:18,490 --> 00:00:21,820 we want to make sure that forwarding remains 8 00:00:21,820 --> 00:00:25,460 correct and consistent. In particular, we would like 9 00:00:25,460 --> 00:00:28,650 to maintain the following in variance: there shouldn't be any 10 00:00:28,650 --> 00:00:32,310 forwarding loops and there shouldn't be any black holes whereby 11 00:00:32,310 --> 00:00:34,690 a router or switch receives a packet and doesn't know 12 00:00:34,690 --> 00:00:36,990 what to do with it. There also shouldn't be cases 13 00:00:36,990 --> 00:00:39,710 where traffic is going where it shouldn't be allowed to 14 00:00:39,710 --> 00:00:43,630 go because of the network being in an inconsistent state. 15 00:00:43,630 --> 00:00:48,090 Let's now consider an example of what might happen when 16 00:00:48,090 --> 00:00:50,600 policies are written to the network, if they're written in 17 00:00:50,600 --> 00:00:54,500 an inconsistent fashion. Let's consider a case where 18 00:00:54,500 --> 00:00:57,880 we have a network that is performing shortest routing 19 00:00:57,880 --> 00:01:00,860 to some destination. And the link weights are 20 00:01:00,860 --> 00:01:03,790 as at shown here in the figure. Traffic in 21 00:01:03,790 --> 00:01:05,810 the network would flow along a path shown 22 00:01:05,810 --> 00:01:09,740 in green. Let's suppose now that an operator wants 23 00:01:09,740 --> 00:01:16,560 to change the network state to shift traffic off of this link. He could do so by 24 00:01:16,560 --> 00:01:20,840 updating the link weight. In doing so, the new shortest path from this top 25 00:01:20,840 --> 00:01:28,780 router would be as follows. But, what if the state in the top switch 26 00:01:30,650 --> 00:01:36,420 occurred before the state in the bottom switch could be updated? In this case, 27 00:01:36,420 --> 00:01:38,910 we would have a potential forwarding loop. 28 00:01:38,910 --> 00:01:41,320 Traffic would proceed to the bottom switch. 29 00:01:41,320 --> 00:01:43,190 But the bottom switch would still have the 30 00:01:43,190 --> 00:01:47,060 old network state, and would continue to forward traffic 31 00:01:47,060 --> 00:01:50,420 to the top switch, resulting in a forwarding loop. 32 00:01:50,420 --> 00:01:52,940 If rules are installed along a path out of 33 00:01:52,940 --> 00:01:56,600 order, packets may reach a switch before the 34 00:01:56,600 --> 00:02:00,130 new rules do. So, in this type of model 35 00:02:00,130 --> 00:02:02,460 we would have to think about all possible packet 36 00:02:02,460 --> 00:02:07,850 and event orderings to ensure that consistent behavior resulted. 37 00:02:07,850 --> 00:02:10,169 So we need atomic updates of the entire 38 00:02:10,169 --> 00:02:12,990 configuration. The solution to this problem is to use 39 00:02:12,990 --> 00:02:17,000 a two phase commit so that packets are either 40 00:02:17,000 --> 00:02:20,940 subjected to the old configuration on all switches, or 41 00:02:20,940 --> 00:02:23,670 to the new configuration on all switches. But 42 00:02:23,670 --> 00:02:26,400 packets aren't subjected to the new policy on some 43 00:02:26,400 --> 00:02:29,780 switches and the old policy on others. The idea 44 00:02:29,780 --> 00:02:33,480 is to tag the packet on ingress so that 45 00:02:33,480 --> 00:02:39,660 the switches maintain copies of both P1 and P2 for some time. When all 46 00:02:39,660 --> 00:02:43,000 switches have received rules corresponding to the 47 00:02:43,000 --> 00:02:46,460 new policy, then incoming packets can be tagged 48 00:02:46,460 --> 00:02:52,120 with P2. After some time, when we're sure that no more packets with P1 49 00:02:52,120 --> 00:02:53,980 are being forwarded through the network, we 50 00:02:53,980 --> 00:02:57,990 can only then remove the rules corresponding to 51 00:02:57,990 --> 00:03:01,075 policy P1. Now, the naive version of 52 00:03:01,075 --> 00:03:03,580 two-phase commit, requires doing this on all 53 00:03:03,580 --> 00:03:06,450 switches at once. Which essentially doubles the 54 00:03:06,450 --> 00:03:09,870 rule space requirements, since we have to store 55 00:03:09,870 --> 00:03:15,880 the rules for both P1 and P2. We can limit the scope of the two phase commit by 56 00:03:15,880 --> 00:03:23,000 only applying this mechanism on switches that involve the affected portions of 57 00:03:23,000 --> 00:03:25,840 the traffic, or the affected portions of the topology.

6 - Inconsistent Policy Write Quiz\_1 - lang\_en\_vs4  
  
1 00:00:00,550 --> 00:00:05,310 So here's a quick quiz. What types of problems can arise from inconsistent 2 00:00:05,310 --> 00:00:11,210 applications of writing policy? Inability to respond to failures, forwarding 3 00:00:11,210 --> 00:00:16,110 loops, a flood of traffic at the controller, or security policy violations?

7 - Inconsistent Policy Write Quiz\_1 Solution - lang\_en\_vs4  
  
1 00:00:00,750 --> 00:00:03,740 Inconsistent writes can result in forwarding loops or 2 00:00:03,740 --> 00:00:06,925 security policy violations where traffic ends up going to 3 00:00:06,925 --> 00:00:09,395 parts of the network where it shouldn't go as 4 00:00:09,395 --> 00:00:12,650 a result of inconsistent switch state. The ability to 5 00:00:12,650 --> 00:00:15,980 respond to failures is orthogonal to consistency. A 6 00:00:15,980 --> 00:00:19,160 flood of traffic at the controller technically involves problems 7 00:00:19,160 --> 00:00:22,120 with reading state in a consistent fashion. But since 8 00:00:22,120 --> 00:00:26,220 there also involves a step where the controller writes 9 00:00:26,220 --> 00:00:29,720 state to the switches, while packets are still arriving at the 10 00:00:29,720 --> 00:00:32,890 controller, I would consider that answer to be correct as well.

8 - Coping With Inconsistency Quiz - lang\_en\_vs4  
  
1 00:00:00,340 --> 00:00:02,050 What are some approaches to coping with 2 00:00:02,050 --> 00:00:07,060 inconsistency? Running different controllers for different switches, keeping 3 00:00:07,060 --> 00:00:09,350 a hot spare replica that has a complete 4 00:00:09,350 --> 00:00:11,650 view of the network state, keeping the old 5 00:00:11,650 --> 00:00:13,790 and new state on the routers and switches 6 00:00:13,790 --> 00:00:16,820 and switching over only when all of the 7 00:00:16,820 --> 00:00:20,760 switches have received the new state Or relying 8 00:00:20,760 --> 00:00:23,250 on the routers themselves to resolve the conflict.

9 - Coping With Inconsistency Quiz Solution - lang\_en\_vs4  
  
1 00:00:00,460 --> 00:00:02,740 In this case, there is only one correct answer, which 2 00:00:02,740 --> 00:00:04,939 is keeping the old and new state on the routers 3 00:00:04,939 --> 00:00:07,650 and switches. This is the two-phase commit approach that we 4 00:00:07,650 --> 00:00:09,630 talked about. Running different controllers 5 00:00:09,630 --> 00:00:11,400 for different switches could obviously 6 00:00:11,400 --> 00:00:14,630 result in an inconsistent state, since each of those controllers 7 00:00:14,630 --> 00:00:18,630 maybe making independent decisions. Keeping a hot spare replica does 8 00:00:18,630 --> 00:00:22,340 no good if the replica also writes state inconsistently to 9 00:00:22,340 --> 00:00:25,460 the network. And resolving conflicts on the routers also doesn't work 10 00:00:25,460 --> 00:00:28,380 because no router has a complete view of the network state.