Mininet pingall timeout

Sometimes simulations are not feasible or practical, and network experiments must be run on actual machines. You can always use a set of interconnected virtual machines, but even steam VMs consume sufficient resources that are difficult to create a network with more than a handful of nodes. Mininet is a system that supports the creation of lightweight logical nodes that can be connected to networks. These nodes are sometimes called containers or, more specifically, network namespaces. Virtual machine technology is not in use. These containers consume quite a few resources that have been created by networks of more than a thousand nodes running on a single laptop. Although mininet was originally designed as a test for networking software (3.4 Software-Defined Networking), it works just as well for demonstrations and experiments involving traditional networks. The Mininet container is a process (or process group) that no longer has access to all native network interfaces of the host system, just as the process that executed the chroot() system call no longer has access to the full file system. Mininet containers are then assigned virtual Ethernet interfaces (see ip link page entries for veth) that are connected to other containers using virtual Ethernet links. Using veth links ensures that virtual links behave like Ethernet, although it may be necessary to disable TSO (17.5 TCP Offload) to view ethernet packages in WireShark as they will appear on (virtual) wire. Any process started in the Mininet container inherits the network interface container view. For efficiency, Mininet containers share the same file system by default. This makes the setting simple, but sometimes causes problems with applications waiting for individualized configuration files in the specified locations. Mininet containers can be configured using different views of the file system, although we will not do so here. Mininet is a form of network emulation, as opposed to modeling. An important advantage of emulation is that all network software, on any layer, simply runs as is. In a simulator environment, on the other hand, applications and protocol implementations must be ported to run within the simulator before they can be used. The downside of emulation is that as the network becomes large and complex emulation can slow down. In particular, it is impossible to emulate the connection speed faster than the base equipment can support. (It is also impossible to emulate network software that is not associated with Linux.) Mininet Group keeps extensive documentation; the three useful starting places are Overview, Introduction, and FAQ. The purpose of this section is to present a series of examples of Mininet. Most examples as an offline file (Mininet does not currently support Python3.) Python3). The Mininet Python2 file configures the network and then runs the Mininet command line interface (which is required to run commands on different node containers). Using offline Python files may make it easier to edit configurations and avoids complex command-line arguments for many standard Mininet examples. The Python code uses what Mininet documentation calls mid-level APIs. The Mininet distribution comes with its own set of examples in the directory of that name. Some of the special interests are listed below; Except for linuxrouter.py examples presented here do not use any of these techniques. bind.py: demonstrates how to give each Mininet node its own private directory (otherwise all nodes share a common file system) controllers.py: demonstrates how to organize multiple SDN controllers, with different connection switches to different controllers limit.py: demonstrates how to set cpu usage limits (and link bandwidth) linuxrouter.py: creates a node that acts as a router. Any host node can act as a router, although, provided we allow forwarding from sysctl net.ipv4.ip_forward=1 miniedit.py: graphics editor for Mininet mobility.py networks: demonstrates how to move a host from one switch to another nat.py: demonstrates how to connect hosts to the Internet tree1024.py: creates a network with 1024 nodes From time to time we will need additional applications, for example, to send, monitor or receive traffic. They are designed to change as necessary to meet the circumstances; they contain several command line options. Most of these additional programs are written, perhaps confusingly, in Python3. Python2 files are started using the python command and the Python3 command is python3. In addition, given that all these programs run under Linux, you can make all python files executable and make sure that the first line is either #!/usr/bin/python or #!/usr/bin/python3 respectively. Mininet only works under the Linux operating system. Windows and Mac users can, however, easily run Mininet in a single Linux virtual machine. Even Linux users may want to do so because running Mininet has non-trivial potential to affect normal operation (the virtual switch process running Mininet, for example, interfered with the pause function on the author's laptop). Mininet group supports the virtual machine with the current installation of Mininet on the download site. The download file is actually a .zip file that unpacks a modest .ovf file that identifies the technical characteristics of the virtual machine and a much larger (~2 GB) .vmdk file representing the virtual disk image. (Some unboxing versions have trouble unpacking very large files; if that happens, search the internet for alternative unboxing.) There are several virtual machine software; two options that are well supported and (as of 2017) for personal use are VirtualBox and VMware Workstation Player. The .ovf file should open in any (in VirtualBox with the option to import the device). However, it may be easier to simply create a new Linux virtual machine and indicate that it is to use an existing virtual disk; then select the downloaded .vmdk file as this drive. Both the login name and password for the virtual machine are mininet. Once logged in, the sudo command can be used to obtain administrator privileges that are required to run Mininet. This is the safest for this on a team basis; for example, sudo python switchline.py. You can also keep the terminal window open, which is constantly logged in as root, for example via sudo bash. Another option is to configure a Linux virtual machine from scratch (for example through an Ubuntu distribution) and then install a Mininet on it, although the preinstalled version also comes with other useful software such as the Pox controller for OpenFlow switches. However, the preinstalled version does not come with any desktop GUI. You can install ubuntu's full desktop using the command (as root) apt-get to install Ubuntu-desktop. This, however, will add more than 4 GB to the virtual drive. A lighter option recommended by the Mininet site is to install an alternative lxde desktop environment; it's half the size of Ubuntu. Install it using apt-get install xinit lxde The standard graphical text editor included in lxde is a leafpad, although of course others (such as gedit or emacs) can also be installed. After installing the desktop, the startx command will be required after logging in to run the graphics environment (although you can automate this). The standard recommendation for new Debian-based Linux systems before you install anything else is apt-get update apt-get update Most virtual machine software offers a special package to improve compatibility with the host system. One of the most annoying incompatibilities is the virtual machine's tendency to grab a mouse and not let it drag outside the virtual machine window. (Usually a special keystroke frees the mouse; on VirtualBox it's the right control key, and on VMWare Player it's Control-Alt.) Installing a compatibility pack (in VirtualBox called Guest Additions) usually requires mounting a CD image, with mount /dev/cdrom /media/cdrom The very installation of mininet can be upgraded as follows: cd /home/mininet/mininet git fetch git checkout master # Or a specific version like 2.2.1 git pull to make the installation the easiest environment for beginners is to install a graphics desktop (eg lxde) and then work in it. This allows you to open xterm and WireShark seamlessly in moderation Enabling copy/paste between virtual system and host is also handy. However, you can also work completely without a desktop by using ssh logins with X-windows redirect enabled: ssh -X -l mininet This requires an X server on the host system, but they are available even for Windows (see, for example, Cygwin/X). At this point, you can open the graphics program at the ssh command prompt, such as wireshark & or gedit mininet-demo.py &, and have the application window displayed correctly (or close to proper). Finally, access to the Mininet virtual machine can be accessed exclusively through ssh terminal sessions, with no X windows, although one then cannot run xterm or WireShark. Running Mininet using the mn command (from 1 root!), without command line arguments, creates a simple network of two hosts and one switch, h1–s1–h2, and runs the Mininet command line interface (CLI). Under the convention, Mininet host names start with h, and switch names start with s; numbering starts at 1. At this point, you can issue different Mininet-CLI commands. For example, command nodes give the following output: available nodes: c0 h1 h2 s1 Node c0 is the controller for the s1 switch. The default controller action causes s1 to behave like an Ethernet training switch (Ethernet 2.4.1 learning algorithm). The intfs command lists the interfaces for each node and the link lists the connections but the most useful command is net, which shows nodes, interfaces and connections: h1-eth0:s1-eth1 h2-eth0:s1-eth2 s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0 From the above we can see that the network is as follows: The next step is to run commands on separate nodes. To do this, we use Mininet CLI and the command name prefix with hostname: The first command here shows that h1 (or, more correctly, h1-eth0) has an IP address of 10.0.0.1. Note that the name h2 in the second is recognized. The ifconfig command also shows the MAC address h1-eth0, which can change but can be something like 62:91:68:bf:97:a0. We'll see in the next section how to get more human readable MAC addresses. There is a special Mininet pingall command that generates pings between each pair of hosts. We can open a full shell window on h1 using the Mininet command below; this works for both host nodes and switch nodes. Note that xterm is running as an administrator. With xterm, the ping h2 command now fails as the h2 hostname is not recognized. We can switch to ping 10.0.0.2, or add entries to /etc/hosts for IP addresses h1 and h2: Since the Mininet system shares its file system with h1 and h2, it means that the names h1 and h2 are now defined everywhere in Mininet (although it will be warned that when another Mininet configuration assigns different addresses to h1 or h2, chaos will be as follows). With xterm on h1 we can try to enter h2 via ssh: ssh h2 (if h2 is defined in / etc / hosts as above). But connection denied: ssh server does not work node h2. We'll get back to that in the next example. We can also launch WireShark, and listen to it on the h1-eth0 interface, and see the progress of our pings. (We can also usually start WireShark from a mininet > using h1 wireshark &) Similarly, we can start xterm on the switch and start WireShark there. However, there is another option, like the default switches, to share all of their network systems with a mininet host system. (In terms of container model, default switches don't get their own network namespace;

they share the root namespace with the host.) We can see this by running the following from the Mininet command line and comparing the output to ifconfig running on the Mininet site, while mininet works, but outside the Mininet process itself. We see these interfaces: eth0 lo s1 s1-eth1 s1-eth2 We see the same interfaces on the c0 controller node, even if the network commands and intfs above did not show interfaces for c0. Launching WireShark on, say, s1-eth1 is a great way to observe traffic on an almost non-functioning network; By default, Mininet nodes are not connected to the outside world. As an example, suppose we run xterm windows on h1 and h2, and run netcat -l 5432 on h2, followed by netcat 10.0.0.2 5432 on h1. We can then watch the ARP exchange, the TCP handshake, content delivery and connection break, most likely there's no other traffic at all. Wireshark filtering is not required. The following example creates the topology below. All hosts are on the same subnet. Mininet-CLI command links determine which switch interface the adjacent switch interface is connected to. Full Python2 switchline.py; to start it This option configures the network and starts the CLI mininet. The default number of node/switch pairs is 4, but this can be changed using the -N command line option, such as python switchline.py -N 5. Next, we describe the selected parts of switchline.py. The program starts by creating a network topology object, LineTopo, extending the built-in Mininet Topo class, and then call Topo.addHost() to create host nodes. (We'll __init()__, but overriding build() is actually more common.) LineTopo(Topo): def __init__(self, **kwargs): Create linear topology super(LineTopo, self).__init__(**kwargs) h = [] # host list; h[0] will h1, etc= [] # list of switches for keys in kwargs: if key == 'N': N=kwargs[key] # add N hosts h1.. hN for i in the range (1,N+1): h.addend(self.addHost('h' + str(i))) The Topo.addHost() method accepts a string, such as h2, and builds a host object of that name. We will immediately add a new host object to the h[]. Next, we do the same for switches using Topo.addSwitch(): # add N switches s1.. sN for i in the range (1,N+1): + str(i))) Now we are building a link, with Topo.addLink. Topo.addLink. that h[0].. h[N-1] represent h1.. Hn. First we create a link to the host switch and then the switch link. for me in the range (N): # Add a link with hello si self.addLink(h[i], s[i]) for me in the range (N-1): # link switches self.addLink(s[i],s[i+1]) Now we go to the main program. We use argparse to support the command line argument -N. def main(**kwargs): parser = argparse. ArgumentParser() parser.add_argument('-N', '-N', type=int) args= parser.parse_args() if args. N is none: N = 4 more: N = args. N Next, we create the LineTopo object defined above. We have also set the log level on the information; if we had problems, we would set it up for debugging. ltopo = LineTopo (N=N) setLogLevel ('info') Finally we are ready to create a clean Mininet object and run it. We've specified the type of switch here, although at the moment it doesn't matter. It is important that we use DefaultController because otherwise the switches will not behave automatically as Ethernet learning switches. The autoSetMacs parameter sets the host MAC address to 00:00:00:00:00:01 to 00:00:00:00:00:04 (for N=4), which can be a great convenience when manually reviewing Ethernet addresses. net = Mininet(topo = ltopo, switch = OVSKernelSwitch, controller = DefaultController, autoSetMacs = True) net.start() The next bit starts /usr/sbin/sshd on each node. This command automatically puts itself in the background; otherwise, we'll need to add & to the line to run the command in the background. for i in the range (1, N +1): hello =net['h' + str(i)] hi.cmd('/usr/sbin/sshd') Finally we start Mininet CLI and, when it comes out, we stop emoding. Using sshd requires a little bit of tweaking if ssh for the root user is not already configured. We must first run ssh-keygen, which creates the /root/.ssh directory, followed by public and private keys files, id_rsa.pub and id_rsa respectively. There is no need in this option to protect the keys with a password. The second step is to go to the .ssh directory and copy id_rsa.pub to a (new) authorized_keys file (if the last file already exists, add id_rsa.pub to it). This will allow a password-free ssh connection between different Mininet hosts. Since we started sshd on each guest, the command ssh 10.0.0.4 on h1 should successfully connect to h4. The first time a connection is made from h1 to h4 (from root), ssh will ask for confirmation and then save the h4 key to /root/.ssh/known_hosts. Since this is the same file for all Mininet nodes, thanks to the common file system, further request to connect from h2 to h4 will be possible immediately; h4 has already been authenticated for all nodes. Now let's run the web server on, say, host 10.0.0.4 switchline.py above. Python includes a simple implementation that files in the directory in which it was started. After switchline.py works, works, xterm to h4, and then change the directory to /usr/share/doc (where there are some HTML files). Then run the following command (8000 is the server port number): python -m SimpleHTTPServer 8000 If it is somewhere running in the background, the output should be redirected to /dev/null, otherwise the server eventually freezes. The next step is to start the browser. If the lxde environment is installed (30.1 Mininet Installation), then chrome browser should be available. Run xterm on the h1 guest, and on h1 run the following (parameter --no-sandbox required to run chromium from 1 root): chrome browser --no-sandbox Assuming chrome opens successfully, enter the following URL: 10.0.0.4:8000. If chrome does not run, try wget 10.0.0.4:8000, which stores what it receives as an index.html file. In any case, you should see the /usr/share/doc directory list. Several directories with subdirecologists named html are iperf, iptables and xarchiver; try navigating to them. In the following example, we create a Mininet example involving a router rather than a switch. The router here is just a multifunctional Mininet host that has IP redirection enabled in the Linux kernel. Support for mininets for hosts from multiple interfaces is somewhat fragile; interfaces may need to be initialized in a specific order, and IP addresses often cannot be assigned at the time the link is created. In the code presented below, we assign IP addresses using node.cmd() calls used to call the Linux ifconfig command (Mininet containers do not fully support the use of an alternative IP-adver command). Our first router topology has only two hosts, one at each end, and N-routers between them; the following is a diagram with N=3. All subnets /24. The software to configure this is routerline.py here is called as python routerline.py -N 3. We will use N=3 in most examples below. A slightly simpler version of the program, which installs topology specifically for N=3, is routerline3.py. In both versions of the program, routing records are created to route traffic from h1 to h2, but are not returned again. That is, each router has a route to 10.0.3.0/24, but only r1 knows how to reach 10.0.0.0/24 (to which r1 is directly connected). We can check one way connection by running WireShark or tcpdump on h2 (possibly first running xterm on h2) and then running ping 10.0.3.10 on h1 (possibly using the command Mininet h1 ping h2). WireShark or tcpdump should display arriving ICMP ping packets with h1, and arriving ICMP destination network unavailable packets with r3 as h2 tries to respond (see protocol protocol 10.4 of Internet message management). It turns out that routing into one considered suspicious; one interpretation is that the packages involved have a source of this should not be possible, perhaps multiplying. Linux provides a configuration option for rp_filter interface - a backtrace filter - to block packet forwarding for which the router has no route back to the package source. This should be disabled for such an example to work with; see the notes in the code below. Despite the lack of connection, we can reach h2 with h1 using the sequence of ssh connections (the software allows sshd on each guest and router): h1: slogin 10.0.0.2 r1: slogin 10.0.1.2 r2: slogin 10.0.2.2 p3: slogin 10.0.3.10 (i.e., h3) To get one-way routing to work from h1 to h2, We needed to say r1 and r2 how to get to your destination 10.0.3.0/24. This can be done with the following commands (which are executed automatically if we set ENABLE_LEFT_TO_RIGHT_ROUTING = True in the program): r1: ip route add to 10.0.3.0/24 via 10.0.1.2 r2: ip route add to 10.0.3.0/24 via 10.0.2.2 To get full, bidirectional connectivity, we can create the following routes to 10.0.0.0/24: r2: ip route add to 10.0.0.0/24 via 10.0.1.1 r3: ip route add to 10.0.0.0/24 via 10.0.2.1 When building the network topology, the single-interface hosts can have all their attributes set at once (the code below is from routerline3.py: h1 = self.addHost('h1', ip='10.0.0.10/24', defaultRoute='via 10.0.0.2') h2 = self.addHost('h2', ip='10.0.3.10/24', defaultRoute='via 10.0.3.1') The routers are also created with addHost(), but with separate steps: r1 = self.addHost('r1') r2 = self.addHost('r2') ... self.addLink(h1,r1,intfName1='h1-eth0', intfName2='r1-eth0') self.addLink(r1, r2, inftname1 = 'r1-eth1', inftname2 = 'r2-eth0') Later on routers get their IPv4 addresses: r1 = net['r1'] r1.cmd('ifconfig r1--eth0 10.0.0.2/24') r1.cmd('ifconfig r1-eth1 10.0.1.1/24') r1.cmd('sysctl net.ipv4.ip_forward=1') rp_disable(r1) The sysctl command here allows you to forward to r1. Calling rp_disable(r1) prevents Linux from sending packages by default if the router does not have a route back to the package source; this is often what you want in the real world, but not necessarily in routing demonstrations. This is also ultimately implemented using sysctl commands. The next step will be to automate the opening of the route from h1 to h2 (and back) using a simple distance routing and update protocol. We present the partial implementation of the RIP information routing protocol as defined in RFC 2453. Vector distance algorithm is described in algorithm 13.1 Distance vector routing update. In short, the idea is to add a cost attribute to the forwarding table, so the records have a form ⟨destination, next_hop, cost⟩. Routers then send ⟨, cost⟩ lists for their neighbors; these lists are referred to by the RIP specification as an update notice. Routers receive these messages, then process them to find out the lowest route to the Destination. The update notification format is as follows: RIP's full specification also includes message requests, but the implementation here does not contain them. The full specification also includes a split horizon, poison reverse and triggered updates (13.2.1.1 Split Horizon and 13.2.1.2 Triggered updates); we also provise them. Finally, while we include code for third case increase next_hop 13.1.1 Distance-Vector upgrade rules, we don't include any test on whether the link is down, so this case never works. The implementation is located in the Python3 file rip.py. Most of the time the program waits to read the update notification from other routers. Every UPDATE_INTERVAL, the program sends its own update messages. All messages through UDP packets are sent by using IP multi-forward, official RIP multi-forward address 224.0.0.9. Port 520 is used for both send and receive. Instead of creating separate themes to receive and send, we set up a short (1 second) recv() timeout, and then after each timeout, we check to see if it's time to send the next update. The update may be up to 1 second late with this approach, but it doesn't matter. The software supports a shadow copy of the RTable real system forwarding table, with an additional value column. The real table is updated every time the route changes in the shadow table. RTable is a dictionary that maps TableKey (consisting of an IP address and mask) to TableValue objects containing the interface name, cost, and next_hop. To run the program, the production approach would be to use Node.cmd() Mininet to run rip.py on each router, for example through r.cmd('python3 rip.py &') (assuming the rip.py file is in the same directory in which mininet was running). For demos, the output of the program can be observed if the program started in xterm on each router. Sending ip multi-cross includes special considerations that do not occur with TCP or UDP connectivity. The first issue is that we ship to a multicast group - 224.0.0.9 - but don't have any multicast routes (multicast trees, 25.5 Global IP Multicast) configured. What we would like to have, on each router, traffic up to 224.0.0.9 diverted to each of the adjacent routers. However, we don't really want to set up multi-route routes; All we want is to reach our immediate neighbors. Tweaking a multi-tree assigns that we know something about network topology and, at the point where RIP comes into play, we don't. The multi-door packages we send don't really have to be forwarded by neighbors (we'll provide this below by installing TTL); the multi-role model here is very local. Even if we configure multi-route, Linux does not provide a standard utility for manual multi-route routing configuration; see ip-mroute.8 man. Yes Yes instead, we create a socket for each individual interface of the router and configure the socket so that it forwards its traffic only from the associated interface. This introduces complications: we need to get a list of all interfaces and then, for each interface, get related IPv4 addresses with network applications. (To make life easier, we assume that each interface has only one IPv4 address.) The getifaddrdict() function returns a dictionary with interface names (strings) as keys and pairs (ipaddr, netmask) as a value. If ifaddrs is a dictionary, for example, ifaddrs['r1-eth0'] can be ('10.0.0.2','255.255.255.0'). We could implement getifaddrdict() simply using the Python netifaces module, although for demonstration purposes we do this here with low-level system calls. We get a list of interfaces using myInterfaces = os.listdir('/sys/class/net/'). For each interface, we will get its IP address and netmask (in get_ip_info(intf)) with the following: s =socket.socket(socket.AF_INET, socket. SOCK_DGRAM) SIOCGIFADDR = 0x8915 # from / usr / include / linux / sockios.h SIOCGIFNETMASK = 0x891b intfpack = struct.pack('256s', bytes(intf, 'ascii')) #ifreq, below, similar to struct ifreq in /usr/include/linux/if.h ifreq=fcntl.ioctl(s.fileno(), SIOCGIFADDR, intfpack) ipaddrn = ifreq[20:24] #20 is an IP adder offset in ifreq ipaddr =socket.inet_ntoa (ipaddrn) netmaskn=fcntl.ioctl(s.fileno(s.fileno()), SIOCGIFNETMASK, intfpack)[20:24] netmask = socket.inet_ntoa(netmaskn) return (ipaddr, netmask) We need to create an outlet here (never connected) in order to call ioctl(). The SIOCGIFADDR and SIOCGIFNETMASK values are from the C language containing the file; Python3 libraries do not make these constants available, but the Python3 fcntl.ioctl() call transmits the values we provide directly to the main C ioctl() call. This call returns its result in C. struct ifreq; ifreq above is the Python version of this. Binary format IPv4 address (or grid) offset 20. We are now in a position for each interface to create a UDP connector that will be used to send and receive on this interface. Most of the information here comes from the pages of Linux socket.7 and ip.7 man. The createMcastSockets (ifaddrs) function takes the dictionary above the name mapping interface (ipaddr, netmask) pairs and, for each intf interface, configures it as follows. Then a list of all newly configured connectors is returned. The first step is to get the interface address and mask, and then convert them to a 32-bit whole format like ipaddrn and netmaskn. Then we enter the subnet corresponding to the interface in the RTable shadow routing table with a cost of 1 (and with next_hop none), via RTable[TableKey(subnetn, netmaskn)] = TableValue(intf, None, 1) Next we create a socket and start it first setting its read timeout to short short Then we set the TTL value used by the original batches to 1. This occurs in the header of the IPv4 Time Live field (9.1 IPv4 Header); this means that no downstream router will ever forward the package. This is exactly what we want; RIP uses multi-divert only to send to nearby neighbors. Sock.setsockopt(socket. IPPROTO_IP, socket. IP_MULTICAST_TTL, 1) We also want to be able to bind the same socket source address, 224.0.0.9 and port 520, to all sockets that we create here (actual bind() call below): sock.setsockopt(socket. SOL_SOCKET, socket. SO_REUSEADDR, 1) The next call forces the socket to receive only packages coming to the specified interface: sock.setsockopt(socket. SOL_SOCKET, socket. SO_BINDTODEVICE, bytes(intf, 'ascii')) We add the following to prevent packets from being delivered to the sender interface; otherwise multi-door delivery can do just that: sock.setsockopt(socket. IPPROTO_IP, socket. IP_MULTICAST_LOOP, False) The next call forces the socket to be sent to the specified interface. Multi-role packets have IPv4 destination addresses, and typically the kernel selects a send interface based on the IP forwarding table. This challenge has the attention that, in fact, telling the core how to route packages sent through this socket. (The kernel may also be able to figure out how to route the package with the subsequent call joining socket multi-adress group.) Sock.setsockopt(socket. IPPROTO_IP, socket. IP_MULTICAST_IF, socket.inet_aton (ipaddr)) Finally, we can join the multi-adress group outlet, introduced 224.0.0.9. We also need an IP address interface, an ipaddr. addrpair = socket.inet_aton('224.0.0.9')+ socket.inet_aton (ipaddr) sock.setsockopt(socket. IPPROTO_IP, socket. IP_ADD_MEMBERSHIP, addrpair) The last step is to bind the socket to the correct address and port, with sock.bind(('224.0.0.9', 520)). This specifies the original address of the original batches; it will not succeed (given that we use the same socket address for multiple interfaces) without SO_REUSEADDR configuration above. The rest of the implementation is relatively non-technical. One of the pleasant problems is the use of select() to wait for the arrival of packages on any of the offices created by createMcastSockets() above; an alternative might be to poll each socket one by one with a short reading timeout, otherwise to create a separate stream for each socket. Call select() accepts the socket list (and timeout value) and returns a nested list consisting of connector that has data ready to be read. Almost always it will be just one of the roses. Then we read the data from s.recvfrom(), recording the original src address to be used when we are further update_tables(). When the socket closes, it should be removed from the select() list, but the sockets here do not close; You can learn more about this from 30.6.1.2 dualreceive.py. The update_tables() function accepts incoming input (proanased in the List of RipEntry objects using parse_msg()) and the IP address from which it comes from, and runs a remote vector algorithm of the Distance-Vector Update Rules 13.1.1. TK is a TableKey object that represents a new destination pair (like (addr, netmask next_hop). but currentnexthop == update_sender. In the following example, routing uses the following topology to emulate competition between two TCP connections h1→h3 and h2→h3. (19 TCP Reno and Congestion Management) and TCP Vegas (22.6 TCP Vegas competition.py). / delays , we just install Link = TCLink in a Mininet() call in main(). The TCLink class is a traffic control link. Further in the topology calls section to addLink(), we add keyword parameters such as bw=BottleneckBW and delay=DELAY. To implement bandwidth limits, Mininet then takes care of creating virtual-Ethernet links with a speed limit. Mininet uses a queue hierarchy (23.7 hierarchical queue) to perform a delay. The hierarchy is managed by the TC (motion control) team, part of the LARTC system. In the above topology Mininet sets the queue h3 as a queue htb (24.11 Linux HTB, 30.8 Linux Traffic Control (tc)) with a netem queue underneath (see man page for tc-netem.8). The latter has a delay option set on demand to 110 ms in our example here. Note that this means that the delay from h3 to r will be 110 ms, and the delay from r to h3 will be 0 ms. Queue configuration is also processed using the tc command. Again, Mininet configures r's r-eth3 interface to have a htb queue with a netem queue underneath. With the command tc qdisc show we can see that the handle of the netem queue is 10:; now we can set the maximum queue size, for example 25 with following command to r: tc qdisc change dev r-eth3 handle 10: netem limit 25 To to organize TCP competition, we need the following tools: sender.py to open a TCP connection and send bulk data, after requesting a specific TCP (Reno or Vegas) dualreceive.py overload control mechanism to retrieve data from two connections and track randomtelnet.py results to send random additional data to break down the effects of the TCP phase. wintracker.py, for number of connection packs has in flight (good cwnd assessor). Python3 Python3 app similar to tcp_stalkc.py, except that it allows the TCP overload algorithm specification. This is done by using the following setsockopt() call: s.setsockopt(socket. IPPROTO_TCP, TCP_CONGESTION, cong), where kong is a renault or cubic or some other available TCP fragrance. The list is located on /proc/sys/net/ipv4/tcp_allowed_congestion_control. See also 22.1 TCP Selection on Linux. The data receiver sender.py is dualreceive.py. It listens to two ports, defaults to 5430 and 5431, and, when both connections have been made, begins to read. The main cycle begins with a select() call, where sset is a list of all (both) connected sockets: sl,_,_ = select(sset, [], []) Sl value is a subset consisting of read-ready data connector. This will usually be a list consisting of a single socket, although with so much data coming in, it can sometimes contain both. Then we call s.recv() for s in sl, and write to count1 or count2 running the total number of received bytes. If the sender closes the socket, it results in a reading of 0 bytes. At this point, dualreceive.py close the socket, at which point it must be removed from the set, because it would otherwise always appear in the sl list. (If the variable PRINT_CUMULATIVE false, the printed values are the amount of data received in the last 0.1 seconds.) If the TCP contest is fair, count1 and count2 should remain roughly equal. When printstats() does not detect changes in count1 and count2, it quits. In Python, the exit() call exits only from the current branch; other topics continue to work. In the 31.3.4 Phase Effects we show that, with a completely determinating travel time, two competing TCP connections can have a superopomoga that differs by as much as 10 simply because of the unfortunate synchronization of transfer times. We must introduce at least some degree of randomization of arrival-time packages to get meaningful results. In 31.3.6 Phase effects and overhead, we used the ns2 overhead attribute for this purpose. This is not available on real networks, however. The next best thing is to introduce some random telnet-like traffic in both phase effects of 31.3.7 and telnet traffic. That's the randomtelnet.py. This program sends packages at random intervals; interval lengths are distributed in exponential, which means that to find the length of the next interval, we select X randomly between 0 and 1 (with even distribution), and then set the length of the timeout interval to -log(X). Packet sizes are 210 bytes (a very custom value for real telnet traffic). The main thing is that the average speed of sending is kept at a small fraction (default 1%) With available throughput bottlenecks, which comes as a permanent bottleneckBW. This means that randomtelnet traffic should not significantly interfere with competing TCP connections (which, of course, do not have an additional interval between packet transmissions, except that dictated by sliding windows). Randomtelnet traffic appears to be quite effective at eliminating TCP phase effects. Randomtelnet.py sends port 5433 by default. Normally we will use netcat (17.6.2 netcat again) as a receiver as we are not interested in measuring snip for this traffic. At the end of the competition, we can look at dualreceive.py's output and determine the total 20th of each connection, as of the time when the first connection to send all its data has just finished. We can also draw pressed at regular intervals by constructing consecutive differences in the values of the aggregate passage. However, it doesn't give us the look of every cwnd connection that's readily available when modeling competition in the simulator. Indeed, it is almost impossible to access the cwnd connection directly because it is a status variable in the sender's core. However, we can do the following best: monitor the number of packages (or bytes) the connection has in flight; this is the difference between the highest bytes and the highest bytes recognized. The highest byte of ACKed is one less than the ACK field value in the last ACK packet, and the highest bytes are sent one less than the SEQ field value, as well as the package length in the last data packet. To obtain these ACK and SEQ numbers, however, requires overslaught on network connections. We can do this by using a packet capture library such as libpcap. The Pcapy Python2 module (not Python3) is a libpcap wrapper. The program is also wintracker.py Pcapy to monitor packages on r-eth1 interfaces and r-eth2 router r. It would be a little more accurate to monitor the h1-eth0 and h2-eth0, but this entails separate monitoring of two different nodes, and the difference is small because h1-r and h2-r links have little delay and no queue. Wintracker.py you must configure to track only two TCP connections that are competing. The libpcap method, and thus Pcapy, works is that we first create filter packets to identify the packages we want to capture. Filter for both host connections 10.0.3.10 and TCP and portrange 5430-5431 Host, of course, h3; packets will be captured if the starting node or destination node is h3. Similarly, packages are captured if the source port or port destination is 5430 or 5431. The connection from h1 to h3 is port 5430 per h3, and connection from h2 to h3 is to port 5431 per h3. To connect h1-h3 every time a package arrives from h1 to h3 (in the following code, we define this because port dport is 5430), we store in seq1 TCP header SEQ field plus package length. Whenever a packet is visible heading from h3 to h1 (i.e. with the original port 5430), we record an ack1 TCP header in the ACK field. The packages themselves are captured as byte arrays, but we can determine the TCP header offset and read the four-byte SEQ/ACK values with the corresponding assistant functions: _,p=cap1.next() #p is an enthusiastic package... (_,iphdr,tcphdr,data) = parsepacket(p) # find headlines sport = int2(tcphdr, TCP_SRCPORT_OFFSET) # extract port numbers dport = int2(tcphdr, TCP_DSTPORT_OFFSET) if dport == port1: # port1 == 5430 seq1 = int4(tcphdr, TCP_SEQ_OFFSET) + len(data) elif sport == port1: ack1 = int4(tcphdr, TCP_ACK_OFFSET) Individual themes are used for each connection because there is no select() option available to return the next captured package of any connection. Both the SEQ and ACK fields have been added to ISNA to them, but this will be canceled when we relecize. SEQ and ACK values are subject to 32-bit wrapping, but subtracting saves us here again. Like the dualreceive.py, the timer shoots every 100 ms and prints out the differences seq1-ack1 and seq2-ack2. It's not completely flow safe, but it's pretty close. There is some noise in the results; we can minimize that by taking an average of a few differences in a row. The next problem is for both senders to start around the same time. We could use two ssh commands, but ssh commands can take several hundred milliseconds to complete. A quicker way is to use netcat to run. On h1 and h2 we run shell scripts such as one below (separate values for $PORT and $CONG required for each of the h1 and h2, what is easiest to implement with individual scripts, say h1.sh and h2.sh): netcat -l 2345 python3 sender.py $BLOCKS 10.0.3.10 $PORT $CONG Then we start both very close to the same time with the following on r (not h3, due to the delay in the link r-h3); these commands usually end in less than ten milliseconds. echo hello | netcat h1 2345 echo hello | netcat h2 2345 Full sequence of steps on h3, start netcat -l ... for output randomtelnet.py (on two different ports) On h1 and h2 run senders randomtelnet.py On h3, run dualreceive.py On h1 and h2, run scripts (eg h1.sh and h2.sh) that wait for the signal and run sender.py On r, send two start triggers via netcat It's somewhat cumbersome; this helps to include everything in one shell script with ssh, which is used to run substrings on the corresponding guest. On the Renault-Vegas schedule at 31.5 TCP Reno vs. TCP Vegas, we set vegas α and β to 3 and 6 respectively. Implementing TCP Vegas on a Mininet virtual machine does not support changing α and β, and the default values are more like 1 and 3. To give Vegas a fight reduce the queue size by r to 10 in competition.py. Here's a graph, with in-flight monitoring packages higher and bandwidth lower: TCP Vegas gets a smaller share of bandwidth (a total of about 40% to TCP Reno's 60%), but it consistently holds its own. It turns out that TCP Vegas helps a small queue size a lot; if the queue size doubles to 20, then Vegas gets a 17% share. at the top of the schedule, we can see Renault sawing compared to Vegas' triangular teeth (slashed down, and also slashed upwards); compared to the red-green schedule at 31.5 TCP Reno vs. TCP Vegas. Teeth shapes are also somewhat mirrored in the graph on the counters, as the pressure is proportional to the disposal of the queue We can apply the same method to compare TCP Reno with TCP BBR. This was done to create a graph in 22.16 TCP BBR. Mininet approach can be used as soon as the TCP BBR module for Linux was released (in its original form); use the simulator, on the other hand, will entail the expectation that TCP BBR will be ported to the simulator. One niceness is that it is important that the discipline of the FQ queue is enabled for the sender of TCP BBR. If it's h2, for example, the following Mininet code (perhaps in competition.py) removes any existing queue discipline and adds fq: h2.cmd('tc qdisc del dev h2-rooteth') h2.cmd('tc qdisc add dev dev h2-eth root fq') The purpose of fq queues is to enable packaging; that is, the transfer of packages at regular, very small intervals. The Linux tc team, to manage traffic, allows the commitment of any implemented queue discipline (23 Queue and Scheduling) to any network interface (usually a router). A hierarchical example appears in Linux HTB 24.11. The tc command is also widely used by Mininet to manage, such as link queue volumes. A clear example of adding FQ queue discipline appears directly above. Two examples presented in this section include simple filtering of bucket markers using tbf, followed by stylish filtering of bucket markers using htb. We will use the last example to apply token-bucket filtering to only one connection class; other connections do not receive filtering. The verbosity of tc-tbf speed control is limited by the processor interrupt timer detail; typically TBF can schedule packages every 10 ms. If the transfer rate is 6 MB/s, or about four packets of 1,500 bytes per millisecond, then tbf will plan 40 packages for transmission every 10 ms. They are, however, more likely to be sent off as a splash at the start of the 10-ms interval. Some TC planners are able to achieve much more subtle control over packaging; e.g. qdisc 30.7 TCP Contest: Renault vs. BBR above. Mininet topology in both cases includes one router between two hosts, h1-r-h2. here will use routerline.py with the parameter -N 1; router then r1 with r1-eth0 interfaces connecting to h1 and r1-eth1 connection to h2. The required topology can also be built using competition.py then ignoring the third host. To send data, we will sender.py (30.6.1.1 sender.py), albeit with the default TCP overload algorithm. To retrieve the data, dualreceive.py use the data, although initially only with one connection sending any significant data. We will set the PRINT_CUMULATIVE false, dualreceive.py prints at intervals the number of bytes received during the last interval; we'll call this modified version dualreceive_incr.py. We will also redirect the stderr message to /dev/null, and start this on h2: python3 dualreceive_incr.py 2>/dev/null We start the primary sender on h1 with the following, where h2 has an IPv4 address 10.0.1.10 and 1,000,000 is the number of blocks: python3 sender.py 1000000 10.0.1.10 5430 Dual program will not make any reading until both connections are enabled, so we also need to create a second connection to h1 to start; this second connection sends only one block of data: python3 sender.py 1 10.0.1.10 5431 At this stage, the double result should generate output somewhat like the next one (with time marks in the first column rounded to the nearest millisecond). Byte count numbers in the middle column are fairly hardware dependent 1.016 14079000 0 1.106 12702000 0 1.216 14724000 0 1.316 13666448 0 1.406 11877552 0 This means that on average h2 receives about 13 MB every 100 m, about 1.0 Gbps. Now we run the command lower by r1 to reduce the speed (tc requires the abbreviation mbit for megabits/sec; it treats mbps as MegaBytes per second). The bucket marker filter settings are fast and serial. The purpose of the limit parameter that is used by netem and several other qdiscs is to determine the maximum queue size for pending packets. Its value here is not very significant, but too low a value can lead to the loss of packets and thus instantly immerse bandwidth. Too high on the other hand can cause buffer bloat (21.5.1 Bufferbloat). tc qdisc add dev r1-eth1 root TBF speed 40mbit burst 50kb limit 200kb We get output something like this: 1.002 477840 0 1.102 477840 0 0 1.202 477840 0 1.302 482184 0 1.402 473496 0 477840 bytes per 100 ms is 38.2 Mbps. an additional 5% or so to 40 Mbps corresponds mainly to package headers (66 bytes out of every 1514, although to see this from WireShark, we need to disable TSO, offload TCP 17.5). We can also change the speed dynamically: TC qdisc change dev r1-eth1 root TBF speed 20mbit burst 100kb limit 200kb Higher TBF usage allows throttle (or police) all traffic through interface r1-eth1. Suppose we want the police to take away only traffic? Then we use a hierarchical token bucket or htb. We created the root node of HTB, with no restrictions, and then create two child nodes, one for police traffic and one for default traffic. To create a htb hierarchy, we will first create a root qdisc and a linked root class. We need the speed of the raw interface, here it is customary to be 1000mbit. Class IDs are formed major:minor, where the primary is the integer of the root handle, and insignificant is another integer. tc qdisc add dev r1-eth1 root handle 1: htb default 10 tc class add dev r1-eth1 parent 1: classid 1:1 htb speed 1000mbit We now create two child classes (not qdiscs), one for limited speed traffic and one for default traffic. The limited-speed class has a class of 1:2 here; the default classid is 1:10. tc class add dev r1-eth1 parent 1: classid 1:2 htb speed 40mbit tc class add dev r1-eth1 parent 1: classid 1:10 htb speed 1000mbit We still need a classifier (or filter) to assign the selected traffic to class 1:2. Our goal is to have police traffic port 5430 (by default, dualreceive.py traffic at ports 5430 and 5431). There are several classifiers available; for example u32 (man tc-u32) and bpf (man tc-bpf). The latter is based on the Berkeley Packet Filter virtual machine for package recognition. However, what we use here - mainly because it seems to work most reliably - is the iptables fwmark mechanism used earlier in the 13.6 routing on other attributes. Iptables is designed to filter - and sometimes modify - packages; we can associate fwmark values from 2 to packets related to TCP port 5430 with the command below (fwmark value does not become part of the package; it exists only when the package remains in the core). iptables --append FORWARD --table mangle --protocol tcp --dport 5430 --jump MARK --set-mark 2 When it is running on r1, the packages forwarded r1 to the TCP 5430 port receive fwmark on arrival. The next step will be to tell the tc subsystem that packages with fwmark 2 value should be placed in class 1:2; this is a class limited to a higher tariff. The next flowid command can be used as a synonym for classid. TC filter add dev r1-eth1 parent 1:0 protocol IP pen 2 fw classid 1:2 We can view all these settings with TC qdisc show dev r1-eth1 tc class show dev r1-eth1 tc filter show dev r1-eth1 parent 1:1table ips --table mangle --list We now check that it all works. As with tbf, we start dualreceive_incr.py on h2 and two senders on h1. This time both senders send large amounts of data: h2: python3 dualreceive_incr.py 2>/dev/null h1: python3 sender.py 500000 10.0.1.10 5430 h1: python3 sender.py 500000 10.0.1.10 5431 If everything works, then shortly after the start of the second we should see something like the output below (after both TCP connections stabilize). The average column is the number of data bytes received to the police port, 5430. 5430. 453224 10425600 1.100 457568 10230120 1.200 461912 9934728 1.300 476392 1065583 2 1.300 401 438744 10230120 With 66 bytes of TCP/IP headers in each 1514-byte package, our requested 40mbps data speed cap should yield about 478,000 bytes every 0.1 sec. The slight reduction above appears to be due to TCP competition; a full speed of 478,000 bytes is achieved after the connection to port 5431 is completed. In this section, we introduce a POX controller for OpenFlow switches (3.4.1 OpenFlow Switches), which allows network-defined software research (3.4 Software-defined networks). In the switchline.py Ethernet switch from previous Mininet() calls included the settings controller =DefaultController; this causes each switch to behave like a normal Ethernet learning switch. By using Pox to create customized controllers, we can explore other options for the switch. Pox is preinstalled on the Mininet virtual machine. Pox is like a Mininet written in Python2. It receives and sends an OpenFlow message in response to events. Event-related messages for our purposes can be grouped here into the following categories: PacketIn: The switch informs the controller of the arriving package, usually because the switch doesn't know how to forward the package or doesn't know how to forward the package without flooding. Often, but not always, PacketIn events will result in the controller providing new forwarding instructions. ConnectionUP: The switch is connected to the controller. This will be the point at which the controller gives the switch its initial instructions for handling packages. LinkEvent: The switch informs the controller that the link becomes available or becomes unavailable; This includes initial reports on the availability of links. BarrierEvent: Switch response to OpenFlow Barrier messages, which means the switch has completed its responses to all messages received to the Barrier, and can now start replying to messages received after the barrier. The Pox app comes with several demo modules illustrating how controllers can be programmed; they are in pox/misc and pox/forwarding directories. The starting point for Pox documentation is the Pox wiki (archived copy on poxwiki.pdf), which, among other things, includes short outlines of those applications. We are currently reviewing some of these programs; most were written by James McCovey and licensed under Apache license. Pox code data structures are very closely related to the OpenFlow switch specification, versions of which can be found in OpenNetworking.org library. As the first example of Pox, assume that we take a copy of the switchline.py file and make the following changes: change the controller specification inside the Mininet() call, from controller =DefaultController to controller=RemoteController. add the following lines immediately after call: c = RemoteController('c', ip='127.0.0.1', port=6633) net.addController(c) This modified version is available as switchline_rc.py, rc for remote controller. If we are currently running this modified version, then pings are not because RemoteController, c, does not exist yet; in the absence of a controller, the typical answer to switches is to do nothing. Now we start Pox, in the /home/mininet/pox directory, as follows; this downloads the pox/forwarding/hub.py Ping file the connection should be restored! The switch connects to the controller at IPv4 127.0.0.1 (more on this below) and TCP port 6633. At the moment, the controller can tell the switch what to do. An hub.py sets each switch as a simple hub by flooding each arriving package from all the other interfaces (although for linear topology switchline_rc.py it doesn't matter much). The corresponding code is here: def _handle_ConnectionUp (event): msg = of.ofp_flow_mod() msg.actions.append(of.ofp_action_output(port = of. OFPP_FLOOD)) event.connection.send(msg) This is the ConnectionUp event handler; it is called when the first reports switch for duty. When each switch connects to a controller, the hub.py code instructs the switch to forward each arriving package to the OFPP_FLOOD virtual port, which means forwarding all other ports. The event parameter has a ConnectionUp class subclass of the event class. It is defined in the pose/openflow/__init__.py. Most switch-event objects throughout Pox include a connection field that the controller can use to send messages back to the switch, and a dpid field representing the switch ID number. Typically, the Mininet s1 switch will have dpid 1, etc. The above code creates an OpenFlow message to change the flow table to msg; is one of several types of switch controller messages defined in the OpenFlow standard. The msg.actions field is a list of actions to take; to this list, we will add a forwarding action to the designated (virtual) port OFPP_FLOOD. Normally we will also add the appropriate rules for packages to be forwarded to the msg.match list, but here we want to forward all packages and therefore do not need to comply. Another - though functionally equivalent - approach is taken in mutilated/miscellaneous/of_tutorial.py. Here, the connectionup event response does not include a connection to the switch (although the connection is stored Tutorial.__init__()). Instead, as the switch tells each arriving package controller, the controller responds by informing the switch to flood the package from each port (this approach will cause unnecessary enough traffic that it will not be used in the production code). The code (slightly merged) looks like this: def _handle_PacketIn (self, event): packet = event.parsed # This is analyzed cell data. packet_in = event.ofp # Actual Message. self.act_like_hub(package, packet_in) def act_like_hub (l, package, packet_in): msg = of.ofp_packet_out() msg.data = packet_in action = of.ofp_action_output(port = of. OFPP_ALL) msg.actions.append(action) self.connection.send(msg) Event is now an instance of the PacketIn class. This time, the switch sends a packet message to the switch. The package and packet_in are two different views of the package; the first is analyzed, and therefore it is usually easier to get information, and the second presents the entire package as it was obtained by the switch. This is the last format that is sent back to the switch in the msg.data box. The virtual port OFPP_ALL equivalent OFPP_FLOOD. For any hub implementation, if we start WireShark on h2 and then ping from h4 to h1, we'll see pings on h2. This, for example, demonstrates that s2 behaves like a hub rather than a switch. The following example of Pox, l2_pairs.py, implements a genuine

Ethernet learning switch. This is the implementation of the rsa-based switch discussed in 3.4.2 Learning Switches in OpenFlow. This module operates on the Ethernet address layer (level 2, part of the l2 name), and streams are specified in pairs of addresses (src,dst). The l2_pairs.py module started using the Pox ./pox.py forwarding.l2_pairs. A simple implementation of the Ethernet learning switch runs into issue: The switch must contact the controller whenever the package source address has not been seen before, so the controller can send back to the forwarding rule switch in order to reach that original address. But the primary lookup in the flow switch table must be at the destination address. The approach used here uses a single OpenFlow table, compared to the 30.9.3 l2_nx.py mechanism. However, the studied flow table records will include compliance rules for both the source and the target address of the package, so that a separate record is required for each pair of communication hosts. Therefore, the number of thread entries is scaled as O(N2), which presents a scaling problem for very large switches, but which we will ignore here. When a switch sees a package with an unmatched (dst,src) address pair, it forwards it to a controller that has two cases to consider: If the controller doesn't know how to get to the destination address from the current switch, it tells the switch to flood the package. However, the controller also records for later use the original address of the package and its arrival interface. If the controller knows that the destination address can be reached from this switch by using the dst_port switch port, it sends to the switch instructions to create a forwarding record for the (dst,src)→dst_port. At the same time, the controller also sends a back forwarding record switch for (src,dst) forwarding through the port by which the package arrived. V.O. supports a partial map from addresses to switch ports in a dictionary table that accepts a pair (switch, destination) as a key and which returns the switching of port numbers as values. The switch is represented by the event.connection object that is used to reach the switch, and the destination addresses are represented as Pox EthAddr objects. The program handles only packetIn events. The basic steps of the PacketIn handler are as follows: First, when a package arrives, we put its switch and source into the table: table[(event.connection,packet.src)] = event.port The next step will be to check to see if there is a record in the table for assignment, looking for a table[(event.,connection packet.dst)]. If there is no record, the package is flooded with the same mechanism as the of_tutorial.py above: we create a package message containing the package to be flooded and send it back to the switch. If, on the other hand, the controller believes that the destination address can be reached by using the port dst_port, it goes as follows. First, we create a reverse record; event.port is the port by which the package just arrived: msg = of.ofp_flow_mod() msg.match.dl_dst = packet.src # reverse dst and src msg.match.dl_src = packet.dst # reverse dst and src msg.actions.append(of.ofp_action_output(port = event.port)) event.connection.send(msg) This is usually a forwarding rule created in hub.py. except that we forward here through a specific event.port rather than a virtual port OFPP_FLOOD, and perhaps more importantly, we add two package matching rules to msg.match. The next step is to create a similar appropriate rule for the src-to-dst thread and enable the transfer package. Therefore, the message changes the table so that it does double duty as a batch message. msg = of.ofp_flow_mod() msg.data = event.ofp # Forward incoming package msg.match.dl_src = packet.src # not canceled this time! msg.match.dl_dst =packet.dst msg.actions.append(of.ofp_action_output(port=dst_port)) event.connection.send(msg) The msg.match object has quite a few potential matching fields; below is taken from Pox-Wiki: Attribute Value in_port Switch Port Number Package arrived at dl_src Ethernet original address dl_dst Ethernet destination address dl_type Ethertype / length (e.g. 0x0800 = IPv4) nw_tos IPv4 TOS/DS bits nw_proto IPv4 (e.g. IPv4 6 = TCP), or lower 8 bitS of ARP code nw_src IPv4 source address nw_dst IP address destination tp_src TCP/UDP source port tp_dst TCP/UDP port destination port You can also create an msg.match object that corresponds to all fields in this package. We can watch forwarders created l2_pairs.py using Linux ovs-ofctl. Assume that you switchline_rc.py, and then the Pox l2_pairs.py module. Next, with mininet, we have h1 ping h4 and h2 ping h4. If we are now launching a command (on a virtual but from the Linux line) we receive cookies = 0x0, ..., dl_src = 00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=output:3 cookie=0x0, ...,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02 actions=output:1 cookie=0x0, ...,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:3 cookie=0x0, ...,dl_src=00:00 00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=output:2 Because we used autoSetMacs =True option in mininet() call in switchline_rc.py, Ethernet addresses assigned to hosts are easy to follow: h1 is 00:00:00:00:01, etc. the first and fourth lines above as a result of h1 pinging h4; we can see from the source port at the end of each line that s1 must be reachable from s2 through port 2 and s3 via port 3. Similarly, the average two lines are the result of h2 pinging h4; h2 lies from the port s2 1. These port numbers correspond to the interface numbers shown in the diagram at 30.3 Multiple switches per line. An l2_nx.py achieves the same Ethernet switch effect as l2_pairs.py, but using only O(N) space. However, it uses two OpenFlow tables, one for destination addresses, and one for source addresses. In the implementation here, the source addresses are held in Table 0, and the destination addresses are in Table 1; this is a backward approach to several tables outlined in the 3.4.2 training switches in OpenFlow. L2 again refers to the 2 network layer, and nx refers to the so-called Nicira extensions to Pox, which allow multiple stream tables to be used. Table 0 is initially configured to try to match the original address. If there is no match, the package is forwarded to the controller and sent to table 1. If a match is matched, the package is sent to Table 1 but not to the controller. Table 1 then searches for matching at destination address. If one is found, the package is forwarded to the destination, and if there is no match, the package is flooded. Using two OpenFlow tables in Pox requires downloading so-called Nicira extensions (hence nx in the module name here). They need a slightly harder command line: ./pox.py openflow.nicira --convert-packet-in forwarding.l2_nx Nicira will also require, for example, nx.nx_flow_mod() instead of of.ofp_flow_mod(). Actions for each table that are set during ConnectionUp event handling do not match. The action becomes the default action if msg.match() rules are not enabled and the priority is low; Recall (3.4.1 OpenFlow switches) that if the package matches multiple table entries, the record with the highest priority wins. The priority here is set to 1; The default Pox priority to be used (i.e.) for later, more specific thread table entries is 32,768. The first step is to organize Table 0 to forward to the controller and to Table 1. msg = nx.nx_flow_mod() msg.table_id = 0 # is not required because it is msg.priority = 1 # low priority msg.actions.append(of.ofp_action_output(port = of. OFPP_CONTROLLER)) msg.actions.append(nx.nx_action_resubmit.resubmit_table=1)) event.connection.send(msg) Next we report table 1 to flood the default packages: msg =nx.nx_flow_mod() msg.table_id =1 msg.priority=1 msg.actions.append(of.ofp_action_output(port=of. OFPP_FLOOD)) event.connection.send(msg) Now we define the PacketIn handler. First there is table 0 of the match on the source of the packages; if there is a coincidence, the source address was seen by the controller, and therefore the package is no longer forwarded to the controller (it is forwarded only to Table 1). msg = nx.nx_flow_mod() msg.table_id = 0 msg.match.of_eth_src = packet.src # corresponds to the source msg.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1)) event.connection.send(msg) Now comes table 1, where we match the destination address. All we know at the moment is that the package with the original address packet.src came from the port event.port, and we forward any packages addressed packet.src through this port: msg = nx.nx_flow_mod() msg.table_id = 1 msg.match.of_eth_dst = packet.src # this rule applies only to packet.src msg.actions.append(of.ofp_action_output(port = event.port)) event.connection.send(msg) Note that there is no network state supported on the controller; analogue here tabular dictionary l2_pairs.py. Suppose we have a simple h1–s1–h2 network. When h1 sends h2, the controller will add a record to table s1 0 indicating that h1 is a known source address. It will also add a record to table s1 1, indicating that h1 can be reached via the port to the left of s1. Similarly, when h2 responds, s1 will have h2 added to its table 0 and then to its Table 1. The purpose of the multipurpose example is to illustrate how different TCP connections between the two hosts can be routed in different ways; in this case, through various main lines. This example and the following are not part of standard mininet or Pox distributions. Unlike other examples discussed here, these examples consist of a Mininet code to configure a specific network topology and a corresponding Pox controller module, which is written to work properly only with mufology. Most real networks evolve over time, making such a hard connection between topology and controller impractical (though sometimes it can work well in data centers). The goal here, however, is to illustrate openFlow's specific capabilities in a (relatively) simple setting. Multipurpose topology includes several trunk lines between host h1 and h2, both on the following diagram; trunk lines are s1–s3 and s2–s4 references. The Mininet multitrunk12.py and the corresponding Pox multitrunkpox.py. By default, the number of trunk rows is K=2, but they can be changed by installation of the K variable. cycle of broadcast traffic, never flooded by the s2-s4 link. TCP traffic takes either s1-s3 trunk or s2-s4 trunk. We will refer to two directions h1→h2 and h2→h1 TCP connections as threads, according to the use in the 11.1 IPv6 header. Only h1→h2 streams will have their routing vary; h2→h1 streams will always take the s1-s3 path. It doesn't matter if the original h1 to h2 connection is open or from h2 to h1. The first TCP stream from h1 to h2 runs through s1-s3. After that, the following links alternate in round fashion between s1-s3 and s2-s4. To achieve this, we must, of course, include TCP ports in openFlow forwarding information. All links will have bandwidth installed in Mininet. This includes using the link = TCLink option; TC here is deciphered as Traffic Management. We don't otherwise use bandwidth limits. TCLinks can also have a set of queue sizes, like the 30.6 TCP Competition: Reno vs. Vegas. For ARP and ICMP traffic, two OpenFlow tables are used as 30.9.3 l2_nx.py. PacketIn messages for ARP and ICMP packages like switches to find out mac addresses of hosts and how the controller learns which switch ports are directly connected to the hosts. TCP traffic is handled differently below. When you initially process ConnectionUp messages, the switches receive their default instructions for processing packages for ARP and ICMP packets, and a switchNode object is created in the controller for each switch. These objects will eventually contain information that a neighbor switch or host reaches each switch port, but at this point none of this information is yet available. The next step is to process LinkEvent messages initiated by the detection module. This module must be included in the ./pox.py command line in order for this example to work. The detection module sends each switch because it connects to a controller, a special detection package in the Reference Layer Detection Protocol (LLDP) format; this package includes the dpid value of the original switch and the port switch by which the original switch sent the package. When the LLDP package is received by a nearby switch that switches it back to the controller, together with the dpid and the port to retrieve the switch. At this point, the controller knows the switches and port numbers at each end of the link. The controller then reports it to our multipurpose module through the LinkEvent event. When processing LinkEvent messages, the multitrunkpox module learns for each switch which ports connect directly to adjacent switches. At the end of the LinkEvent phase, which usually takes a few seconds, each SwitchNode switch is aware of all the directly connected neighbor switches. The room was quite big and comfortable. After h1 and h2 exchange a pair of packetin related events to tell multitrunkpox which switch ports are connected to hosts. Ethernet addresses are also examined. For example, if we perform h1 ping h2, then the information contained in the SwitchNode graph will be completed. Now suppose h1 tries to open a TCP connection to h2, for example through ssh. The first package is the TCP SYN package. We create a thread for the package, flow = Flow(psrc, pdst, ipv4.srcip, ipv4.dstip, tcp.srcport, tcp.dstport), and then see if the path is already assigned to this thread in flow_to_path. For the very first package, this will never be the case. If the path does not exist, we create one by first selecting the trunk: trunkswitch = picktrunk(flow) path = findpath(flow, trunkswitch) The first path will be the Python list [h1, s5, s1, s3, s6, h2], where the switches are represented by SwitchNode objects. Supposedly, the last step is to trigger the result = create_path_entries (thread, path) to create forwarding rules for each switch. Using the path as outlined above, SwitchNode objects know which s5 port to use to achieve s1, etc. Because the first TCP SYN package required pre-ARP exchange, and because the ARP exchange will result in s6 learning which port to achieve h2, this should work. But in reality it's not, at least not always. The problem is that Pox creates separate internal themes for processing the ARP package and processing TCP packets, and the former thread may not have yet installed the h2 location in the corresponding SwitchNode object by the time the last thread calls create_path_entries() and requires an h2 location. This state of race is unfortunate, but it cannot be avoided. As a backup, if you cannot create a path, we flood the TCP packet along the s1-s3 link (even if the s2-s4 link trunk is selected) and wait for the next TCP package to try again. Very soon s6 will know how to achieve h2, and so create_path_entries() will succeed. If we run everything, create two xterms on h1 and then create two ssh connections h2, we can see the forwarding of records using ovs-ofctl. Let's run limits of attention only to these stream entries from foo = tcp, we get (with a little sorting) cookies = 0x0, ..., tcp,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:02,nw_src=10.0.0.1,nw_dst=010.0.0.2,tp_src=59404,tp_dst=22 actions=output:1 cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:01,dl_dst=00 00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,tp_src=59526 ,tp_dst=22 actions=output:2 cookie=0x0 , ..., tcp,dl_src=00:00:00:00:02,dl_dst=00:00:00:00:01,nw_src =0 10.0.0.2,nw_dst=10.0.0.1,tp_src=22.tp_dst=59404 actions=output:3 cookie=0x0, ..., The first two entries represent h1→h2 streams. The first connection has a source TCP port 59404 and is routed through the trunk s1-s3; we see that the output port with s5 is port 1, which is really the port that s5 uses to achieve s1 (exit of the Mininet communications command includes s5-eth1<->s1-eth2). Similarly, the source port that is used on the s5 second connection with outbound TCP port 59526 is 2, which port s5 uses to achieve s2. The s5 switch reaches host h1 via port 3, which can be seen in the last two recordings above that match the backflows of h2→h1. The OpenFlow timeout here is endless. It is not a good idea if the system will run indefinitely, with a steady stream of short-term TCP connections. This, however, makes it easier to view links with ovs-ofctl before they disappear. The implementation of the production would require a end-to-end timeout, and then have to ensure that connections that idle longer than the waiting interval were properly restored when they resumed sending. The multipurpose strategy presented here is comparable to multi-Path, 13.7 ECMP. In both cases, traffic is divided between several ways to improve traffic jams. Here, a separate TCP connection is assigned to the trunk by the controller (and can be reassigned upon request, possibly to improve the load balance). In ECMP, it is commonly assigned tcp connections to paths using a pseudo-random hash server, in which case the approach here offers the potential to better control the distribution of traffic between trunk links. In some configurations, however, ECMP may route packages through multiple links to the round robin package by package instead of connecting by connection; this allows for much better load balancing. OpenFlow has low-level support for this approach in the mechanism of the selected group. An item that matches traffic with a table can forward traffic to a so-called group instead of through a port. The action of the selected group is to select one of the output action sets (often on a round basis) and apply this action to the package. Basically, we could implement this on s5 to have consecutive packages shipped either to s1 or s2 in round robin fashion. In practice, Pox support for selected groups appears to be under-developed at the time of writing (2017) to make it practical. The following example demonstrates simple load balancing. The topology is somewhat reversed in the previous example: there are now three nodes (N=3) at each end, and only one trunk line (K=1) (there are also no input/exit switches left and right). The right hosts act as servers, and are renamed t1, t2, and t3. The servers all receive the same IPv4 address, 10.0.0.1. This will usually cause chaos, but servers are not allowed to talk to each other and the controller ensures that servers are not aware of each other. Specifically, the controller ensures that the servers never all simultaneously respond to an ARP request who has a 10.0.0.1 with r. Mininet loadbalance31.py file and the corresponding Pox loadbalancepox.py. Node r is a router, not a switch, and so its four interfaces are assigned to separate subnets. Each host is on its own subnet, which it shares with r. Router r then connects to a single switch, s; connection from s to controller c. The idea is that each TCP connection from any of hello to 10.0.0.1 is connected, via s, to one of the ti servers, but different connections will connect to different servers. In this implementation, the server selection is round-robin, so the first three TCP connections will connect to t1, t2, and t3 respectively, and the fourth will reconnect to t1. The t1-t3 servers configured for all have the same IPv4 address 10.0.0.1; there is no overwriting of the address made for packages coming from the left. However, as in the previous example, when the first package of each new TCP connection from left to right arrives on s, it is forwarded to c, which then selects a specific ti and creates a forwarding rule for that connection. As in the previous example, each TCP connection includes two Flow objects, one in each direction, and separate openFlow forwarding records are created for each thread. There is no need for ways; the main operation of TCP connection routing is as follows: server = pickserver(flow) flow_to_server[flow] = server addTCPrule(event.connection, stream, server +1) # ti is in the port i + 1 addTCPrule (event.connection, flow.reverse(), 1) # port 1 leads to r The biggest technical problem is ARP: usually r and ti will contact each other via ARP to find the appropriate LAN addresses, but it will not end with identical IPv4 addresses. So instead we create static ARP records. We know (checking) that the MAC address r-eth0 is 00:00:00:00:04, and so mininet file runs the following command on each of ti: arp -s 10.0.0.2 00:00:00:00:00:04 This creates a static ARP entry on each of ti that leaves them knowing the MAC address for the default router 10.0.0.2. As a consequence, none of them issue an ARP search request r. Another direction is similar, except that r (which is not actually on the load balancing area) should think that 10.0.0.1 has one MAC address. So we give each of ti the same MAC address (which would generally cause even more chaos than giving them all the same IPv4 address); this address 00:00:00:00:01:ff. Then we set a permanent ARP entry to r with ARP -s 10.0.0.1 00:00:00:00:ff Now, when h1, say, sends a TCP package before 10.0.0.1, r forwards it to the MAC address 00:00:00:00:01:ff and then forwards it to any of the t1.. t3 it was instructed by the controller to forward it. The package arrives at ti with the correct IPv4 address (10.0.0.1) and the correct MAC address (00:00:00:00:01:ff) and therefore accepts. The answers are similar: ti sends r to mac address 00:00:00:00:04. As part of connectionup processing, we set rules so that ICMP packages on the left are always directed to t1. Therefore, we have one responder to ping requests. It is possible that some important ICMP messages - such as fragmentation required, but a DF flag installed - will be lost as a result. If we run applications and create xterm windows for h1, h2 and h3 and, of each, connect to 10.0.0.1 via ssh, we can say that we have reached t1, t2 or t3 respectively by running ifconfig. The Ethernet interface on t1 is called t1-eth0, and similarly for t2 and t3. (Finding another way to discern ti is not easy.) An even easier way to see the connection rotation is to run h1 ssh 10.0.0.1 ifconfig on a mininet > request multiple times sequentially and note successive interface names. If we create three connections and then run ovs-ofctl dump-flows s and look at tcp records with destination address 10.0.0.1, we get this: cookie = 0x0, ..., tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:01:ff,nw_src=10.0.1,nw_dst=10.0.0.01.tp_src=35110,tp_dst=22 actions=output:2 cookie=0x0,..., tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:01:ff ,nw_src=10.0.2.1 ,nw_dst=10.0.0.1,tp_src=44014,tp_dst=22 actions=output:3 cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:01:ff,nw_src=010.0 0.3.1,nw_dst=10.0.0.1,tp_src=55598,tp_dst=22 actions=output:4 Three different threads accept the original ports 2, 3 and 4 on the p, according to t1, t2 and t3. This latest example of a Pox controller takes an arbitrary Mininet network, studies topology, and then sets OpenFlow rules so that all traffic is forwarded in the shortest way hopcount measured. OpenFlow package forwarding rules are set on demand when traffic between the two hosts is first seen. This module is compatible with loop topology, provided that the module spanning_tree also loaded. We begin with the spanning_tree of the module. This uses the openflow.discovery module, as in 30.9.4 multitrunk.py, to build a map of all connections, and then runs an algorithm spanning tree 3.1 Covers tree algorithm and redundancy. The result is a list of switching ports on which there should be no flooding; flooding is then disabled by setting the OpenFlow attribute NO_FLOOD on these ports. We can see the commutor ports that have been disabled using NO_FLOOD using ovs-ofctl show s. One niceness is spanning_tree that the module is never quite sure when the network is complete. Therefore, it recalcates the covered tree after each LinkEvent. We can see spanning_tree in action if we create a Mininet network of four switches in the loop, as in the 9.0 exercise below, and run the following: ./pox.py openflow.discovery openflow.spanning_tree forwarding.l2_pairs If we run an ovs-ofctl show for each switch, we get something like the following: s1: (s1-eth2): ... NO_FLOOD s2: (s2-eth2): ... NO_FLOOD we can check using the mininet link command that s1-eth2 and s2-eth2 connected interfaces. We can check with tcpdump -i s1-eth2 that no packages circulate indefinitely. We can also check with ovs-ofctl dump threads that s1-s2 link is not used at all, even for s1-s2 traffic. This is not surprising; The l2_pairs ultimately examines outgoing addresses from flooded ARP packets that are not sent at the s1-s2 link. If s1 hears nothing from s2, it will never learn to send anything to s2. The l2_multi module, on the other hand, creates a complete map of all network links (separate from the map created by the spanning_tree module), and then calculates the best route between each pair of hosts. To calculate routes l2_multi uses floyd-Warshall's algorithm (described below), which is a form of remote-vector algorithm optimized in the presence of a full network card. (The shortest path algorithm 13.5.1 The shortest path-first algorithm may be the quickest choice.) To avoid having to rebuild the forwarding map on each LinkEvent, l2_multi does not create routes until you see the first package (not counting lldp packages). Up to this point, as a rule, the network is stable. If we run the example above using the Mininet rectangle topology, we will find again that the tree covered has turned off the flood link s1–s2. However, if we have h1 ping h2, we see that the traffic h1→h2 accepts the link s1-s2. Here is part of the result of ovs-ofctl dump streams s1: cookie = 0x0, ..., priority=65535,icmp,in_port=1,...,dl_src=00:00:00:00:01,dl_dst=00:00:00 00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,...,icmp_type=8... actions=output:2 cookie=0x0, ..., priority=65535,icmp,in_port=1,... 0,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,...,icmp_type=0... actions=output:2 Note that l2_multi creates separate flow table rules not only for ARP and ICMP, but also for ping (icmp_type=8) and pingly (icmp_type=0). Such fine-grained compliance rules are a matter of preference. Here's a quick outline of floyd-Warshall's algorithm. We assume that the switches are numbered on {1,...,N}. The form k<=N:; at the beginning of stage k, we assume that we have found a better path between each i and j for which each intermediate switch is on the way less than k. For many (i,j) pairs of this path may not be. At the stage before, we examine, with the inner loop, all pairs (i,j). We look if there is a path from I to and second way from to j. If there is, we will combine the ways i-to-k and k-to-j to create a new path i-to-j, which we will call P. If there were any the previous i-to-j path, then we add P. If there was a previous i-to-j Q path that is longer than P, we replace Q with P. At the end of the k=N stage, all paths were detected. Exercises receive fractional (floating point) numbers allowing interpolation of new exercises. Exercise 2.5 differs, for example, from exercises 2.0 and 3.0. Exercises marked ♢ have solutions or hints on 24.18 Mininet solutions. 1.0. When implementing RIP 30.5 IP routers with simple remote vector implementation, add Split Horizon (13.2.1.1 Split Horizon). 2.0. When implementing RIP 30.5 IP routers with simple remote-vector implementation, add support for link failures (rule 3, 13.1.1 Remote Vector Update Rules) 3.0. Explain why example 30.9.3 is l2_nx.py table 0 and Table 1 will always have the same records. 4.0. Suppose we try to eliminate the source addresses l2_pairs implementation. by default, all switches report all packets to the controller, and the controller then tells the switch to flood the package. if the package from ha to HB goes to the switch S, and S reports the package to the controller, and the controller knows how to reach HB from S, then the controller sets the rules for forwarding to S for the purpose of HB. The controller then tells S to resend the package. In the future, S will not report HB packets to the controller. when S tells the controller a package from ha to HB, the controller notes that the ha can be reached through the port on the S by which the package arrived. Why doesn't it work? Hint: consider the example of switching (30.3 Multiple switches in line), with sending h1 to h4, h4 sending to h1, h3 sending to h1 and finally h1 sending to h3. 5.0. Suppose we make the following change to the above strategy: if the package from ha to VTV comes when the jumper S, and S reports the package to the controller, and the controller knows how to achieve both ha and NG from S, then the controller sets the rules for redirection to S for the directions ha and NG. The controller then tells S to resend the package. In the future, S will not report packages ha or HB to the controller. Show that it still doesn't work for the switching example. 6.0. Suppose we try to implement an Ethernet switch as follows: the default switch action for an unmatched package is to flood it and send it to the controller. if the package from ha to HB goes to the switch S, and S reports the package to the controller, and the controller knows how to achieve both ha and HB from S, then the controller sets the rules for forwarding in S for the directions ha and HB. In the future, S will not report packages with these areas to the controller. Unlike exercise 4.0, the controller then tells S to flood the package from ha to HB, even if it can be forwarded directly. Traffic is sent to below: h1 h2 h3 ▏▏▏ s1−−−<s2−−−−s3 (a)♢. Show that if the traffic is h1 pings h2, h3 pings h1, all three switches will know where h3 is located. (b). Show that if the traffic is: h1 pings h2, h1 pings h3, then none of the switches will know where h3. Recall that each ping for the new destination begins with an ARP broadcast. Broadcast packets are always sent to the controller because there is no matching purpose. 7.0. In 30.9.5 loadbalance31.py, we could configure ti to have a default router 10.0.0.3, let's say, and then created a corresponding static ARP record for 10.0.0.3: IP route add to default via 10.0.0.3 dev ti-eth0 arp -s 10.0.0.3 00:00:00:00:04 Still working, although ti think their router is on 10.0.0.3 and it's actually in 10.0.0.2. Explain why. (Hint: how does the IPv4 router address actually use ti?) 8.0. As discussed in the text, the condition of the competition may occur on the example of 30.9.4 multitrunk.py, where at that time the first package TCP controller still does not know where h2 is, even if he must find out that after processing the first ARP package. Explain why a similar state of the race cannot occur in 30.9.5 loadbalance31.py. 9.0. Create a network of mininets with four hosts and four switches, as shown below: h1−−−-s1−−−−−−-h2--h2 ▏▏▏ h4−−−−−−−−−--h3 Switches must be used by an external controller. Now let Pox be this controller, with ./pox.py openflow.discovery openflow.spanning_tree l2_pairs.py 10.0. Create the topology below with Mininet. Run the L2_multi Pox module as a controller, openflow.spanning_tree and identify the tree that you created. Also define the path taken by icmp movement from h1 to h2. By 2.