The CS 551/651 Lab Manual

Contents

1 Introduction

During the course, you will be developing and putting together a mini-Internet of your own, all within a single virtual machine! We have structured this activity as 3 labs (some of which have multiple parts).

To do these labs, at the beginning of the semester, you will be given three things:

1. This document, which contains all the information you need for the labs.
2. A VM image that you can run on your laptop to do the lab.
3. A personalized git repository which contains starter files that you will need. You will use this repository to save your work and also to submit your lab code and reports (details are discussed in each of the labs). If you have not used git before, please familiarize yourself with git using one of the many tutorials on the Web.

For doing the labs, you will need to be familiar with **C and Python**. If you haven't programmed in these languages before (especially C), **you will find the course very difficult**.

1.1 Labs

Lab 1 asks you to implement a router in software. Routers contain many different pieces of functionality, including packet forwarding, ARP, and ICMP. Your router will have statically configured routing tables, but should otherwise have a significant subset of modern router functionality.

Lab 2 requires you to configure a mini-Internet consisting of a Tier-1 ISP with two stub Autonomous Systems "East" and "West". This involves configuring intra-domain routing (OSPF) and an inter-domain routing protocol (BGP). At the end of this project, you should be able to ping and traceroute from a client machine in "West" to server machines in "East".

Lab 3 introduces you to transport protocol design and implementation. You will implement cTCP, a nearly full-fledged version of TCP but one which runs in user-space (not in the OS kernel). You will add sliding window flow control and congestion control. For the latter, you will implement TCP BBR congestion control, a recent TCP design from Google. Then, you will test this implementation on the mini-Internet you designed in Lab 2.

1.2 Notes

Some **important** points to remember:

- Labs 1-3 are cumulative, which means that Lab 3 depends on all other labs being correct. Therefore, you cannot afford to skip a lab or leave it unfinished: please keep this in mind as you schedule your work through the semester. (Also see Lab Late Submissions Policy).
- All the labs are to be completed **individually.** Please remember that you have signed an academic integrity agreement, and have agreed that any integrity violation will be automatically reported to SJACS.
- As you develop the labs, we suggest you **frequently commit your work to your git repository**. This will help you recover from unforeseen failures.
- When you write code, pay attention to the coding style. Coding styles are important for computer scientists to follow because most programming projects involves large teams, and having a uniform coding style helps readability across the group. It is good practice for you to try to mimic the coding style in the files we have provided.

1.3 Lab Late Submissions Policy

Three times during the semester, a student may extend the due date of a programming assignment by twenty four hours without needing prior permission. These are known as "grace days." In order to use a grace day, you must fill out this grace day request form **before** the lab's non-extended deadline. (You will need to be logged into your USC account to access this form).

Please note that grace days are in place of "excused late" submissions, not in addition to. If you request additional grace days from the instructor, you must have a documented reason for each grace day used to accompany your request. Once you have used your grace days, any late submission will not be accepted and graded as a 0.

Note: There is no grace period. Even if you submit a few minutes after the deadline, you will need to use a grace day (even if the wireless network in your dorm room is down or you have a github issue, etc.). It is your job to be on time and not cut it too close. Remember Murphy's Law and leave time for things to "go wrong."

1.4 Getting Started

1. Download Oracle's VirtualBox, the program that will let you run the virtual machine.
2. Download the VM (we strongly suggest using a wired connection, or sitting close to an access point). Import the virtual machine into VirtualBox using File->Import Appliance.
   (a) **Important: Please do not upgrade the kernel on this VM. It contains kernel modifications necessary to do some of the labs**
3. Start the VM and log in. The user account name is *cs551*, with password *cs551Networks*.
4. If you have not used VirtualBox before, you will need to configure some network information. Follow the instructions here.
5. In the terminal, cd to your home directory: `cd ~/`
6. Clone the git repo that has been assigned to you, using the command:

   ```
   git clone https://github.com/USC-Govindan-Classes/<your-github-login>-cs551.git
   ```

7. After these steps, follow the instructions in the README for building and running the code.

After these steps, you should follow the VM Setup Instructions and then Getting Used to Mininet.

2 VM Setup Instructions

2.1 Using the VM

The VM we have provided you has a graphical user interface, meaning you can use it for your development. However, it is a lot slower than developing on your host OS, so you may want to consider SSHing into it (see below).

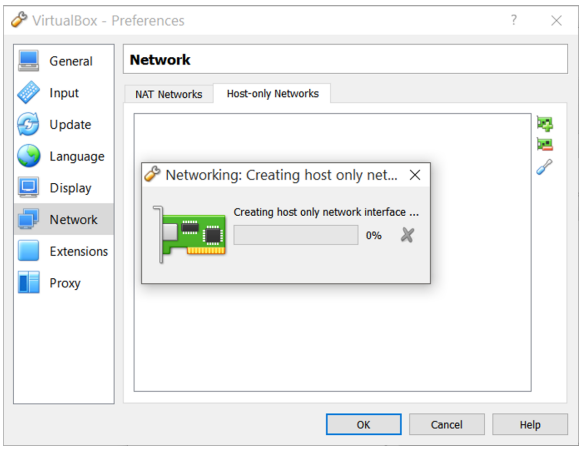
If you change some VM settings (such as increasing RAM, increasing video memory), then the VM won't be as slow.

2.2 Network Settings for VM

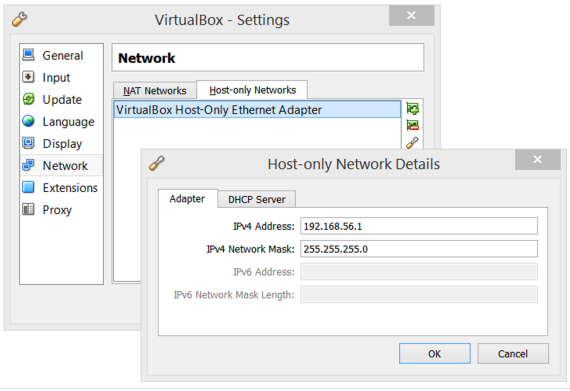
First, make sure the VM is powered off.

In order to use the VM, you will need to set up the network adapter. Under VirtualBox's preferences (File->Preferences or VirtualBox->Preferences in Mac), click on Network, then Host-only Networks. (In newer VirtualBox versions, click on File -> Host Network Manager.)

If nothing is listed, you need to first add an adapter by clicking on the +.

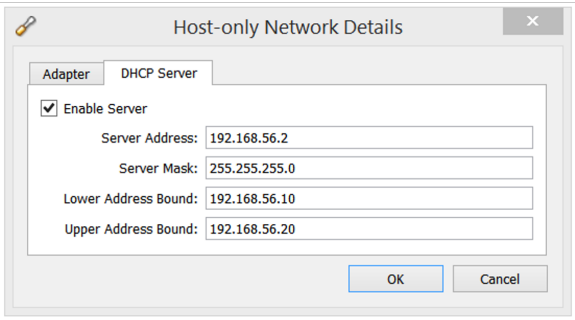


Check that the adapter configuration looks like this (the adapter name may be different):



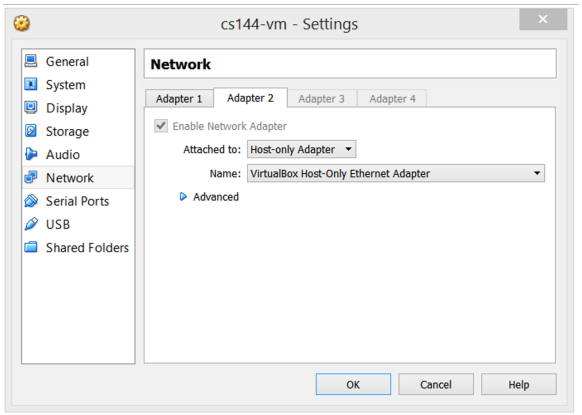
If the IPv4 address is different, you may want to change it to the same value as the one shown above to avoid possible conflicts later on.

You will then need to enable the DHCP server for this adapter. Use the following settings:



This will ensure that your VM receives an IPv4 address within the range `192.168.56.10 - 192.168.56.20` (although it's okay if it doesn't).

The next step is to make sure that the adapter was properly assigned to the VM. Go to the VM settings by clicking on the VM and clicking Settings->Network. Make sure the "Adapter 2" configuration looks like this (the "Name" may be different than the one in this image but should be the same as the name you saw earlier) as shown in the image below. Do ****not** delete "Adapter 1", which will enable you to preserve network access and use git from within your VM.



2.3 SSHing into the VM

SSHing into the VM (the guest OS) from your computer (the host OS) allows you to access the VM through a terminal on your host OS. After setting up the Network Settings (above), you can power on the VM. Open a terminal to check its IP address:

```
ifconfig
```

This will spit out a list with `eth0`, `eth1`, `lo`, etc. Look for an IP address of the form `192.168.56.*` under `inet addr`. For example:

```
% ifconfig
...
eth0        Link encap:Ethernet  HWaddr 00:0c:29:c2:5f:16
            inet addr:192.168.56.11  Bcast:182.168.56.255  Mask:255.255.255.0
            ...
```

Most likely it will appear under `eth0` or `eth1`. If you cannot find the right IP address, and one of eth0 or eth1 is missing, run this command:

```
sudo dhclient eth0
```

(or `eth1`, depending on which one is missing):

```
sudo dhclient eth1
```

If you do `ifconfig` again, an IP address of the form `192.168.56.*` should appear.

Then, you can SSH on Mac or Linux by typing the following in your terminal (on the Mac/Linux side):

```
ssh cs551@192.168.56.11 (or whatever IP address ifconfig gave you)
```

On Windows, you will need to use third-party software (e.g. MobaXterm, PuTTY, or Cygwin).

3 Getting Used to Mininet

3.1 Mininet Testing Environment

The labs rely on two tools: Mininet and POX. Mininet emulates a network with a single router and POX ensures that this router can communicate with your code. To make your job easier, we have written scripts that start up Mininet and POX in the proper order. To become familiar with these tools, you can run this command inside the `lab1` directory:

```
./run_all.sh
```

The script starts Mininet and POX in 2 different screen sessions. You can check that both are running correctly by attaching to each one. In general, it might be a good idea to run this script (and all others in subsequent labs) as root. One easy way to do this is to first run the following command, which gives you a shell as root:

```
sudo bash
```

For Mininet: Attach to the screen using `screen -r mn`. You should see something like this:

```
*** Shutting down stale SimpleHTTPServers
*** Shutting down stale webservers
server1 192.168.2.2
server2 172.64.3.10
client 10.0.1.100
sw0-eth1 192.168.2.1
sw0-eth2 172.64.3.1
sw0-eth3 10.0.1.1
*** Successfully loaded ip settings for hosts
 {'server1': '192.168.2.2', 'sw0-eth3': '10.0.1.1', 'sw0-eth1': '192.
*** Creating network
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
client server1 server2
*** Adding switches:
sw0
*** Adding links:
(client, sw0) (server1, sw0) (server2, sw0)
*** Configuring hosts
client server1 server2
*** Starting controller
*** Starting 1 switches
sw0
*** setting default gateway of host server1
server1 192.168.2.1
*** setting default gateway of host server2
server2 172.64.3.1
*** setting default gateway of client client
client 10.0.1.1
192.168.2.0
172.64.3.0
10.0.1.0
*** Starting SimpleHTTPServer on host server1
*** Starting SimpleHTTPServer on host server2
*** Starting CLI:
mininet>
```

Detach the screen and return to the original terminal by typing `Ctrl-A Ctrl-D` (not just `Ctrl-D`).

For POX: `screen -r pox` should show something like this:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:.home.mininet.cs144_lab3.pox_module.cs144.ofhandler:*** ofhandler: Successfully loaded ip settings for hosts
 {'server1': '192.168.2.2', 'sw0-eth3': '10.0.1.1', 'sw0-eth1': '192.168.2.1', 'sw0-eth2': '172.64.3.1', 'client': '10.0.1.100', 's

INFO:.home.mininet.cs144_lab3.pox_module.cs144.srhandler:created server
DEBUG:.home.mininet.cs144_lab3.pox_module.cs144.srhandler:SRServerListener listening on 8888
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.4/Apr 19 2013 18:28:01)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY.  This program is free software,
and you are welcome to redistribute it under certain conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
Ready.
POX> INFO:openflow.of_01:[Con 1/108531946806346] Connected to 62-b5-90-22-dc-4a
DEBUG:.home.mininet.cs144_lab3.pox_module.cs144.ofhandler:Connection [Con 1/108531946806346]
DEBUG:.home.mininet.cs144_lab3.pox_module.cs144.srhandler:SRServerListener catch RouterInfo even, info={'eth3': ('10.0.1.1', 'b2:bc
('172.64.3.1', '16:bc:76:50:2d:3f', '10Gbps', 2), 'eth1': ('192.168.2.1', '9a:5f:a0:7c:a1:65', '10Gbps', 1)}, rtable=[('10.0.1.100'
th3'), ('192.168.2.2', '192.168.2.2', '255.255.255.255', 'eth1'), ('172.64.3.10', '172.64.3.10', '255.255.255.255', 'eth2')]
```

Once again, detach the screen with `Ctrl-A Ctrl-D`.

Now all that is left to do is run the router logic (i.e. the code you need to write). To check that the setup was done properly, you should start by running the reference solution binary for Lab 1: Build Your Own Router that we provide (`~/cs551/lab1/router/sr_solution` in the VM). You can either run it in the current SSH terminal (which is not a good idea because then you will not have a prompt anymore and will not be able to run any tests), or in a new SSH terminal (simply open a new connection), or in a screen session. Let's see how to do it with screen:

```
mininet@mininet-vm:~/cs144_lab3$ ./sr_solution
Using VNS sr stub code revised 2009-10-14 (rev 0.20)
Loading routing table from server, clear local routing table.
Loading routing table
---------------------------------------------
Destination     Gateway         Mask    Iface
0.0.0.0         10.0.1.100      0.0.0.0 eth3
192.168.2.2             192.168.2.2     255.255.255.255 eth1
172.64.3.10             172.64.3.10     255.255.255.255 eth2
---------------------------------------------
Client mininet connecting to Server localhost:8888
Requesting topology 0
successfully authenticated as mininet
Loading routing table from server, clear local routing table.
Loading routing table
---------------------------------------------
Destination     Gateway         Mask    Iface
0.0.0.0         10.0.1.100      0.0.0.0 eth3
192.168.2.2             192.168.2.2     255.255.255.255 eth1
172.64.3.10             172.64.3.10     255.255.255.255 eth2
---------------------------------------------
Router interfaces:
eth3    HWaddr32:08:c6:65:32:42
        inet addr 10.0.1.1
eth2    HWaddr9e:cc:66:f6:d0:b5
        inet addr 172.64.3.1
eth1    HWaddrae:e8:be:ee:7f:ee
        inet addr 192.168.2.1
 <-- Ready to process packets -->
```

To detach the screen and return to the main terminal, use `Ctrl-A Ctrl-D`. You can return to the session anytime with `screen -r sr`. If you want to kill the session while attached to it, use `Ctrl-D`.

If everything is running correctly, you should be able to run the following command from your ssh terminal:

```
ping 192.168.2.2
```

To test your code, just use your own binary instead of the solution.

3.2 Running Mininet Tests

The Mininet topology is connected to the VM (through interface `eth0`). Therefore, it is possible to execute pings, traceroutes to server1 and server2 from the VM itself. Another good way of running tests is from the Mininet CLI. To get access to it, just attach the Mininet screen session (`screen -r mn`), then type the command you want to execute preceded by the name of the Mininet host (client, server1, server2) you want executing it. For example to `ping` host server1 from host server2, run:

```
mininet> server2 ping -c 3 server1
```

To `traceroute` the router's eth3 interface from server1, run:

```
server1 traceroute -n 10.0.1.1
```

4 Lab 1: Build Your Own Router

4.1 Caveat

PLEASE READ THIS SECTION CAREFULLY.

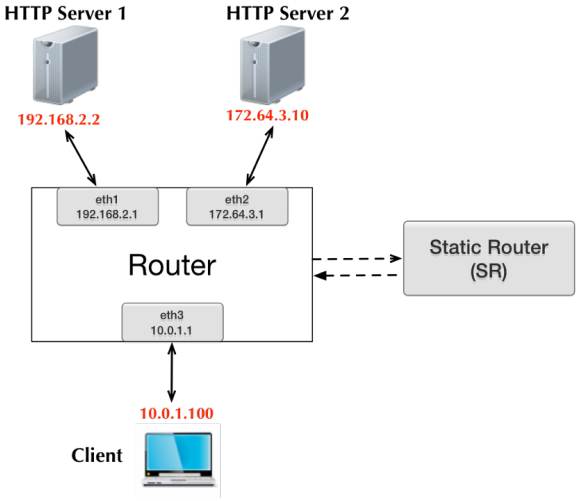
This lab is due before the add/drop deadline. If you have trouble understanding what is required of you for this lab, or find it too difficult, or are unable to devote enough time to the lab, it probably means you lack the necessary background for the class, or have too many commitments this semester. In that case, please consider dropping the class.

4.2 Introduction

In this lab assignment you will be writing a simple router configured under a static routing table and a static networking topology. Your router will receive raw Ethernet frames. It will process the packets just like a real router, then forward

them to the correct outgoing interface. Your task is to implement the forwarding logic so packets go to the correct interface.

Your router will route real packets to HTTP servers sitting behind your router. When you have finished your router, you should be able to access these servers using regular client software (e.g., *wget* or *curl*). In addition, you should be able to *ping* and *traceroute* to and through a functioning Internet router. This is the topology you will be using for your development:



You will use Mininet to set up these topologies of emulated routers and process packets in them. Once your router is functioning correctly, you will be able to perform all of the following operations:

- Ping any of the router's interfaces from the VM (eth1, eth2, eth3);
- Traceroute to any of the router's interface IP addresses (eth1, eth2, eth3);
- Ping any of the HTTP servers from the VM (two HTTP servers);
- Traceroute to any of the HTTP server IP addresses (two HTTP servers);
- Download a file using HTTP from one of the HTTP servers (two HTTP servers).

To implement this functionality, you will need to read the following standards documents (also called, for historical reasons, Internet Request for Comments or RFCs).

- RFC 791: Internet Protocol (IP)
- RFC 792: Internet Control Message Protocol (ICMP)
- RFC 826: Address Resolution Protocol (ARP)

4.3 Getting Started

4.3.1 Starter Code

All the code you need for this assignment is under `lab1` folder of your assigned git repo. You will implement all of your code inside this folder, and commit your lab using git.

4.3.2 Understanding a Routing Table File

Each line in the routing table (`rtable`) file is a routing entry and has the following format:

```
prefix        next_hop     netmask          interface
```

Here is the default routing table that you will find on the VM. The first entry is the default route.

```
    0.0.0.0         10.0.1.100    0.0.0.0           eth3
192.168.2.2     192.168.2.2   255.255.255.255   eth1
172.64.3.10     172.64.3.10   255.255.255.255   eth2
```

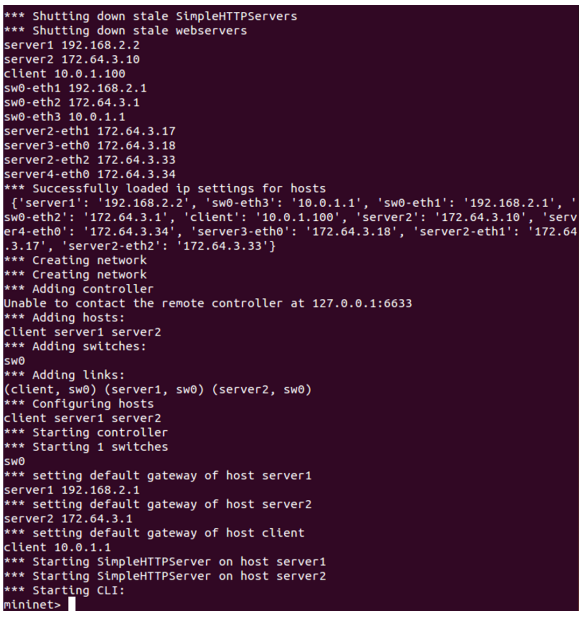
4.3.3 Building and Running

The assignment relies on two tools: Mininet and POX. Mininet emulates a network with a single router and POX ensures that this router can communicate with your code. To make your job easier, we have written scripts that start up Mininet and POX in the proper order. You simply need to run:

`sudo ./run_all.sh`

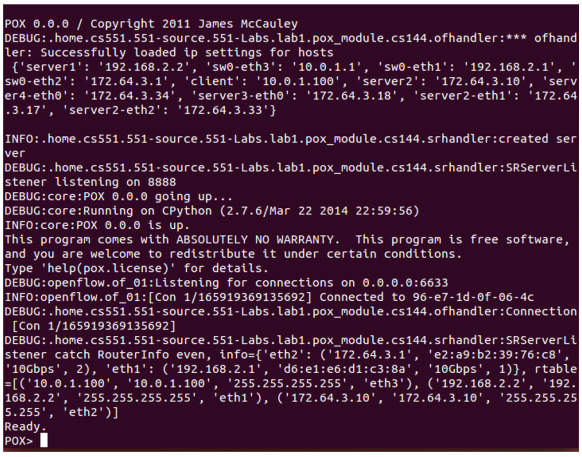
The script starts Mininet and POX in 2 different screen sessions. You can check that both are running correctly by attaching to each one.

For Mininet: Attach to the screen using `sudo screen -r mn`. You should see something like this:



Detach the screen and return to the original terminal by typing `Ctrl-A Ctrl-D` (not just `Ctrl-D`).

For POX: `sudo screen -r` pox should show something like this:

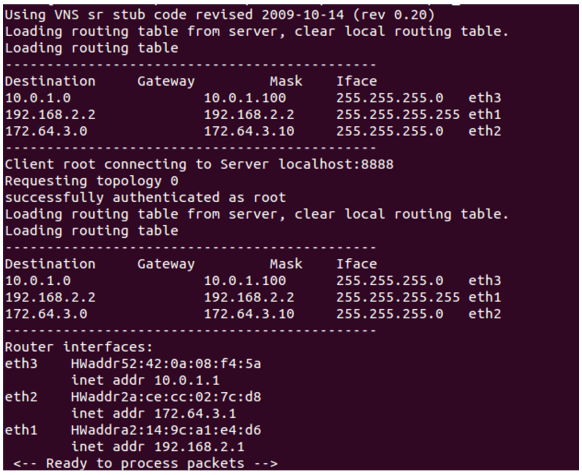


Once again, detach the screen with `Ctrl-A Ctrl-D`.

Now all that is left to do is run the router logic (i.e. the code you need to write). To check that the setup was done properly, you should start by running the reference solution binary that we provide (`lab1/sr_solution` in the VM). You can either run it in the current SSH terminal (which is not a good idea because then you will not have a prompt anymore and will not be able to run any tests), or in a new SSH terminal (simply open a new connection), or in a screen session. Let's see how to do it with screen here (do `chmod a+x sr_solution` to make the solution binary an executable):

```
screen -S sr
./sr_solution
```

The output should look something like this: (there may be small differences, such as different HWaddrs and a slightly different mask for the `172.64.3.0` network).



To detach the screen and return to the main terminal, use `Ctrl-A Ctrl-D`. You can return to the session anytime with `screen -r sr`. If you want to kill the session while attached to it, use `Ctrl-D`.

If everything is running correctly, you should be able to run the following commands from your **mininet screen**:

```
client ping server1
pingall
```

The first command runs a ping from `client` to `server1`, the second command runs a ping between each pair of `client`, `server1` and `server2`. When these commands run, you should be able to see messages on the terminal window where `sr_solution` runs.

Now, to build and test your lab code (see below on what you need to do for the lab), you can simply do as follows:

```
cd router
make
./sr
```

If your implementation is correct, you should see similar output (as our reference solution) when you run the ping commands listed above.

If you run into issues, try `sudo ./killall.sh` and `make clean` first.

4.4 Background: Routing

This section has an outline of the forwarding logic for a router, although it does not contain all the details. There are two main parts to this assignment: **IP forwarding** and **handling ARP**.

When an IP packet arrives at your router, it arrives inside an Ethernet frame. Your router needs to check if it is the final destination of the packet, and if not, forward it along the correct link based on its forwarding table. The forwarding table names the IP address of the next hop. The router must use ARP to learn the Ethernet address of the next hop IP address, so it can address the Ethernet frame correctly.

4.4.1 IP Forwarding and ARPs

Given a raw Ethernet frame, if the frame contains an IP packet whose destination is not one of the router's interfaces:

1. Check that the packet is valid (is large enough to hold an IP header and has a correct checksum).
2. Decrement the TTL by 1, and recompute the packet checksum over the modified header.
3. Find out which entry in the routing table has the longest prefix match with the destination IP address.
4. Check the ARP cache for the next-hop MAC address corresponding to the next-hop IP. If it's there, send it. Otherwise, send an ARP request for the next-hop IP (if one hasn't been sent within the last second), and add the packet to the queue of packets waiting on this ARP request.

This is a high-level overview of the forwarding process. More low-level details are below. For example, if an error occurs in any of the above steps, you will have to send an ICMP message back to the sender notifying them of an error. You may also get an ARP request or reply, which has to interact with the ARP cache correctly.

4.4.2 Protocols to Understand

1. **Ethernet**. You are given a raw Ethernet frame and have to send raw Ethernet frames. You should understand source and destination MAC addresses and the idea that we forward a packet one hop by changing the destination MAC address of the forwarded packet to the MAC address of the next hop's incoming interface.
2. **Internet Protocol**. Before operating on an IP packet, you should verify its checksum and make sure it is large enough to hold an IP packet. You should understand how to find the longest prefix match of a destination IP address in the routing table described in the Getting Started section. If you determine that a datagram should be forwarded, you should correctly decrement the TTL field of the header and recompute the checksum over the changed header before forwarding it to the next hop.
3. **Internet Control Message Protocol**. ICMP sends control information. In this assignment, your router will use ICMP to send messages back to a sending host. You will need to properly generate the following ICMP messages (including the ICMP header checksum) in response to the sending host under the following conditions:

   Echo reply (type 0) Sent in response to an echo request (*ping*) to one of the router's interfaces. (This is only for echo requests to any of the router's IPs. An echo request sent elsewhere should be forwarded).

   Destination net unreachable (type 3, code 0) Sent if there is a non-existent route to the destination IP (no matching entry in routing table when forwarding an IP packet).

   Destination host unreachable (type 3, code 1) Sent after **five ARP requests** were sent to the next-hop IP without a response.

   Port unreachable (type 3, code 3) Sent if an IP packet containing a UDP or TCP payload is sent to one of the router's interfaces. This is needed for *traceroute* to work.

   Time exceeded (type 11, code 0) Sent if an IP packet is discarded during processing because the TTL field is 0. This is also needed for *traceroute* to work.

   Some ICMP messages may come from the source address of any of the router interfaces, while others must come from a specific interface: refer to RFC 792 for details. As mentioned above, the only incoming ICMP message destined towards the router's IPs that you have to explicitly process are ICMP echo requests. You may

want to create additional structs for ICMP messages for convenience, but make sure to use the packed attribute so that the compiler doesn't try to align the fields in the struct to word boundaries: GCC Type Attributes. In addition to the RFC 792, you can also look at ICMP wiki page for the detailed packet format.

4. **Address Resolution Protocol.** ARP is needed to determine the next-hop MAC address that corresponds to the next-hop IP address stored in the routing table. Without the ability to generate an ARP request and process ARP replies, your router would not be able to fill out the destination MAC address field of the raw Ethernet frame you are sending over the outgoing interface. Analogously, without the ability to process ARP requests and generate ARP replies, no other router could send your router Ethernet frames. Therefore, your router must generate and process ARP requests and replies.

   To lessen the number of ARP requests sent out, you are required to cache ARP replies. Cache entries should timeout after 15 seconds to minimize staleness. The provided ARP cache class already times the entries out for you.

   When forwarding a packet to a next-hop IP address, the router should first check the ARP cache for the corresponding MAC address before sending an ARP request. In the case of a cache miss, an ARP request should be sent to a target IP address about once every second until a reply comes in. If the ARP request is sent five times with no reply, an ICMP destination host unreachable is sent back to the source IP as stated above. The provided ARP request queue will help you manage the request queue.

   In the case of an ARP request, you should only send an ARP reply if the target IP address is one of your router's IP addresses. In the case of an ARP reply, you should only cache the entry if the target IP address is one of your router's IP addresses.

   Note that ARP requests are sent to the broadcast MAC address (`ff-ff-ff-ff-ff-ff`). ARP replies are sent directly to the requester's MAC address.

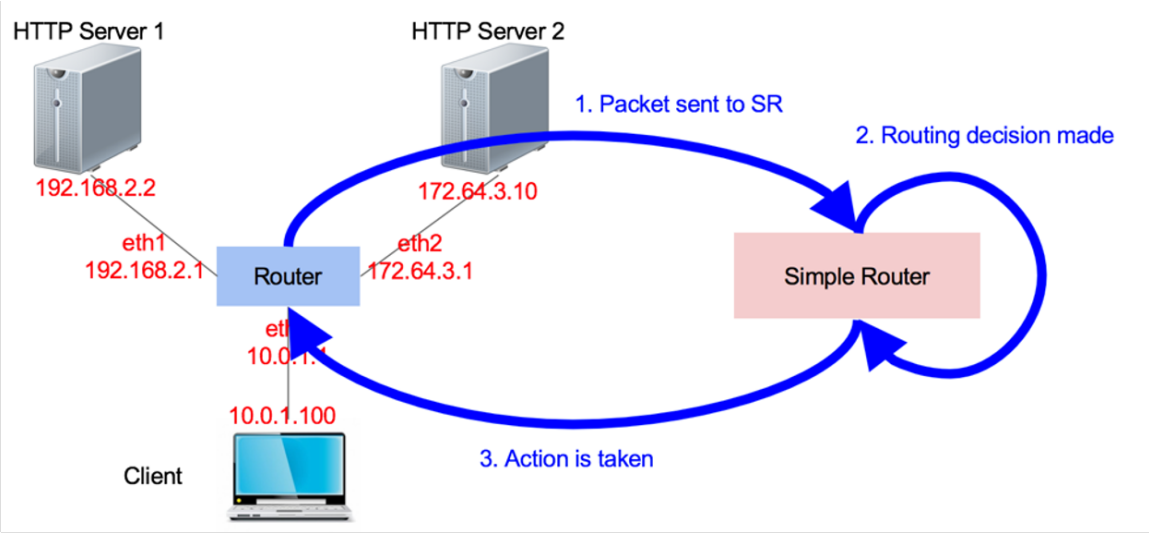
5. **IP Packet Destinations.** An incoming IP packet may be destined for one of your router's IP addresses, or it may be destined elsewhere. If it is sent to one of your router's IP addresses, you should take the following actions, consistent with the section on protocols above:

   - If the packet is an ICMP echo request and its checksum is valid, send an ICMP echo reply to the sending host.
   - If the packet contains a TCP or UDP payload, send an ICMP port unreachable to the sending host.
   - Otherwise, ignore the packet.

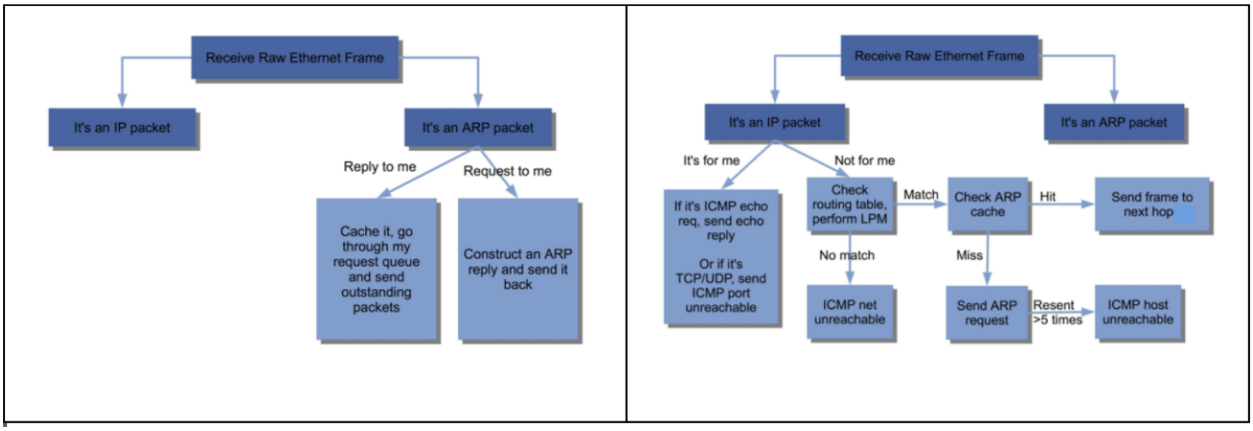
   Packets destined elsewhere should be forwarded using your normal forwarding logic.

4.5 Building your router

4.5.1 Overview



The above figure shows the packet path inside the emulated Mininet environment for this assignment. Your code is responsible for making the routing decision: (a) look at the routing table; (b) figure out which interface to forward the packet to; (c) make necessary changes to packet. Specifically, your router will handle ARP and IP packets. The following two figures show the functional flow chart.



4.5.2 Code overview

1. Basic Functions Your router receives and sends Ethernet frames. The basic functions to handle these functions are:

   - `sr_handlepacket(struct sr_instance sr, uint8_t /packet, unsigned int len, char/ interface)`. This method, located in `sr_router.c`, is called by the router each time a packet is received. The packet argument points to the packet buffer which contains the full packet including the Ethernet header. The name of the receiving interface is passed into the method as well.
   - `sr_send_packet(struct sr_instance* sr, uint8_t buf, unsigned int len, const char* iface)`. This method, located in `sr_vns_comm.c`, will send `len` bytes of `buf` out of the interface specified by `iface`. The `buf` parameter should point to a valid Ethernet frame, and `len` should not go past the end of the frame. You should not free the buffer given to you in `sr_handlepacket()` (this is why the buffer is labeled as being "lent" to you in the comments). You are responsible for doing correct memory

management on the buffers that `sr_send_packet` borrows from you (that is, `sr_send_packet` will not call `free()` on the buffers that you pass it).

- `sr_arpcache_sweepreqs(struct sr_instance *sr)`. The assignment requires you to send an ARP request about once a second until a reply comes back or you have sent five requests. This function is defined in `sr_arpcache.c` and called every second, and you should add code that iterates through the ARP request queue and re-sends any outstanding ARP requests that haven't been sent in the past second. If an ARP request has been sent 5 times with no response, a destination host unreachable should go back to all the sender of packets that were waiting on a reply to this ARP request.

2. Data Structures

   - The Router (`sr_router.h`). The full context of the router is housed in the struct `sr_instance` (`sr_router.h`). `Sr_instance` contains information about topology the router is routing for as well as the routing table and the list of interfaces.
   - Interfaces (`sr_if.c/h`). After connecting, the server will send the client the hardware information for that host. The stub code uses this to create a linked list of interfaces in the router instance at member `if_list`. Utility methods for handling the interface list can be found at `sr_if.c/h`. Note that The server and client here are in the context of the VNS system (which emulates the topology). You can refer to `sr_add_interface` (`sr_if.c`) and `sr_handle_hwinfo` (`sr_vns_comm.c`) for details.
   - The Routing Table (`sr_rt.c/h`). The routing table in the stub code is read on from a file (default filename rtable, can be set with command line option -r) and stored in a linked list of routing entries in the current routing instance (the member name is $\text{routing}_{\text{table}}$).
   - The ARP Cache and ARP Request Queue (`sr_arpcache.c/h`). You will need to add ARP requests and packets waiting on responses to those ARP requests to the ARP request queue. When an ARP response arrives, you will have to remove the ARP request from the queue and place it onto the ARP cache, forwarding any packets that were waiting on that ARP request. Pseudocode for these operations is provided in `sr_arpcache.h`. The base code already creates a thread that times out ARP cache entries 15 seconds after they are added for you. You must fill out the `sr_arpcache_sweepreqs()` function in `sr_arpcache.c` that gets called every second to iterate through the ARP request queue and re-send ARP requests if necessary. Psuedocode for this is provided in `sr_arpcache.h`.
   - Protocol Headers (`sr_protocol.h`). Within the router framework you will be dealing directly with raw Ethernet packets. The stub code itself provides some data structures in `sr_protocols.h` which you may use to manipulate headers easily. There are a number of resources which describe the protocol headers in detail. Network Sorcery's RFC Sourcebook provides a condensed reference to the packet formats you'll be dealing with Ethernet, IP, ICMP, and ARP.

You should only modify `sr_router.c/h` and `sr_arpcache.c/h`. Do NOT modify or add other files.

4.6 Debugging

Debugging is a critical skill in programming and systems programming specifically. Because your error might be due to some tiny, low-level mistake, trying to read through pages and pages of `printf()` output is a waste of time. While `printf()` is of course useful, you will be able to debug your code much faster if you also log packets and use *gdb*.

4.6.1 Protocols: Logging Packets

You can log the packets received and generated by your SR program by using the -l parameter. The file will be in pcap format, so you can use *tcpdump* to read it.

```
./sr -l logfile.pcap
```

Besides SR, you can also use Mininet to monitor the traffic that goes in and out of the emulated nodes, i.e., router, server1 and server2. Mininet provides direct access to each emulated node. Using server1 as an example, to see the packets in and out of it, go to the Mininet CLI:

```
mininet> server1 sudo tcpdump -n -i server1-eth0
```

4.6.2 Router

Using GDB We encourage you to use *gdb* to debug any router crashes, and *valgrind* to check for memory leaks. These tools should make debugging your code easier and using them is a valuable skill to have going forward.

Debugging Functions We have provided you with some basic debugging functions in `sr_utils.h`, `sr_utils.c`. Feel free to use them to print out network header information from your packets. Below are some functions you may find useful:

`print_hdrs(uint8_t *buf, uint32_t length)` Prints out all possible headers starting from the Ethernet header in the packet.

`print_addr_ip_int(uint32_t ip)` Prints out a formatted IP address from a `uint32_t`. Make sure you are passing the IP address in the correct byte ordering.

4.7 Testing

To make sure that you have implemented things correctly, you should run the following command:

```
$ sh run_tests.sh
```

If you pass all of these tests, you will get a full score for this lab, otherwise, you will only get partial credit (depending on how many of the tests you pass).

There are two sets of tests, defined in `base.xml` and `extended.xml`. The latter file uses an extended topology, with two nodes (server3, server4) connected to server2. These tests include:

- Pings and traceroutes between different nodes in the topology
- Iperf to make sure your router implementation is not too slow
- Valgrind to check for memory leaks
- Downloading a large image from each of the HTTP servers
- Checking to see that your implementation sends ICMP host unreachable

You might want to read the XML test source files to understand how the tests work and what they are trying to do. The tests use scripts in the `tester` directory, feel free to look at files in that directory as well.

Some of these tests, in particular, some of the traceroute tests may take a while, so please be patient. If you think your program works correctly, but some of the tests fail, reboot your VM and re-run the tests again.

After executing the tests in each of `base.xml` and `extended.xml`, the testing software will print out your score for each set of tests.

4.8 Frequently asked questions

What if `run_test.sh` does not spawn ./router/sr?

- You might need to try this multiple times.

The test script gets stuck at "waiting for iperf to start up"

- This might be caused by `sr` early termination. Please try multiple times and allocate more memory for the VM (8G would be fine). In grading, we will test your implementation multiple times. As long as your implementation passes the tests once, you will get a full score.

How to debug my implementation if I can not pass a particular test?

- Use `sr_solution` to get the correct packet trace and compare your trace with the correct one to find out what might go wrong.

4.9 Submission

- Final source code: Remember one of the goals of this assignment is for you to build a static router that can perform basic routing functions. Therefore, your final code MUST be runnable as a single shell command (`./router/sr`). We will test your code in the same setup.
- **Report.pdf**: This file should be 1-3 page document describing your **high-level design**, and **implementation**, together with any limitations in your implementation. You should also include this file in the `lab1` directory.

Once you have finished the lab, you should do:

```
git commit -a -m 'Lab 1 submission'
```

```
git push
```

You should also, of course, be committing to your github repo frequently to save your work.

5 Lab 2: Build Your Own Internet

5.1 Introduction

In this assignment, you will learn how to build and operate a layer-3 network using traditional distributed routing protocols, how different networks managed by different organizations interconnect with each other, and how protocols, configuration, and policy combine in Internet routing.

More specifically, you will first learn how to set up a valid forwarding state within an **autonomous system** (AS) using OSPF, an intra-domain routing protocol (Part A). Then, you will learn how to set up valid forwarding state between different ASes, so that an end-host in one AS can communicate with a server in another AS via an intermediate AS (Part B). To do that, you will need to use the only inter-domain routing protocol deployed today: BGP. You will configure both OSPF and BGP through Quagga software routing suite [1], which runs on several virtual routers in your virtual machine (VM).

The rest of the document is organized as follows. We first give you a crash course on how to configure Quagga routers, then describe the setup you will have to use. Then we describe the two parts of the project: configuring OSPF within a domain, and configuring iBGP and eBGP. We conclude the document with how to test your code and how to submit the document.

5.2 Before you begin

All the files that you need for this lab are in the `lab2` sub-directory of your git repo. Before you begin the lab, you should move folder configs inside `lab2` to the `/root` directory. Once you have done this, the path to the configs file should be `/root/configs`. If you do not do this, you will get a lot of errors while working on the lab.

Here is a high-level overview of this lab (see figure below):

- In Part A, we give you the topology of a large ISP(Internet-2, which is the ISP that connects higher education institutions in the US), with some of the details of the topology filled in and we ask you to configure all the routers and hosts in this topology.
- In Part B, we give you a multi-AS topology, where two ASs (East and West) are connected to Internet-2. The West AS is configured such that one of the routers is the static router you developed in Lab 1. We ask you to configure iBGP within Internet-2, and eBGP between East, West and Internet-2. Once you have done this, you will be able to download a webpage at a client host in West from a Web server running on East.

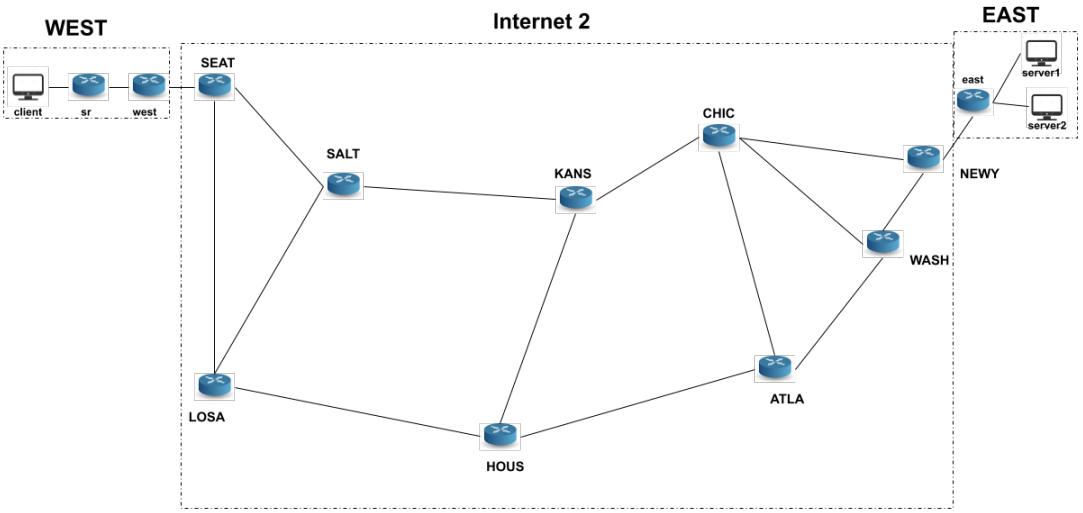


Figure 1: In this lab, you are given this topology, and asked to configure OSPF in Internet2, and BGP between EAST and Internet2 and WEST and Internet2

For Part A, before you begin configuring OSPF, you should, at a terminal window, type the following commands (inside the `lab2` directory):

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ python internet2.py
```

For Part B, before you begin configuring BGP, you should, at a terminal window, type the following commands (inside the `lab2` directory):

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ sh config.sh
$ python multiAS.py
```

Since you will be doing this often, you might want to create shell scripts for your convenience. Even if you do, it would help if you tried to understand what these commands do, so that you can debug any issues that might arise. The above commands will terminate inside mininet, so you should open another terminal to execute following commands. Or you could use detachable tools like screen or tmux.

Once you have started the topology, you should be able to log in to any node in the topology in order to configure it. For example, the Internet-2 topology contains a node called WASH. To access that node, you would use (inside the `lab2` directory: henceforth, any commands we specify for this lab should be run inside this directory, so we will omit this detail in our descriptions below) `sudo ./go_to.sh WASH`. Then, type `vtysh` to access the WASH router. You

can configure it, as described below.

5.3 Configuring IP Routers

Traditional IP routers cannot be configured through OpenFlow. Instead, you need to configure them through a Command Line Interface (CLI). In this lab, we use an open-source routing protocol stack called Quagga. Quagga routers are configured through a CLI, which is similar to the CLIs used in real routers (e.g., from Cisco or Juniper).

In this section, we briefly describe how to configure a Quagga router. However, this is a very short introduction, we strongly encourage you to take a look at the official documentation to get more information.

5.3.1 The Quagga Command Line Interface

When you enter the Quagga CLI (see below for details how to access routers), you should see the following line:

```
router_name#
```

At any time when you are in Quagga CLI, you can use `?` to see all the possible commands that you can type at that point. For example, you should see the following (partial) output when you type ? when you first enter the CLI.

```
router_name# ?
clear   Reset functions
configure   Configuration from vty interface
exit        Exit current mode and down to previous mode
no      Negate a command or set its defaults
ping        Send echo messages
quit        Exit current mode and down to previous mode
show        Show running system information
traceroute  Trace route to destination
write   Write running configuration to memory, network, or terminal
```

Showing configuration: The command `show` will print various snapshots of the router configuration. To see what types of information can be shown, just type `show ?`. For example, `show running-config` will print the configuration that is currently running on the router. You can shorten the commands when there is no ambiguity. For instance, `show run` is equivalent to `show running-config`. Similar to the Linux terminal, you can also use auto-completion by pressing the tab key.

Switching to configuration mode: To configure your router, you must enter the configuration mode with `configure terminal` (`conf t` for the short version). You can verify that you are in the configuration mode by looking for the "config" prefix in your CLI prompt. You should see the following when you are in configuration mode.

```
router_name(config)#
```

If you want to cancel parts of the configuration, you can prefix the command you want to remove with **no**.

5.3.2 Configuring router interfaces

A router interconnects IP networks through several router interfaces. When a router receives a packet from one interface, it forwards it to another interface based on some pre-calculated forwarding decisions. Each interface must have an IP address configured and must be in a different subnet. To configure the IP address for an interface, you will first enter the configuration mode, and then specify the name of the interface you want to configure. For example, you can use the following commands to configure `interface <interface_name>` to `1.0.0.1/24`.

```
router_name# conf t
router_name(config)# interface <interface_name>
router_name(config-if)# ip address 1.0.0.1/24
```

There are several ways to verify the current configuration has been updated correctly. You can use `show run` to examine the current router configuration. You can also show detailed information for interfaces by command show interface. To look at a specific interface, use `show interface <interface_name>`.

Once you have configured an IP address and a subnet to an interface, the router knows that packets with a destination IP in this subnet must be forwarded to this interface. You can use the following commands to show the subnets that are directly connected to your router.

```
router_name# show ip route connected
C>* 1.0.0.0/24 is directly connected, <interface_name>
```

We see that `1.0.0.0/24` is directly connected and reachable with the interface `<interface_name>`. At this stage, any packet with a destination IP that is not in a directly connected subnet will be dropped. If you want your router to know where to forward packets with an IP destination in a remote subnet, you must use routing protocols, such as OSPF or BGP.

5.3.3 Configuring OSPF

OSPF routers flood IP routes over OSPF adjacencies. For Quagga routers, they continuously (and automatically) probe any OSPF-enable interface to discover new neighbors to establish adjacencies with. By default, Quagga router will activate OSPF on any interface whose prefix is covered by a `network` command under the router ospf configuration. For instance, the following commands would activate OSPF on any interface whose IP address falls under `1.0.0.0/24` or `2.0.0.0/24`:

```
router_name# conf t
router_name(config)# router ospf
router_name(config-router)# network 1.0.0.0/24 area 0
router_name(config-router)# network 2.0.0.0/24 area 0
```

OSPF has scalability issues when there is a large number of routers. To mitigate such issues, the router topology can be hierarchically divided into what is called "areas". In this assignment, your network is small and you do not need more than one area: **you will only use the area 0**.

With OSPF, each link between two routers is configured with a weight, and only the shortest paths are used to forward packets. You can use the following commands to set the weight of a link connected to `<interface_name>` to 900:

```
router_name# conf t
router_name(config)# interface <interface_name>
router_name(config-if)# ip ospf cost 900
```

You can use the following command to check the OSPF neighbors of a router:

```
router_name# show ip ospf neighbor
```

```
Neighbor ID Pri State Dead Time Address Interface RXmtL RqstL DBsmL
1.0.0.2 1 Full/Backup 1.0.0.2 newy:1.0.0.1 0 0 0
2.0.0.2 1 Full/Backup 2.0.0.2 newy:2.0.0.1 0 0 0
```

We see that the router has established two OSPF sessions with two neighbors. The first one is connected via the interface `1.0.0.1` and its IP is `1.0.0.2`. The second one is connected via the interface `2.0.0.1` and its IP is `2.0.0.2`. Since you are now connected to two other routers through OSPF, they can send you information about the topology of the network. Let's take a look at the routes received by OSPF using the following command.

```
router# show ip route ospf
O 1.0.0.0/24 [110/10] is directly connected, newy, 07:09:33
O 2.0.0.0/24 [110/10] is directly connected, atla, 06:14:24
O>* 10.104.0.0/24 [110/20] via 2.0.0.2, atla, 00:00:10
```

You can see that our router has learned how to reach the subnet `10.104.0.0/24`. The O at the beginning of each line indicates that the router has learned this subnet from OSPF. To reach it, it must send the packets to its neighbor router `2.0.0.2`. If you want to have more information about the routers of this OSPF area, you can use `show ip ospf` database.

5.3.4 Configuring BGP

While OSPF is only used to provide IP connectivity within an AS, BGP can also be used to advertise prefixes between different ASes. Unlike OSPF, BGP routers will not automatically establish sessions. Each session needs to be configured individually. The following commands show you how to start a BGP process and establish two BGP sessions with neighboring routers. The integer following router bgp indicates your AS-number. Here, the local AS-number is 2.

```
router_name# conf t
router_name(config)# router bgp 2
router_name(config-router)# neighbor 150.0.0.1 remote-as 15
router_name(config-router)# neighbor 2.0.0.2 remote-as 2
```

By default, whenever the remote-as is different from the local number (here, 2), the BGP session is configured as an external one (i.e., an eBGP is established). In contrast, when the remote-as is equivalent to the local one, the BGP session is configured as an internal one. Here, the first session is an eBGP session, established with a router in another AS (`150.0.0.1`), while the second one is internal session (iBGP), established with a router within your AS (`2.0.0.2`). You can check the state of your BGP sessions using the following command.

```
router_name# show ip bgp summary

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
2.0.0.2 4 2 3 6 0 0 0 00:01:16 0
150.0.0.1 4 15 2009 1979 0 0 0 01:31:42 1
```

You can see that our two BGP sessions are up. Notice that if the State/PfxRcd column shows "Active" or "Idle" instead of a number, that indicates that the BGP session is not established.

You might find it useful to manage several BGP neighbors as a group. By doing so, you can apply configurations to the whole group rather than typing similar commands for every neighbor, which could be tedious. Check `peer-group` command in quagga documents for more information.

After BGP sessions are established, you can advertise prefixes using `network` command. For example, the following command advertise a prefix `10.104.0.0/24`.

```
router_name(config-router)# network 10.104.0.0/24
```

Notice that you will only use this command to start advertising prefixes that are directly connected to your network. For prefixes that can only be reached reach indirectly through other networks (e.g. prefixes advertised by your neighbor), you simply propagate the existing advertisements.

After you do that, your neighbor `150.0.0.1` will receive this advertisement and know that it can forward you all the packets with a destination IP in the subnet `10.104.0.0/24`.

You can check the routes learned from other BGP neighbors using the following command:

```
router_name# show ip route bgp
B>* 2.101.0.0/24 [20/0] via 2.0.0.2, interface used, 00:03:17
```

In this example, we can see that neighbor `2.0.0.2` has advertised one prefix: `2.101.0.0/24`. The B at the beginning of the line indicates that the router has learned this prefix from BGP.

5.3.5 Configuring BGP policies

NOTE: You can skip this section, since in Lab 2 we do not configure BGP policies. We have included this material in case you're interested in knowing how policies are actually configured in a router.

After you have BGP sessions running successfully, you might want to configure some BGP policies. For example, you may want to prefer one provider because it is cheaper than another provider. For another example, you may want to avoid sending traffic to one AS for a particular prefix for security reasons. BGP offers several ways to configure such policies. The LOCAL-PREF attribute can help you influence outbound routing, while the MED attribute, AS-Path prepending, or selective advertisements can help you influence the inbound routing. In Quagga, you can use `route-maps` to implement these policies.

1. Route-map Route-maps allows you to take actions on BGP advertisements immediately after an advertisement has been received, or right before being sent to a neighbor. A route-map is composed of route-map entries, and each entry contains three parts: **matching policy**, **match, and set**.

   The matching policy can either be "permit" or "deny". It specifies whether to permit or deny the routes that match the conditions in the match part. Intuitively, only permitted routes will go through the actions in the set part, while denied routes won't be considered further. The match part is a boolean predicate that decides on which route to apply the actions to, and the set part defines the actual actions to take if a route matches.

   Let's take a look on what you can match on:

```
router_name# conf t
router_name(config)# route-map <MY_ROUTE_MAP> permit 10
router_name(config-route-map)# match ?
as-path     Match BGP AS path list
    community       Match BGP community list
    interface       Match first hop interface of route
    ip          IP information
metric          Match metric of route
    origin          BGP origin code
    peer            Match peer address
...         ...
```

   As you can see, you can match pretty much any attribute contained in a BGP UPDATE including: AS-PATH, community value, IP addresses or subnets, etc.

   A BGP community can be seen as a label or a tag that can be attached to any route. You can use this attribute in both the match part and the set part of a route-map. As a convention, a community is often expressed as two separate integers, with the first one identifying the AS number that defined the community (either to use in routes internally within the AS, on routes the AS passes to neighbors, or for neighbors to attach to routes passed to the AS). More than one community can be attached to a route.

   Now let's take a look on what you can do with the matched routes using the set part.

```
router_name# conf t
router_name(config)# route-map <MY_ROUTE_MAP> permit 10
router_name(config-route-map)# set ?
as-path     Transform BGP AS-path attribute
    community       BGP community attribute
    ip          IP information
    metric          Metric value for destination routing protocol
    local-pref      BGP local preference path attribute
    origin          BGP origin code
...             ...
```

This set command enables you to modify any route attributes. Among others, you may find the following attributes most useful in this project: local-pref, community attributes, AS-path (to perform AS-path prepending.) Notice that these operations will only be applied on routes that match the condition specified in match part, if you use the permit clause.

A route-map can contain multiple entries. The order in which entries are processed is given by a sequence number (in the previous example, 10). The entry with the lowest sequence number is executed first. For example, if there are two entries for a route-map, with sequence number 10 and 20 respectively, a route will be first examined using the entry with sequence number 10, and then using the entry with sequence number 20. Notice that by default, if a route matches an entry, it won't be considered for the following entries with higher sequence numbers. Please check the official Quagga document to learn more about the default behaviors and how to change that if needed.

Important: Whenever you use a route-map, an entry that denies everything is implicitly added with the largest sequence number. That is to say, any routes that do not match in the previous entries would by default be denied. You need to add an additional entry to permit any routes to change this behavior.

2. Applying route-maps to BGP sessions Once you have defined a route-map, you can apply it to a BGP session. A route-map can either be applied on incoming routes or on outgoing routes. For example, you can use the following commands to apply a route-map to the outbound advertisements to a BGP neighbor `150.0.0.1`.

   ```
   router_name# conf t
   router_name(config)# router bgp 2
   router_name(config-router)# neighbor 150.0.0.1 route-map <MY_ROUTE_MAP> out
   ```

   The keyword `out` at the end of the last line means that this route-map is applied to the routes your router advertises. The keyword `in` would have applied the route-map to the routes the neighbor is advertising to your router.

3. Example Let's assume multiple routers in your AS (AS 15) have an eBGP session with one of your providers (AS 2). However, you strongly prefer your provider to send traffic to you through one of them. You have agreed with your provider to use a BGP community value equal to `15:100` (15 is your AS number) to indicate the path you prefer.

   The following commands show you how you can configure the route-map and apply it to the preferred router to tag any outgoing routes with such community.

   ```
   router_name# conf t
   router_name(config)# route-map <COMMUNITY_VALUE> permit 10
   router_name(config-route-map)# set community 15:100
   router_name(config-route-map)# exit
   router_name(config)# router bgp 15

   router_name(config-router)# neighbor 2.0.0.2 route-map
   <COMMUNITY_VALUE> out
   ```

   Your provider can check that your prefixes have been advertised with the BGP community value with this command: `show ip bgp community 15:100`. The prefixes listed are the prefixes with the community value `15:100`.

   Tip: Sometimes it takes time for BGP to converge, so you might not see the results corresponding to your changes immediately. You can use `clear ip bgp` to force it to converge faster.

   Now your provider can match on this tag to set the local-preference attribute accordingly and force traffic to exit via the chosen exit point. The following commands show you how your provider can do that.

```
router_name# conf t
router_name(config)# ip community-list standard <TAG> permit 15:100
router_name(config)# route-map <LOCAL_PREF> permit 10
router_name(config-route-map)# match community <TAG>
router_name(config-route-map)# set local-preference 1000
router_name(config-route-map)# exit
router_name(config)# router bgp 2
router_name(config-router)# neighbor 15.0.0.2 route-map <LOCAL_PREF> in
```

You can check that the local-preference has been correctly set with the following command:

```
router# show ip bgp
    Network Next Hop Metric LocPrf Weight Path
    *> 10.104.0.0/24 1.0.0.1 0 100 0 2 i
    *> 15.1.0.0/24 150.0.0.1 0 1000 0 15 i
```

The prefix advertised from AS 15 (`15.1.0.0/24`) with the community value `15:100` now has a local-preference value equals to 1000. In contrast, another prefix (`10.104.0.2/24`) without the community value have the default local-preference value 100.

This section only sketched the basics for route-maps. In this project, one of the learning goals is to get yourself more familiar with route-maps in order to implement the correct BGP policies.

5.3.6 Details on Configuring Quagga

After following the instructions above, you can then use the script `go_to.sh` to switch to a router or a host. For example, with the following command, you will access Internet-2's router LOSA.

```
> ./go_to.sh LOSA
```

LOSA is a router, to access the CLI of LOSA, just use the following command.

```
LOSA> vtysh
```

Type 'exit' to leave the CLI, and another 'exit' to leave the router.

From your VM, you can also go to a host. For example, if you want to go to the host which connects to router SEAT, you can use the following command.

```
> ./go_to.sh SEAT-host
```

When you are in a host, you can use `ifconfig` to see the interface which connects to the router. In this case, the name of the interface is seat.

Important: When you want to configure your router, **DO NOT** edit the config files directly. Instead, always use Quagga CLI.

Important: The current running configurations are not automatically synchronized with the configuration files you can find in the directory configs. **You must synchronize them manually**. To write your current configuration to the configuration files, you can use the following command. You will have to do that for each router.

```
router_name# write file
```

We recommend you to regularly save your configuration (the directory configs) to your own machine, since all your configuration will be reset if we reboot your VM. In case of a reboot, you can quickly put back your configurations in the routers by copy/pasting your configurations into the Quagga CLI. However, see below: in the lab, we ask you to write a Python script to load the saved configurations automatically to each router in the topology.

5.4 The Lab: Part A

In Part A, we ask you to configure OSPF in Internet2 (see Figure 1). In the next part, you will configure BGP at the border routers, and then be able to send traffic end-to-end. Thus, Part B depends on Part A.

5.4.1 Getting Started

To configure OSPF, you need two things: (a) the network topology, and (b) network configuration information (IP addresses, link costs etc.). We provide you with the code that creates a network topology in Mininet (more precisely, in MiniNExT). You can start this topology by typing the set of commands below in a terminal:

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ python internet2.py
```

Once you do this, you can log in to a router using the commands described above. After logging in to the router, you should be able to configure OSPF on Internet2, discussed below.

To configure OSPF, however, you will need configuration information. This configuration information is in the `netinfo` directory and is common both to Part A and Part B. This contains the following files:

`hosts.csv` This lists the hostnames of all the hosts in i2 (which is relevant for this part) and in East and West (relevant for Part B).

`routers.csv` This lists all the routers in each network (i2, East, West) and their loopback addresses. You will need these addresses for iBGP configuration.

`links.csv` This lists all the links in each network, together with interface names, IP addresses and link costs.

`asns.csv` This lists the AS numbers that you should use for each of the 3 ASes. (Used in Part B when you configure BGP).

You should read these files carefully. It might help to visualize these numbers by printing out Figure 1 on a sheet of paper and then annotating IP addresses etc. This annotation will help you avoid configurations, and will also help you interpret your results (e.g., understanding if the traceroute took the correct path).

5.4.2 Configuring OSPF

You are now ready to configure OSPF on Internet2. You should follow these two steps.

1. **Step 1**. Configure OSPF to enable end-to-end connectivity between all the hosts inside your AS. Before configuring OSPF, you will have to configure all the IP addresses for each interface of your routers and hosts.

   For routers, there are several interfaces we already created for you, so you only need to configure the correct IP address to corresponding interfaces using the commands in this section. You can view all the existing router interfaces using the command `show interface` in Quagga CLI. For hosts, you will also have to configure a default gateway (Internet2 has one host connected to each router). For example, if you want to configure the IP address and the default gateway for the host connected to NEWY router, you can use something like the following commands (the actual IP address may be different, you will have to look at the files in the netinfo directory):

   ```
   > sudo ifconfig newy 4.101.0.1/24 up
   > sudo route add default gw 4.101.0.2 newy
   ```

   Make sure that each router and host can ping its directly connected router before you start the OSPF configuration. After you have configured OSPF for each router, test host-to-host connectivity with pings. Do not move on to other steps of the assignment before you verify that every host can ping every other hosts.

2. **Step 2** Assign the OSPF weights based on Figure 1. Use traceroute between each pair of hosts attached to the Internet2 routers, to verify that the paths with smallest costs are used.

3. **What to expect** If your configuration is working correctly, you should be able to:

   · Ping or traceroute from any router to any other router or host
   · Ping or traceroute from any host to any other host or router

   Be careful: these will work only after OSPF has converged. How can you determine if OSPF has converged?

5.4.3 Scripting Configurations

Once you have a working configuration, we would like you to write the following two Python scripts (**Note:** it is important that your script names match the names suggested below, and these names should be placed in the `lab2` folder).

· `load_configs.py`: This script should automate the loading of the configs for all the routers in Internet2. When you save the configs in Quagga using the write file Quagga saves the configs in files with the .sav extension. For example, for each router, it will save configuration files named `zebra.conf.sav` and `ospfd.conf.sav`. Your script should read these files and load them automatically to each router appropriately. To do this, we give you two hints:
  · Look at the code for `go_to.sh` to understand how to run a command at a router.
  · Look at the manual for vtysh to understand how to pass commands to vtysh.
· `config_i2_hosts.py`: This script should automate configuring each Internet2 host's interface and configuring the default route.

5.4.4 Submitting and Testing

You are required to submit the following files for Part A. To do this, do a `git add` for these files, then commit and push them to your github repo:

· The saved configs for each router. You will save these in a directory called `configs` within the `lab2` directory. Inside the configs directory, there should be a directory for each router (e.g., LOSA, WASH), and these should contain `zebra.conf.sav` and `ospfd.conf.sav` files for that router.
· The scripts `load_configs_i2.py` and `config_i2_hosts.py`. If you have done everything correctly, we should be able to type the following commands in the lab2 folder to load the configs to each router and to configure the host interfaces.

```
$ python load_configs.py configs
$ python config_i2_hosts.py
```

To test if you have implemented everything correctly, you can run the following command. This command automatically runs a bunch of tests on your implementation. Note that you should quit all the mininet terminals in your other screens initiated by internet2.py. Otherwise the tester won't start correctly. **If you pass all of these tests, you will get a full grade for Part A.**

```
$ sudo python ../tester/generic_tester.py partA.xml
```

Look inside the `partA.xml` file to figure out what tests we plan to run on your code. It might be a good idea to start testing your code early, don't wait until the last minute to do so. Running these tests take time, so please be patient; in particular, the testing script may wait a minute or more for the network to converge before running the tests. At the end of the run, the tester will print your score for Part A.

5.5 The Lab: Part B

In this part, you are asked to configure BGP between EAST and Internet2 and WEST and Internet2 (see Figure 1). You will also need to configure iBGP inside Internet2.

5.5.1 Getting Started

As with Part A, to configure BGP, you need two pieces of information: (a) the network topology, and (b) the IP addresses and other configuration information. In this part also, we have pre-built the topology for you. We have also already configured all routers **within** the two ASs EAST and WEST. To start the topology for this part, type the following commands, which you might want to automate in a shell script.

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ sh config.sh
$ python multiAS.py
```

To configure BGP, you can use the information in the `netinfo` directory, as described above. In addition to configuring BGP, you will also use the OSPF configuration from Part A for testing.

5.5.2 Configuring BGP

1. **Step 1** Configure internal BGP sessions (iBGP) between all pairs of routers (full-mesh) in Internet2. Please use the interface that connects to a host when specifying a BGP neighbor. These addresses are listed in ****routers.csv in the netinfo directory. You might need to specify using this particular interface when sending announcements to other BGP routers (hint: there is a command to do so). Verify that each of your routers does have an iBGP session with all the other routers with the command show ip bgp summary.

   For the router itself, add the router-id with its own ip address:

   ```
   router_name(config)# bgp router-id <ip_address>
   ```

   For each iBGP neighbors, please also add this command and think about why it is essential for iBGP neighbors.

   ```
   router_name(config-router)# neighbor <ip_address> next-hop-self
   ```

2. **Step 2** In Figure 1, determine where in the topology e-BGP needs to be configured (i.e., between which routers). Configure e-BGP between them. In this lab, we do not apply any routing policies, so you need not configure any policies. However, you should configure e-BGP so that only aggregated prefixes (not /24s) that cover the entire AS are advertised.

3. **What to expect** If you have configured e-BGP correctly, you should be able to ping from `client` in WEST to `server1` or `server2` in EAST. For this to work, you will need to copy the simple router executable file (`sr`) that you developed in Lab 1 into the `lab2` folder. (Important: Lab 2 depends on Lab 1!). Then, you will need to run the following commands on the terminal, after routing has converged (how do you know if routing has converged?):

   ```
   $ screen -S pox -d -m ~/pox/pox.py cs144.ofhandler cs144.srhandler
   $ expect pox_expect
   $ screen -S sr -d -m ./sr
   ```

   As always, you might want to put these commands in a shell script because you might be using these often. Before you do so, try to understand what these commands are doing.

   Once you have ping working, you should try traceroute from `client` in WEST to `server1` or `server2` in EAST. In addition, we have configured web servers on `server1` and `server2`, so you should be able to use HTTP to download files from these servers at any node in the network not just client. Having completed Lab 1, you should be able to figure out which files can be downloaded from these two servers.

5.5.3 Scripting Configurations

Once you have a working configuration, we would like you to write the following Python script (**Note:** it is important that your script names match the names suggested below, and these names should be placed in the `lab2` folder).

`load_configs_multiAS.py` This script should automate the loading of the OSPF and BGP configs for all the routers in Internet2. When you save the configs in Quagga using the write file Quagga saves the configs in files with the .sav extension. For example, for each router, it will save configuration files named `zebra.conf.sav`, `ospfd.conf.sav` and `bgpd.conf.sav`. Your script should read these files and load them automatically to each router appropriately. To do this, see the hints we gave you for Part A. This script will be called with:

```
$ python load_configs_multiAS.py configs_multiAS
```

5.5.4 Testing and Submitting

You are required to submit the following files for Part B. To do this, do a `git add` for these files, then commit and push them[1] to your github repo:

- The saved BGP and OSPF configs for each router. You will save these in a directory called `configs_multiAS` within the lab2 directory. Inside the `configs_multiAS` directory, there should be a directory for each router (e.g., LOSA, WASH, east), and these should contain zebra.conf.sav, ospfd.conf.sav and bgpd.conf.sav files for that router.
- The script `load_configs_multiAS.py`. If you have done everything correctly, we should be able to type the following commands in the `lab2` folder to load the configs to each router and to configure the Internet2 host interfaces (this script is from Part A).
  ```
  $ python load_configs_multiAS.py configs_multiAS
  $ python config_hosts_i2.py
  ```

To test if you have implemented everything correctly, you can run the following command. This command automatically runs a bunch of tests on your implementation. **If you pass all of these tests, you will get a full points for Part B.**

```
$ python ../tester/generic_tester.py partB.xml
```

Look inside the `partB.xml` file to figure out what tests we plan to run on your code. It might be a good idea to start testing your code early, don't wait until the last minute to do so. Also, when running this automated tester, please be aware that some tests might take several tens of seconds or more, especially those that traceroute through your simple router. At the end of the run, the tester will print your score for Part B.

5.6 Frequently asked questions

Why does `ping` not work after configuring all routers?

- It takes time for OSPF/BGP to converge. Wait and retry.

iBGP sessions do not come up even if you followed 5.3.4

- If you use loopback address (5.3.4 is an example without specifying what the ip is, which could correspond to any interface at a router) to establish the session, you need to force the router to use the loopback address to talk to neighbors by using "neighbor update-source loopback" (if you do not use loopback address to establish the session, you do not need to do so). So, after configuring "neighbor <ip> remote-as <AS>", you also need to add the following command "neighbor <peer-group-name> update-source host (loopback address)" (For more, see this).

What if I want to use $sr_{solution}$ in lab2b?

[1] Of course, you should be frequently committing your work to the git repo as you develop your project to avoid losing your work.

- You may, but we will deduct 5% of the score for lab2b.

5.7 References

1. Quagga Routing Suite.
2. Quagga Routing Suite - Documentation.
3. Tmux, a terminal multiplexer.
4. Internet2.
5. Nping.

6 Lab 3: Build Your Own Transport

6.1 Introduction

This lab is in two parts. Please read through the entire lab description before beginning part 1; there are some parts that are easier to build and make more sense with the whole picture in mind.

Your task is to implement a reliable, stop-and-wait (part 1a) and sliding window (part 1b) transport layer on top of the IP layer. Your implementation will result in a minimal, stripped-down version of TCP called cTCP that runs as user-level code (regular TCP code usually runs in the kernel). To cTCP, you will also add a new congestion control protocol called BBR that was designed by engineers at Google and is reported to be in use at Google. cTCP can fully interoperate with regular TCP implementations. You will implement both the client and server components of cTCP.

6.2 Getting Started

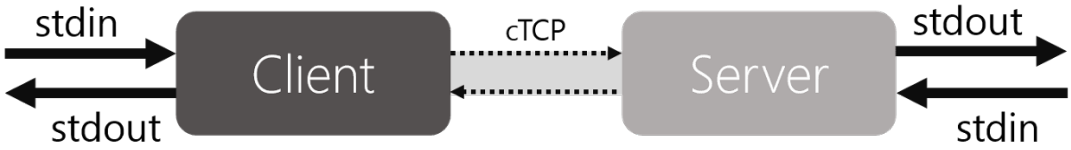
As with other labs, you will use the VM and the git repo provided to you earlier (see VM Setup Instructions).

Implementing cTCP requires being able to send IP packets which your code adds a transport (TCP) header to. Doing this in a way that doesn't confuse the operating system requires some modifications to the OS kernel. You must therefore test your code in the virtual machine that we're providing, since that VM contains the necessary code modifications (described below). You may write code elsewhere, but it must compile and be tested using the VM provided. **Do not upgrade your Ubuntu distribution in the VM**, or you will lose this special kernel and cTCP will stop working.

cTCP creates TCP sockets that the operating system is not aware of. When the kernel receives a TCP segment for a cTCP connection, it does not recognize the connection and immediately sends a reset (segment with the RST bit set) in response, telling the other side to terminate the connection. The modified kernel does not send resets in response to unrecognized TCP segments and so allows cTCP, running in a Linux process, to manage a TCP connection.

6.3 Part 1a: Stop and Wait cTCP

The pictures in this section show how your cTCP implementation will work after you complete Part 1. The client reads a stream of input data from standard input (STDIN), breaks it up into cTCP segments, and sends the segments to the server. The server reads these segments, ignoring corrupted and duplicate segments, outputting the corresponding data, in order, to STDOUT. Connections are bidirectional: the server also reads from STDIN and sends segments, and the client also receives segments and outputs data to STDOUT. BOTH can be sending data segments at the same time!



Each "side", or direction, of the connection can be closed independently. When both sides of the connection are closed, the client and server clean up the associated state (linked lists, structs, anything being used specifically for this connection). This cleanup is often called "tear down", as in "tear down the connection."

Your implementation is TCP-compliant, which means your client can talk to real web servers! In your client, you can type into STDIN and issue an HTTP GET request to a web server (`www.google.com:80` for example):

```
GET / HTTP/1.1
Host: www.google.com
<press enter>
```

And get a response:

```
HTTP/1.1 200 OK
Date: Sun, 17 May 2015 12:00:00 GMT
Expires: -1
...
```

Although cTCP can interoperate with other TCPs, you do not have to implement the TCP state machine; the starter code implements most of this for you. Your implementation only has to handle connection close events (FIN).

Not all websites will respond. Ones that are known to work include the following:

- `www.google.com:80`
- `yuba.stanford.edu:80`
- `sing.stanford.edu:80`
- `www.scs.stanford.edu:80`

6.3.1 Requirements

Delivery:

- Reliable delivery. One side's output should be identical to the other side's input, regardless of a lossy, congested, or corrupting network layer. Ensure reliable transport by having the recipient acknowledge segments received from the sender; the sender detects missing acknowledgements and resends the dropped or corrupted segments. Duplicate segments should not be outputted.
- Timely delivery. Your solution should be reasonably fast. For instance, it should not take 2 seconds for every segment sent from the sender to show up at the receiver.
- Efficient delivery. Your solution should be reasonably efficient: it should not require gigabytes of memory or send many unnecessary segments. E.g., sending each segment 100 times to protect against losses is not a valid implementation strategy.

Robustness

- Large inputs. It MUST correctly handle inputs much larger than the window size.

- Binary files. It MUST be able to transfer binary (non-ASCII) files correctly. Be sure to test this!
- Retransmissions. cTCP should retransmit a segment FIVE times (i.e. send up to a total of SIX times). No more and no less. After that, it should assume that the other side of the connection is not responsive and terminate and tear down the connection.

cTCP

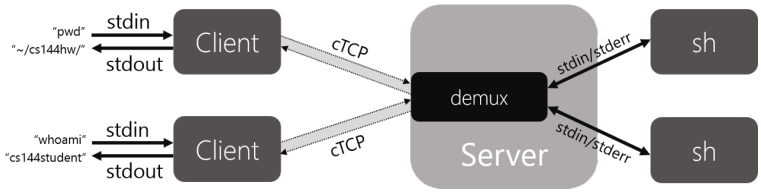
- Connection teardown. When data transmission on one side is completed, it wants to tear down its TCP connection. It is very important to handle teardown properly and gracefully. Imagine a client that has just finished sending data and wants to initiate connection termination (the transport layer notices this by receiving an EOF (Ctrl + D) from STDIN). First, the client sends a FIN segment to the other side to indicate the end of input and enters a state called FIN-WAIT. When at FIN-WAIT state, client continues to receive data from server and process the segments in its queues. However, it does not accept any more data from its application layer (STDIN). In the server side, it will acknowledge the FIN segment and continue to send segments to the client. Finally, if server does not have any more segments to send (including segments either from application or from queues), it will send a FIN to the client. Client will acknowledge the FIN and terminate the connection after waiting for double the maximum segment life to make sure that ACK has been received by the server. Server also tears down the connection after receiving the ack. The procedure is the same when server wants to initiate the tear down. It is important to consider that once all received segments have been outputted and sent segments acknowledged, you may tear down the connection. Like TCP, a connection is not torn down until both sides have finished transferring data, sent FINs, and have their FINs acknowledged. Here is a more detailed description of tear-down process.
  - Note that web servers tend to piggyback ACKs on top of FINs, i.e. you will receive segments with both FIN+ ACK flags and should handle these correctly.
  - To be entirely correct, at least one side should wait around for twice the maximum segment lifetime after sending a FIN and an ACK to a received FIN in case that last ACK got lost (the way TCP uses the FIN_{WAIT} state), but this is not required.
  - When sending an ACK to a FIN segment, the ackno should be 1 greater than the seqno of the FIN. i.e. treat the FIN segment as a "1-byte" segment with no data.
  - A FIN must also be acknowledged (and retransmitted if necessary).
- Cumulative acknowledgements. Recall that TCP uses cumulative acknowledgements. The acknowledgement field of a segment contains the first byte that a receiver has not yet received (i.e. the byte it is expecting next). Your implementation must follow these semantics so it can interoperate with other TCP implementations. Your cTCP implementation does not have to handle sequence number overflowing/wrapping around in the lifetime of a connection. Your connections MUST always use 1 as the initial sequence number. Remember sequence numbers are incremented in terms of number of bytes, not segments. Real TCP connections start with a randomly generated initial sequence number for each connection (you will learn later this quarter about how not doing so can make a connection vulnerable to some attacks). However, for simplicity, sequence numbers for cTCP start at
  1. The underlying library will do the translation into a randomized

  sequence number space.
- Stop-and-wait. Part 1a cTCP implements the stop-and-wait protocol with a window size of `MAX_SEG_DATA_SIZE` (constant defined in ctcp.h). For Part 1a, cTCP needs to support each direction of a connection having only a single segment with data in the network at any time. That segment will be at most (`MAX_SEG_DATA_SIZE` + headers) bytes in size. It may send out as many acknowledgements (non-data segments) as it wants, without having to wait.
- Piggybacked ACKs. Your cTCP needs to be able to receive and output a segment that has both data and ACK. However, your implementation does not need to send piggybacked ACKs and can output data and ACK segments separately.

6.4 Part 1b: Sliding Windows and Network Services

In Part 1b, you will extend cTCP to support multiple client connections to the server, as well as supporting a sliding window of n * `MAX_SEG_DATA_SIZE` (defined in ctcp.h), where $n >= 1$.

In Part 1b, you can initiate several copies of an application on the server side that will output to STDERR. A client can communicate with a server-side application by sending messages to the server, which will pipe the messages to STDIN of the application. The STDERR output of the application is piped back to the server, which will transmit the message back to the client. This lets you hook your cTCP up to programs which act as network services.



Some applications you can test on include (but are not limited to) the following:

- sh
- cat
- date
- grep –line-buffered "your-string-here"
- ./test (found in with the starter package)
  - This is a sample C program that takes input from STDIN and outputs to STDERR. You can modify this to make it whatever you want!
  - To compile this program, you have to run `gcc test.c -o test`

6.4.1 Requirements

Part 1b must support all the functionality required of Part 1a, but also the following additional functionality:

In-order delivery Your server and client should ensure that data is written in the correct order, even if the network layer reordered segments. Your receiver should buffer as many bytes as the client may send concurrently.

Multiple clients Your server should handle multiple client connections. That means it will need to keep state for each client and separate message queues, as well as other supporting structures.

- For simplicity, the server can have at most 10 clients connecting to it simultaneously.
- Note: Connection teardown will not work properly when multiple clients connect. You will not need to handle this case. However, connection teardown should still work if only a single client connects.

Sliding window Support window sizes greater than `MAX_SEG_DATA_SIZE`. You should be able to receive multiple segments at once, each of size at most `MAX_SEG_DATA_SIZE` + headers. You should buffer the appropriate number of segments for retransmissions and output.

- Note: A sliding window of size n * `MAX_SEG_DATA_SIZE` may have more than n segments, if some segments are smaller than `MAX_SEG_DATA_SIZE`. However, the sum of the data size of these segments is less than or equal to n * `MAX_SEG_DATA_SIZE`.
- You should account for the situation when two hosts have different window sizes. For example, host A may have a receive window size of 2 * `MAX_SEG_DATA_SIZE` and is communicating with host B who has a receive window size of 5 * `MAX_SEG_DATA_SIZE`.
- Note: You will not be able to interoperate with web servers with n > 1 (i.e. a sliding window larger than `MAX_SEG_DATA_SIZE` in size).

Application support - Your implementation must be able to talk to an application (like sh) started on a server.

- Note: Connection teardown will not work on the server when an application is running there. You will not be able to type `Ctrl + D`. However, connection teardown must still work for a normal client-to-server connection.

6.5 Implementation Details for Part 1

6.5.1 The Code

One thing to think about before you start is how you will maintain your send and receive windows. There are many reasonable ways to do so, but your decisions here will significantly affect the rest of your code. In Part 1a cTCP only needs to handle one outstanding segment, but in Part 1b it will need to handle a send window as well as a receive window. Cumulative acknowledgements mean the send window can be a simple FIFO queue, but lost segments mean the receive window will need to handle when there are holes. Memory management in systems software is extremely important, so think through where and when you will allocate and free any dynamic structures you need.

6.5.2 Files

Here are all the files that come in the starter code. Feel free to make new helper files as needed, but if you do, make sure to add them to the Makefile. We encourage you to look at the starter code if you have questions on how it works: it is not very long.

- `ctcp.h` - Header file containing definitions for constants, structs, and functions you will need for the cTCP implementation. Please read this file carefully because the comments are very useful!
- `ctcp.c` - This is the **only** file you will need to modify. Contains the cTCP implementation.
- `ctcp_linked_list.h/ctcp_linked_list.c` - Basic linked list functions. Optionally use this to keep track of unacknowledged segments, etc.
- `ctcp_sys_internal.h/ctcp_sys_internal.c` - Internal library that handles cTCP to TCP translations, etc. You will not need to look at this code at all and will not be using anything from it.
- `ctcp_sys.h` - Definitions for system and connection-related functions like reading input, writing output, sending a segment to a connection, etc. You will be using most of these functions!
- `ctcp_utils.h/ctcp_utils.c` - Helper functions like checksum, current time, etc.
- `test.c` - Sample application that can be used. Feel free to modify this however you want. It will not be used in grading.
- Makefile - No need to edit unless you're adding new files.
- README - Information on how to build and run cTCP
- `ctcp_README` - You will fill this out. See Section 6's cTCP README.

6.5.3 Data Structures

1. **Segments** You will be sending and receiving cTCP segments (which are like TCP segments with a few fields taken out for simplicity – the library will transform cTCP segments into actual TCP segments. You can assume no TCP options will be used.). This struct is defined in `ctcp_sys.h`.

```
typedef struct ctcp_segment {
uint32_t seqno;    /* Sequence number (in bytes) */
uint32_t ackno;    /* Acknowledgement number (in bytes) */
uint16_t len;      /* Total segment length (including ctcp header) */
uint32_t flags;    /* Flags */
uint16_t window;   /* Window size, in bytes */
uint16_t cksum;    /* Checksum */
char data[];       /* Start of data */
} ctcp_segment_t;
```

Every segment will have a 32-bit sequence number (seqno), a 32-bit acknowledgement number (ackno), as well as 0 or more bytes of payload (data). All fields (other than data) must be in network byte order (meaning you will have to use htonl()/htons() to write those fields and ntohl/ntohs to read them).

- seqno: Each segment transmitted must be numbered with a sequence number. The first segment in a stream has a sequence number of 1. The sequence number indicate bytes. That means if you send a segment with 10 bytes of data that has a seqno of 10000, then the next segment you send should have a seqno of 10010.
- ackno: Cumulative acknowledgement number. This says that the sender of the segment has received all bytes earlier than ackno, and is waiting for a segment with a sequence number of ackno. Note that ackno is the sequence number you are waiting for that you have not received yet.
- len: Length of the segment. Should include the header and payload data size.
- flags: TCP flags. This should be 0 for normal segments, and ACK for ack segments (you can set its value by doing `segment->flags | ack_flag`, where `ack_flag` is the ACK flag converted to the right byte-order). To signal the close of a connection, you must use the FIN flag. Flags are defined in ctcp.h.
- window: The TCP window size. Should be an integer multiple of `MAX_SEG_DATA_SIZE` (defined in ctcp.h).
- cksum: 16-bit cTCP checksum. To set this, you must set the cksum field to 0, then call on the `cksum()` function (defined in `ctcp_utils.h`) over the entire segment. Note that you shouldn't call `htons()` on the checksum value produced by the `cksum()` function – it is already in network byte order.
- data: Payload to the cTCP segment. This should be empty for FIN/ACK-only segments.

2. **Connection State** The `ctcp_state_t` structure (in `ctcp.c`) encapsulates the state of each connection. You should add more fields to this struct as you see fit to keep your per-connection state. Some fields that might be useful to keep include the current sequence number, acknowledgement number, etc. A new `ctcp_state_t` is created by `ctcp_init()`, which is called by the library when there is a new connection.

```
    struct ctcp_state  {
    struct ctcp_state *next;   /* Next in linked list */
    struct ctcp_state **prev;  /* Prev in linked list */

    conn_t *conn;      /* Connection object -- needed in order to
                          figure out destination when sending */
    linked_list_t *segments;  /* Segments sent to this connection*/

    /* FIXME: Add other needed fields. */
};
```

This `ctcp_state_t` struct is passed into cTCP functions so it can be used to store received segments, etc. There is also a global list of `Ctcp_state_t` structs called `state_list` (in `ctcp.c`) that can be used to figure out which segments need resending and which connections to time out.

3. **Configuration** The `ctcp_config_t` struct (in ctcp.h) contains values to adjust your cTCP implementation.

```
    typedef struct {
    uint16_t recv_window;  /* Receive window size (a multiple of MAX_SEG_DATA_SIZE) */
    uint16_t send_window;  /* Send window size (receive window size of other host) */
    int timer;             /* How often ctcp_timer is called, in ms */
    int rt_timeout;        /* Retransmission timeout, in ms */
} ctcp_config_t;
```

It is passed in when there is a new connection (in `ctcp_init()` in ctcp.c). You may want to store this struct or its values in a global variable or within the `ctcp_state_t` struct for later access.

6.5.4 Functions

1. **Functions To Implement.** Your task is to implement the following functions in ctcp.c (more details about what they should do can be found in ctcp.h):

   - `ctcp_init()`: Initialize state associated with a connection. This is called by the library when a new connection is made.
   - `ctcp_destroy()`: Destroys connection state for a connection. You should call either when 5 retransmission attempts have been made on the same segment OR when **all of the following** hold:
     - You have received a FIN from the other side.
     - You have read an EOF or error from your input ($\text{conn}_{\text{input}}$ returned -1) and have sent a FIN.
     - All segments you have sent have been acknowledged.
   - `ctcp_read()`: This is called if there is input to be read. Create a segment from the input and send it to the connection associated with this input.
   - `ctcp_receive()`: This is called when a segment is received. You should send ACKs accordingly and output the segment's data to STDOUT.
   - `ctcp_output()`: This should output cTCP segments and is called by `ctcp_receive()` if a segment is ready to be outputted. You should flow control the sender by not acknowledging segments if there is no buffer space for outputting.
   - `ctcp_timer()`: Called periodically at specified rate. You can use this timer to inspect segments and retransmit ones that have not been acknowledged. You can also use this to determine if the other end of the connection has died (if they are unresponsive to your retransmissions).

2. **Helper Functions.** The following functions are needed when implementing the above functions (the following functions can be found in `ctcp_sys.h`):

   - `conn_bufspace()`: Checks how much space is available for output. `conn_output()` can only write as many bytes as reported by `conn_bufspace()`.
   - `conn_input()`: Call on this to read input that then needs to be put into segments to send off.
   - `conn_send()`: Call on this to send a cTCP segment to a destination associated with a provided connection object.
   - `conn_output()`: Call on this to produce output to STDOUT from the segments you have received. You will first have to call on `conn_bufspace()` to see if there is enough space for output.

   Linked list helper functions can be found in `ctcp_linked_list.h`. Other helper functions can be found in `ctcp_utils.h`.

6.6 Testing

6.6.1 Building and Running the Code

More specific details on how to build and run can be found in the README included with the starter code. To build the code, simply run make in the command-line.

1. Server Mode

```
sudo ./ctcp -s -p [server port]
sudo ./ctcp -s -p [server port] -w [server send window multiple] --
[program args] (Part 1b)
```

2. Client Mode

```
sudo ./ctcp -c [server]:[server port] -p [client port]
sudo ./ctcp -c [server]:[server port] -p -w [client send window multiple] (Part 1b)
```

6.6.2 Debugging

Note: Any debug statements should be printed to STDERR, not STDOUT. This is because STDOUT is being used for the actual program output and it will be confusing for the grader. Your final solution must not output any debugging information (as it will cause tests to fail).

To output to STDERR, use the following function:

```
fprintf(stderr, "Debug message here", ...);
fprintf(stderr, "Here's an int: %d", int_number);
```

We encourage you to use gdb and valgrind. These tools should make debugging your code easier and using them is a valuable skill to have going forward.

6.6.3 Logging

Logging can be enabled with -l on one host to view a log of all segments sent and received:

```
sudo ./ctcp [arguments] -l
```

This will create a .csv file of the form [timestamp]-[port], where timestamp is the timestamp that the host started, and port is the port associated with this host. This should have all the header information you need to track segments and to see if the right fields are set.

The Data field contains a hex dump of the data sent. You can convert to this to ASCII using any hex-to-ASCII converter.

6.6.4 Interoperation

Your program must interoperate with the reference implementation, reference, included in the assignment package. It must also be able to communicate with web servers (such as www.google.com). Some web servers may not respond; however, it must work with google.com.

6.6.5 Unreliability

Since connections between machines are pretty reliable, we've included tools that will help you simulate an environment with bad network connectivity. They allow you to drop, corrupt, delay, and duplicate segments that you send over the network.

To invoke the tools, run your program with one or more of the following command-line options:

```
--drop <drop percentage>
--corrupt <corrupt percentage>
--delay <delay percentage>
--duplicate <duplicate percentage>
```

The percentage values are integers in the range 0-100 (inclusive), and they determine the probability that an outgoing segment will be affected by the associated action. You can use the following flag to set the random seed value:

```
--seed <seed value>
```

For example, running the following command will drop 5% and corrupt 3% of all outgoing segments:

```
./ctcp --drop 5 --corrupt 3 -c localhost:5555 -p 6666
```

If you also include the -d command-line argument, the program will print out more details about the alterations that are being made to your segment.

The tools will alter your segments deterministically. This means that if you run your program twice using the same seed value, the segments will be affected in an identical fashion. This should make it easier for you to reproduce bugs in your code and help you in the debugging process.

Running your program with these flags will only affect outgoing segments. If you would like your incoming segments to be affected as well, you will need to run both instances of cTCP with the command-line flags.

6.6.6 Large/Binary Inputs

Don't assume that your implementation will always work correctly if you've only tested it by typing messages in the console. Try testing with large files (students have used novels from Project Gutenberg), and test with binary executable files as well (cTCP itself!). You can use input/output redirection to pass large files. **Make sure** you use these options carefully as they overwrite the contents of the file.

```
ctcp-server> sudo ./ctcp [options] > newly_created_test_binary
ctcp-client> sudo ./ctcp [options] < original_binary
```

To verify the transfer is correct, use the cmp utility to check the two files match (read the manual by running man cmp). To look at their contents in detail, you can use od -h.

6.6.7 Memory leaks

Your submission at the end of Part 1b should be free of memory leaks. If there are leaks, we will deduct points for your submission. To test for memory leaks, you can run the following command:

```
sudo valgrind --leak-check=full --show-leak-kinds=all ./ctcp [params for cTCP]
```

6.6.8 Tester

We will provide a tester which will run your program against several tests. This is a Python script that must be run in the same directory as the reference binary and your code. It will assume your code compiles into a binary called **ctcp**, which should be the case if the Makefile was not modified. These tests are a subset of the ones we will use for grading. They are not comprehensive and you should do your own extensive testing. However, passing all the tests we give you is a good indication of your program's robustness. **Make sure to remove all print statements before running the tester!**

```
sudo python ctcp_tests.py
sudo python ctcp_tests.py --tests [specific test numbers to run] --timeout [test timeout in secs]
sudo python ctcp_tests.py --tests 1 2 3
```

You can increase the test timeout if some tests are failing (as there is still some randomness and test results may differ slightly with each run). However, if your tests only pass consistently with a large timeout (longer than 20s), then there is a problem with your code.

To get a list of all tests:

```
sudo python ctcp_tests.py --list
```

The script will print out your score for these tests.

6.6.9 Testing on real topologies

The tests above do not run CTCP on a real network topology. After passing (most) of the tests in `ctcp_tests.py` you should try to run it on (a) a dumbbell topology on mininet, and (b) the mini-Internet you defined in Lab 2.

Dumbbell topology. A dumbell topology has two routers connected to each other in the middle, with 2 or more clients attached to one of the routers, and 2 or more servers attached to the other router. Then, we can test ctcp between each

client server pair, or between two clients and the same server. If you pass the previous tests, then, your CTCP should pass the following test:

```
sudo python ../tester/generic_tester.py ctcp_mininet.xml
```

Take a look at the XML file, which describes what tests we have defined.

To debug this (or the Mini-Internet topology below), instead of running the tester script, you can also execute the `lab3_topology` and start the client and server by hand in Mininet (using the commands above for starting ctcp servers and clients). If you do this, be sure to add a *-m* flag to the command line, like so:

```
sudo ./ctcp -m -s -p [server port]
sudo ./ctcp -m -c [server]:[server port] -p [client port]
```

Mini-Internet topology. At this point, you should be able to run CTCP on the mini-Internet you set up in Lab 2. To try this, do the following:

```
cd ../lab2
sudo python ../tester/generic_tester.py ctcp.xml
```

This test runs a CTCP client on a node in the West AS, and a server on a node in the East AS, and downloads a large file across the network. The packets flow through the simple router that you developed. If you pass this test, you should feel proud: you have built your own version of the Internet!

6.6.10 FAQ

Most questions and answers can be found on Piazza. This is merely a list of logistical questions.

1. Should ACKs be piggybacked on top of outgoing data segments?
    - Piggybacking ACKs is preferable but not required and you will neither gain nor lose points either way. However, you do need to handle receiving acks + data from reference and webservers.
2. Can we assume that the len field in `ctcp_receive()` is correct?
    - No, you must examine the length field of the segment and should not assume the cTCP segment you receive is the correct length. The network might truncate or pad segments.
3. Why is a double pointer prev used for the `ctcp_state_t` linked list?
    - This is a clever trick that enables it to delete elements of a linked list efficiently with a bit less pointer rewiring than if prev were a single pointer.
4. What sliding window protocol are we supposed to implement for Lab 3?
    - You MUST use selective repeat.
5. Why do all commands have to be run with sudo?
    - cTCP uses raw sockets, which require root in order to open. You can avoid having to type in sudo each time if you run the shell as root (sudo bash).
6. What's the difference between input/receive and output/send?
    - Input/output is done locally, whereas send/receive is done over a connection.
        - Input is the term we use for text/binary received on a client via STDIN.
        - Receive is when a segment is received from a connection.
        - Output is the text/binary printed out to STDOUT.
        - Send is when a segment is transferred from one client to another.
7. What should I do if an error occurs (e.g. `conn_output()` returns -1)?

- It's up to you. We suggest terminating the connection (via a call to `ctcp_destroy()`).

8. How do you get the EOF to show up?
   - The reference solution does this by doing `fprintf(stderr, ...)` when it reads in an EOF (Ctrl + D) from STDIN. You can add this to your code too.
9. How come when connecting to a webserver, the data I send is 1 byte bigger than what I expect?
   - Webservers expect messages that end with `\r\n`; however, this is difficult to input into the terminal. When communicating with webservers, the library automatically converts all `\n` s to `\r\n` s (hence the 1 extra byte).
10. Will I need to handle overlapping segments?
    - No, you will not need to handle the case when segments overlap (i.e. a new segment has bytes that partially overlap with the bytes of the previous segment).
11. Why am I getting valgrind memory leaks from the starter code (e.g. in `convert_to_ctcp()` or `start_client()`)?
    - Make sure you are freeing all segments passed in from `ctcp_receive()` and freeing the `ctcp_config_t` struct passed into `ctcp_init()`.

6.7 Part 2: BBR Congestion Control

BBR is a new congestion control protocol developed for the Internet. It is described in this paper. The paper also contains sample pseudo-code for BBR which should help you implement BBR. Finally, the patches to the Linux kernel for BBR are also publicly available, which you can take a look at.

You are expected to implement add BBR to cTCP, both to the client and the server side.

6.7.1 Understanding BBR

As we studied in class, TCP's native congestion control has some shortcomings:

1. Packet loss does not indicate Congestion: TCP reduces its congestion window (`cwnd`) when packet loss is detected then TCP will assume that current network is congested. However, in reality, packet loss is not only caused by congestion but also network error (e.g. wrong checksum, noise, etc). Therefore, some packets lost doesn't mean current tunnel is crowded. But due to TCP's congestion avoidance mechanism, the `cwnd` will be decreased so TCP cannot fully utilizing the tunnel. Therefore TCP doesn't fit perfectly with today's data center networks, which usually have high bandwidth and frequent packet loss.

2. No packet loss also does not indicate No congestion: If there's few packet loss, traditional TCP's `cwnd` will fast increases then occupies all the buffer space on the receiver side. This indeed fully utilizes the bandwidth, but leaves a long queue for receiver to process, which increases RTT. When TCP was invented (1980s), the receiver buffer was quite small so filling up the buffer is not a big issue. However, today's hardware supports GB-sized buffers. If we still use the aggressive policy to fill the buffer, RTT will be unacceptable.

BBR does congestion control differently:

1. Decoupling "reliable packet transmission" and "congestion control" in TCP. BBR lets TCP only work on the first part and BBR estimates the congestion control policy.
2. Using Bandwidth-Delay-Product (BDP) to estimate the traffic condition, i.e. the congestion condition, instead of separately estimating packet loss or delay.
3. Trying to fullying utilize the bandwidth while keeping buffer occupation low, which minimizes the RTT.
4. Besides `cwnd`, BBR uses a new concept, pacing, to control burstiness in sending packets.

6.7.2 BBR Logic

Since packet loss is a bad indicator of congestion condition, we need the "true" indicator, BDP. Therefore, we should accurately estimate max(bandwidth) and min(RTT). The first condition guarantees that TCP can run at 100 percent bandwidth utilization. The second guarantees there is enough data to prevent receiver starvation but not overfill the buffer. These two values change separately.

BBR computes RTT like this: When the sender received a ACK for a previous sent packet. Then the RTT will be estimated as:

```
min_RTT = min(min_RTT, cur_time - pkt_sent_time)
```

Similarly, BBR computes bandwidth as:

```
max_BW = max(max_BW, (data_delivered - data_before_pkt) / (cur_time -
pkt_sent_time) )
```

Note that if the data rate on the sender side is application-limited, the calculated bandwidth will be less than actual bandwidth.

With $\min_{RTT}$ and $\max_{BW}$, BBR is able to calculate the BDP, the remaining space in the tunnel:

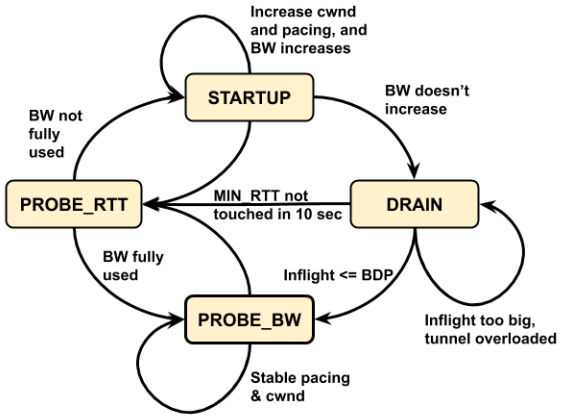
```
BDP = min_RTT * max_BW
```

By comparing BDP with inflight (the data sent but not acked) and monitoring `min_RTT` and `max_BW`, BBR will know which state it is currently in so it can adjust the sending strategy, which is controlled by `cwnd` and pacing rate. `Cwnd` defines the maximum data sender can send within a period of time. Pacing, added by BBR, is used to tell the sender how it should send this piece of data. This is important because there may be burst if the `cwnd` is big and the sender sends the data altogether. The details of choosing `cwnd` and pacing are well explained in the BBR code patch.

The paper doesn't discuss fairness between standard TCP and BBR in different situations. You can test in your code if you are interested.

6.7.3 BBR Code Organization

You should first look at the main function of BBR patch: `bbr_main`, which operates a state machine that can be described as the graph below:



Four states explained:

- `STARTUP`: exponential growth to quickly fill pipe (like slow-start), will stop growth when bw estimate plateaus, not on loss or delay
- `DRAIN`: drain the queue created in STARTUP stage
- `PROBE_BW`: cycle `pacing_gain` to explore and fairly share bandwidth
- `PROBE_RTT`: occasionally send slower to probe min RTT

All the implementation details are well explained in the inline comments of the patch code.

6.7.4 Incorporating BBR to your CTCP implementation

After understanding BBR code, you can add it to your ctcp implementation. You are encouraged to look at the BBR kernel code patches (yes, there is a reference code this time!). But simply copying the code will not work because it is not designed for your CTCP implementation. You should rewrite the code yourself, so you can understand better how BBR works.

We require you to write your BBR code in separate files. To do this, create two files `ctcp_bbr.h` and `ctcp_bbr.c` under the same directory as the rest of the CTCP code (you should already have done this in part 1). Add all your BBR code to these two files. You should also update your ctcp.c and ctcp.h to enable BBR functions.

6.7.5 Testing BBR

We do not provide a tester for BBR. You should try to create your own test cases for BBR to make sure your submission works correctly. In particular you should try the ctcp tests described above: with packet losses, and for large binary inputs. You should so try to draw graphs similar to the ones in the BBR paper (e.g., BBR's estimated BDP vs time) to understand if your implementation works correctly. You can also create simple topologies in Mininet, such as the dumb-bell topology discussed in class, to test whether BBR behaves correctly in the presence of multiple flows.

Every time your code runs, it should output a log file that contains the timestamp and BDP. The file name should be "bdp.txt" and the format of each line is "timestamp BDP". When your ctcp sends a packet, it should append a new line to the file with current timestamp (get it from `current_time()` in `ctcp_utils.h`). You only need to log BDP for sending so you don't need to log for re-sending. The BDP should be ****expressed in units of bits.** This will help us generate graphs to test if your implementation is working correctly. Here is what your output should look like. The BDP and the timestamp should be separated by commas.

```
1508350555362,95820
1508350555389,96181
1508350555408,96374
```

1. **Plotting BDP** A log file, bdp.txt, can be used to evaluate your BBR implementation. We provide a simple script, `lab3_plot_bdp.py` in the github repo under lab3 to plot BDP of your BBR implementation.
2. **Dumbbell topology**

   You will use the same dumbbell topology to test your implementation as part 1. First, you should start the topology by running:

   ```
   sudo python lab3_topology.py
   ```

   Then, you can start the client and the server with:

   ```
   sudo ./ctcp -m -s -p [server port]
   sudo ./ctcp -m -c [server]:[server port] -p [client port]
   ```

   You can start two servers and two clients together and check your BDP output.

3. **Checksum issue** Mininet may cause checksum error when it is used to test cTCP. If you experience the checksum error at the client side, you can disable checksum in your cTCP code and report the issue in your report.
4. **Grading**
   (a) Write a report with your code's BDP plot
   (b) Test your BBR-CTCP on dumbbell topology:
      i. In one terminal, go to `lab3` folder and run `sudo python lab3_topology.py`.
      ii. In another terminal, go to `lab3` which should have two files `file.txt` and `bbr_mininet.sh`.
      iii. In the same folder, run the script `sudo ./bbr_mininet.sh [551_HOME_PATH]` (the `551_HOME_PATH` should be FULL path, not relative path).
   (c) Test your BBR-CTCP on I2 topology:
      i. Go to lab2/ which should have `bbr_i2.sh`.
      ii. Copy `file.txt` here.
      iii. Run the script `sudo ./bbr_i2.sh [551_HOME_PATH]`.

6.8 Frequently asked questions

Unable to connect to a web server: null connection

- please try to commit your changes, reinstall your VM and start over

I am using the reference ctcp to debug the mininet-related tests. But it failed.

- The reference provided does not support [-m] option, which is required in the mininet-related test cases. So please do not use the reference (which can pass all basic test cases).

How could I debug my code for the mininet-related tests?

- Change `2> /dev/null` to something like `2> /tmp/client_logs` in the `run_tests.py` (line 350 and 363) so that you can see the `fprintf(stderr...)` in those log files (make sure you use /tmp/server$_{\text{logs}}$ for the server command).

6.9 Submission

You will submit Part 1 and Part 2 separately. For each part, you will need to prepare a Report.pdf file and submit your code by committing to your git repository. Please name these files `Report_1.pdf` and `Report_2.pdf`.

6.9.1 Reports

Each report should be 2-3 page document (a page being 30-40 lines) with no more than 80 characters per column to make it easier to read. This file should be included with your submission. It should contain the following sections:

- Program Structure and Design - Describe the high-level structure of your code, by insisting on what you actually added to the code. You do not need to discuss in detail the structures that you inherited from the starter code. This should be the longest and most important part of the README. Use this as an opportunity to highlight important design aspects (data structures, algorithms, networking principles) and provide greater detail on those areas for your grading TA to understand. You are strongly encouraged to make this README as readable as possible by using subheadings and outlines. Do NOT simply translate your program into a paragraph of English.
- Implementation Challenges - Describe the parts of code that you found most troublesome and explain why. Reflect on how you overcame those challenges and what helped you finally understand the concept that was giving you trouble.

- Testing - Describe the tests you performed on your code to evaluate functionality and robustness. Enumerate possible edge cases and explain how you tested for them. Additionally, talk about general testing strategies you used to test systems code.
- Remaining Bugs - Point out and explain as best you can any bugs that remain in the code.

6.9.2 Submission

When you are ready to submit Part 1, you should add `Report_1.pdf` to the `lab3` folder, then do the following:

```
git commit -a -m 'Lab 3 Part 1 submission'
git push
```

When you are ready to submit Part 2, you should add `Report_2.pdf` to the `lab3` folder, then do the following:

```
git commit -a -m 'Lab 3 Part 2 submission'
git push
```