CS 425: Computer Networking

IIT Kanpur

Project 4: Building your Own Internet Router

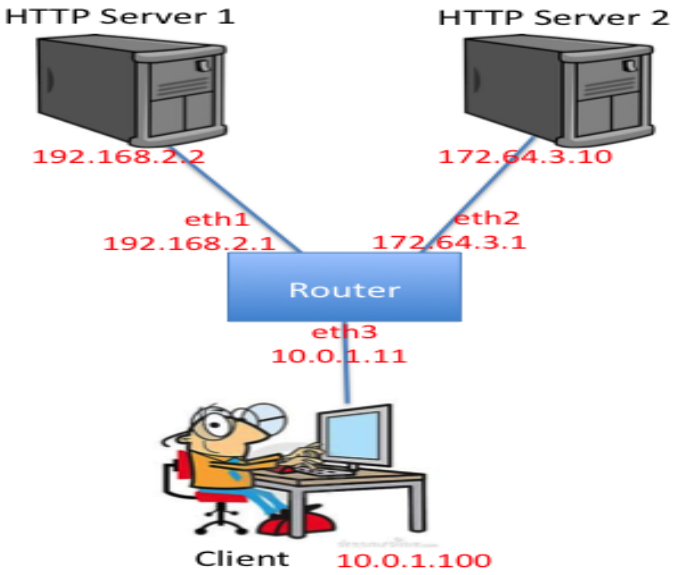
0. Due Date

This project is due on **October 30, 2016 at 11:55 pm via moodle**. The late submission policy and penalty discussed in the first class applies.

1. Introduction

In this project you will be writing a simple router with a static routing table. Your router will receive raw Ethernet frames. It will process the packets just like a real router, and forward them to the appropriate outgoing interface. We'll make sure you receive the Ethernet frames; your job is to create the forwarding logic so packets go to the correct interface.

Your router will route real packets from an emulated host (client) to two emulated application servers (http server 1/2) sitting behind your router. The application servers are each running an HTTP server. When you have finished the forwarding path of your router, you should be able to access these servers using regular client software. In addition, you should be able to ping and traceroute to and through your functioning Internet router. A sample routing topology is shown below:



If the router is functioning correctly, all of the following operations should work:

- Pinging from the client to any of the router's interfaces (192.168.2.1, 172.64.3.1, 10.0.1.1).
- Tracerouting from the client to any of the router's interfaces
- Pinging from the client to any of the app servers (192.168.2.2, 172.64.3.10)
- Tracerouting from the client to any of the app servers
- Downloading a file using HTTP from one of the app servers

Additional requirements are laid out in the 'Required Functionality' section.

2. Network Emulation using Mininet

This Project runs on top of Mininet which was built at Stanford. Mininet allows you to emulate a topology on a single machine. It provides the needed isolation between the emulated nodes so that your router node can process and forward real Ethernet frames between the hosts like a real router. You don't have to know how Mininet works to complete this assignment, but more information about Mininet (if you're curious) is available here (http://mininet.github.com/walkthrough/)

Environment Setup

Please download the VM from URL communicated by your TA's. You should have 7z installed to extract the downloaded file. On booting the VM, you will find a folder project4 which contains all files referred in this project. Follow steps below to setup and verify if everything is working as expected.

```
cd ~/project4/
sudo ./config.sh
```

Configuration Files

There are two configuration files.

- ~/project4/IP_CONFIG: Listed out the IP addresses assigned to the emulated hosts.
- ~/project4/router/rtable (also linked to ~/project4/rtable): The static routing table used for the simple router.

The default *IP_CONFIG* and *rtable* should look like the following:

```
> cat ~/project4/IP_CONFIG
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
> cat ~/project4/rtable
10.0.1.100  10.0.1.100  255.255.255.255 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
```

3. Test Connectivity of Your Emulated Topology

Start Mininet emulation:

```
> cd ~/project4/
> export PYTHONPATH=$PYTHONPATH:/usr/lib/python2.7/site-
packages/:/usr/lib/python2.6/site-packages/:~/project4/pox_module/
> sudo ./run_mininet.sh
```

You should see some output like the following:

```
*** 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': '192.168.2.1',
'sw0-eth2': '172.64.3.1', 'client': '10.0.1.100', 'server2': '172.64.3.10'}
*** 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
c0
*** 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 host client
client 10.0.1.1
*** Starting SimpleHTTPServer on host server1
*** Starting SimpleHTTPServer on host server2
*** Starting CLI:
mininet>
```

Keep this terminal open, as you will need the Mininet command-line for debugging. (Do not do ctrl-z)

Start POX controller:

Mininet requires a controller, which we implemented in POX. *In a second terminal*, run the following command:

```
> cd ~/project4/
> sudo ./run_pox.sh
```

You should be able to see some output like the following:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:.home.cse425.project4.pox_module.CSE425.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', 'server2': '172.64.3.10'}

INFO:.home.cse425.project4.pox_module.CSE425.srhandler:created server
DEBUG:.home.cse425.project4.pox_module.CSE425.srhandler:SRServerListener
listening on 8888
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.12/Jul 1 2016 15:12:24)
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
```

Please note that you have to wait until Mininet is able to connect to the POX controller before you continue to the next step.

Once Mininet is connected, you will see the following output in the POX terminal:

```
INFO:openflow.of_01:[Con 1/113051457621070] Connected to 66-d1-d7-f9-0c-4e
DEBUG:.home.cse425.project4.pox_module.CSE425.ofhandler:Connection [Con
1/113051457621070]
DEBUG:.home.cse425.project4.pox_module.CSE425.srhandler:SRServerListener catch
RouterInfo even, info={'eth3': ('10.0.1.1', '7a:76:bb:6c:13:a4', '10Gbps', 3),
'eth2': ('172.64.3.1', '66:41:62:c1:87:cb', '10Gbps', 2), 'eth1':
('192.168.2.1', 'd2:5f:01:b1:77:f7', '10Gbps', 1)}, rtable=[('10.0.1.100',
'10.0.1.100', '255.255.255.255', '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')]
```

```
Ready.
POX>
```
Keep POX running. (Do not do ctrl-z)

Start reference solution:

Now you are ready to test out the connectivity of the environment setup. To do so, *open a third terminal* and run the reference solution binary file “sr_solution”:
```
> cd ~/project4/
> ./sr_solution
```
You should be able to see some output like the following:
```
Loading routing table from server, clear local routing table.
Loading routing table
-----------------------------------------
Destination    Gateway        Mask    Iface
10.0.1.100             10.0.1.100     255.255.255.255 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 cse425 connecting to Server localhost:8888
Requesting topology 0
successfully authenticated as cse425
Loading routing table from server, clear local routing table.
Loading routing table
-----------------------------------------
Destination    Gateway        Mask    Iface
10.0.1.100             10.0.1.100     255.255.255.255 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    HWaddr7a:76:bb:6c:13:a4
        inet addr 10.0.1.1
eth2    HWaddr66:41:62:c1:87:cb
        inet addr 172.64.3.1
eth1    HWaddrd2:5f:01:b1:77:f7
        inet addr 192.168.2.1
 <-- Ready to process packets -->
```

In this particular setup, 192.168.2.2 is the IP for server1, and 172.64.3.10 is the IP for server2. You can find the IP addresses in your IP_CONFIG file.

Testing router functionality

To test the router functionality, you issue commands in the Mininet console. To run a command on an emulated host, type the host name followed by the command. Three examples follow.

For example, the following command issues 3 pings from the client to the server1.

```
mininet> client ping -c 3 192.168.2.2
```
You should be able to see the following output.
```
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
```

```
64 bytes from 192.168.2.2: icmp_seq=1 ttl=63 time=245 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=63 time=64.1 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=63 time=82.9 ms
```

You can also use traceroute to see the route between client to server1.

```
mininet> client traceroute -n 192.168.2.2
```

You should be able to see the following output.

```
traceroute to 192.168.2.2 (192.168.2.2), 30 hops max, 60 byte packets
 1  10.0.1.1  19.113 ms  58.006 ms  58.019 ms
 2  192.168.2.2  109.997 ms  110.002 ms  110.005 ms
```

Finally, to test the web server is properly working at the server1 and server2, issue an HTTP request by using *wget* or *curl*.

```
mininet> client wget http://192.168.2.2
```

You should see the following output.

```
--2016-09-26 19:28:51--  http://192.168.2.2/
Connecting to 192.168.2.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 161 [text/html]
Saving to: ‘index.html.1’

index.html.1        100%[===================>]     161  --.-KB/s    in 0s

2016-09-26 19:28:51 (52.7 MB/s) - ‘index.html.1’ saved [161/161]
```

Finally, to close Mininet, simple type "exit" in the Mininet console, and to close POX type ctrl-D.

4. Starter Code

In this project, you will replicate the functionality of sr_solution. To help you get started, we provide some starter code.

After following the steps in the 'Environment Setup' section, you should have all the pieces needed to build and run the router.

The starter source code

You can build and run the starter code as follows:

```
> cd ~/project4/router/
> make
>./sr
```

The starter code initializes the routing table given a file and it gets the router ready to receive packets.

Logging Packets

You can log the packets received and generated by your SR program by using the "-l" parameter with your SR program. The file will be in pcap format, i.e., you can use wireshark or tcpdump to read it.

```
> ./sr -l logname.pcap
```

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

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

5. General Forwarding Logic

To get you started, an outline of the forwarding logic for a router follows, although it does not contain all the details. There are two main parts to this assignment: Handling ARP and IP forwarding

5. 1 IP Forwarding

Given a raw Ethernet frame, if the frame contains an IP packet that is not destined towards one of our interfaces:

- Sanity-check the packet (meets minimum length and has correct checksum). You may simply drop the packet if either of these checks fails.
- Decrement the TTL by 1, and recompute the packet checksum over the modified header.
- Find out which entry in the routing table has the longest prefix match with the destination IP address.
- 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.

Obviously, this is a very simplified version of the forwarding process, and the low-level details follow in the 'Protocols to Understand' section. For example, if an error occurs in any of the above steps, you will have to send the appropriate ICMP message back to the sender notifying them of the error.

5.2 Handling ARP

Given a raw Ethernet frame, if the frame contains an ARP packet, check:

- Is it a request for the router's MAC address? If so, send an ARP reply.

- Is it a reply to a request the router previously sent out? If so, cache the reply and forward the IP packets waiting on this reply.

Details on how to handle ARP packets are discussed in the 'Protocols to Understand' section.

6. Protocols to Understand

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.

Internet Protocol

Before operating on an IP packet, you should verify its checksum and make sure it meets the minimum length of 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 "Environment Setup" 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.

Internet Control Message Protocol

ICMP is a simple protocol that can send control information to a host. 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; ICMP type 8) to one of the router's interfaces. This message is only sent for echo requests to any of the router's IPs. An echo request sent elsewhere should be forwarded to the next hop address as usual.

- 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 if 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. The source address of an ICMP message can be the source address of any of the incoming interfaces, as specified in RFC 792. 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:

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 time out 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 a received 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.

7. 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.

8. Code Overview

Basic Functions

Your router receives a raw Ethernet frame and sends raw Ethernet frames when sending a reply to the sending host or forwarding the frame to the next hop. The basic functions to handle these functions are:

```
void 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.

```
int 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 an arbitrary packet of length, len, to the network out of the interface specified by iface.

You *should not* free the buffer given to you in `sr_handlepacket` (you'll see the comment "lent" or "borrowed" whenever you should not free a buffer). 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).

The project requires you to send an ARP request about once a second until a reply comes back or we have sent five requests. To do so, you must implement the function `sr_arpcache_sweepreqs` defined in *sr_arpcache.c*. This function is 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. You must add this functionality to the `handle_arpreqs` function defined in *sr_router.c*. Pseudocode for these operations is provided in `sr_arpcache.h`.

9. 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.

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 (member routing_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 how to use the ARP cache and request queue 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.

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 good reference to the packet formats you'll be dealing with:

- Ethernet
- IP
- ICMP

- ARP

For the actual specifications, there are also the RFC's for ARP (RFC826), IP (RFC791), and ICMP (RFC792).

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.

10. Length of Assignment

This project is considerably harder than the first 3 projects, so get started early. The reference solution is 520 lines of code.

Of course, your solution may need fewer or more lines of code, but this gives you a rough idea of the size of the assignment to a first approximation.

Reference Binary

To help you debug your topologies and understand the required behavior we provide a reference binary and you can find it at *~/project4/sr_solution* in your directory:

We have talked about how to use it in the "Test Connectivity of Your Emulated Topology" section, please refer to the instructions there.

11. Required Functionality

- The router must successfully route packets between the Internet and the application servers.

- The router must correctly handle ARP requests and replies.

- The router must correctly handle traceroutes through it (where it is not the end host) and to it (where it is the end host).

- The router must respond correctly to ICMP echo requests.
- The router must handle TCP/UDP packets sent to one of its interfaces. In this case the router should respond with an ICMP port unreachable.
- The router must maintain an ARP cache whose entries are invalidated after a timeout period (timeouts should be on the order of 15 seconds).
- The router must queue all packets waiting for outstanding ARP replies. If a host does not respond to 5 ARP requests, the queued packet is dropped and an ICMP host unreachable message is sent back to the source of the queued packet.
- The router must not needlessly drop packets (for example when waiting for an ARP reply)
- The router must enforce guarantees on timeouts--that is, if an ARP request is not responded to within a fixed period of time, the ICMP host unreachable message is generated even if no more packets arrive at the router. (Note: You can guarantee this by implementing the sr_arpcache_sweepreqs function in sr_arpcache.c correctly.)

Also, don't forget to fill out your readme!

12. Extra Credit

Supporting Outbound Two-way TCP Traffic

Note that the extra credit part for this project is "on your own" meaning that you can only ask clarification questions and not get help from the TAs. The extra credit is basically a simple firewall. If you implement the extra credit, it should be able to be turned on/off. So we can test your required functionality.

Note that simply allowing packets from the internal hosts (in this assignment, from the client to any of the servers) to go through the router is not enough to establish a working connection to an external service, because most (if not all) TCP/IP services entail two-way communications. Therefore, packets that belong to a flow initiated by an internal end-host that arrive to the external interface must be allowed through the secure router. To support this feature the router maintains a "flow table" that contains all the active (and allowed) flows that traverse it. In this context, a flow is defined as a 5-tuple <srcIP, dstIP, IPprotocol, src-port, dst-port>.

When the first "internal" packet arrives at the router, two entries are added to the flow table, one for each direction of communication. The entry for the external-to-internal flow can be generated by inverting the order of source and destination IP addresses and ports. When a packet arrives to the external interface, the router checks if it matches one of the entries in the flow table. If it does then the packet is not dropped and it is forwarded to the internal interface.

Entries remain on the flow table as long as packets that match these entries go through the firewall. To support this feature each entry has a time-to-live (TTL). Each time a packet matching the flow entry is received, the entry's TTL is set to X seconds. The router periodically scans the flow table and removes all entries whose TTL has expired. (Note: You should update the entries associated with both directions of a flow when a packet is received.)

The flow table can hold up to Y entries at each time. You can make this parameter Y a constant. If a new entry needs to be added when the flow table is full, first a scan is initiated to determine if one or more stale entries exist in the flow table. If all entries are valid, then an ICMP response is returned (Destination Unreachable - Port Unreachable) to the originator and a log entry is generated.

A proper firewall supports adding explicit rules to allow/disallow flows to traverse the firewall in a "rule table". You don't need to worry about adding exceptions or creating a "rule table" i.e., you reject all external to internal traffic by default unless it matches an entry in the flow table. You don't need to worry about responding properly to pings or traceroutes in SE mode. Also you don't need to worry about packet fragments i.e., if a packet fragment arrives with the transport level header missing then the fragment (as well all other subsequent fragments) are dropped.

Your log of dropped packets should have the following format:

<srcIP, dstIP, protocol, src-port, dst-port, drop-code>

The drop-code for 'flow not allowed' is 2. The drop-code for 'flow table full' is 3.

Here's an example:

<1.2.3.4, 5.6.7.8, UDP, 54321, 23, 2>
<4.3.2.1, 9.8.6.5, TCP, 12345, 80, 3>

13. Submission

You should submit your completed router as a tar file via moodle. In the README, describe design decisions that you made, and any trade-offs that you encountered in the design.