CSCI 551 Lab1 Report

In this lab #1, we first imported the virtual machine and then created a host only Ethernet adapter. We assigned this adapter to the VM. We also configured a DHCP server so that the VM gets assigned an IP dynamically from our DHCP server in the IP address range 192.168.56.10 to 192.168.56.20. We used 2 important pieces of software namely: mininet and pox. Mininet provides network emulation within a single router and pox helps being a communication bridge between our C code and the emulated network in the router. We also used screen to multiplex between different sessions of programs: our router solution, mininet console, pox console etc. The mininet topology can be accessed via the mininet console and is also connected to the VM through interface eth0 which enables us to use diagnostic network commands such as ping and traceroute to our topology components (server1, server2, client) right from the VM.

There are 3 interfaces on our router namely eth1, eth2 and eth3 with assigned IP addresses: 192.168.2.1, 172.64.3.1 and 10.0.1.1 respectively; to each of these interfaces a host is connected. HTTP Server 1 is connected to eth1, HTTP Server 2 is connected to eth2 and a client is connected to eth3. Using my C code which plugs into the mininet router via pox has the prime functionality of forwarding packets to the correct interface based on the header fields. With our functionality we will be able to use diagnostic commands such as ping and traceroute to all of the three router interfaces as well as to the hosts (2 HTTP servers and 1 client). In addition, using either curl or wget we would be able to download files off the HTTP servers.

There is a test script provided which tests all the functionality mentioned above in basic as well as an extended topology. The router also makes use of a routing table (rtable file) which has the fields: prefix, next_hop, netmask and interface. There is a makefile provided in the lab1/router/ directory which makes a single executable called “sr” (static router) from my C code and then plugging it into the static router via pox.

We know that whenever an ethernet frame arrives at 1 of the router’s interface, within it is encapsulated an IP packet. The router then checks if the destination IP address in the packet is the router itself, if yes then it consumes it and does processing i.e., sends an ICMP echo reply if the packet was an ICMP packet with echo request; if the packet is not destined for 1 of the router interfaces, then the router consults the routing table (rtable) and performs the following logic:

```
For each table entry do
        If (destAddr & Subnet mask) == Subnet Number
                If NextHop is an interface
                        Deliver packet directly
                Else
Deliver packet to destination via the next hop (and use ARP/cache to get MAC address of next hop)
                EndIf
        EndIf
EndFor
```

We know that IP protocol is a network (L3) layer protocol and is used for host-to-host delivery i.e., delivery from source IP to destination IP whereas Ethernet is a Data Link (L2) layer protocol and is used for per hop delivery to deliver frames based on source and destination MAC address which keeps on changing each hop.

Let's say we want to send an IP packet from client 1 to server 1 via our router then the source IP would be 10.0.1.100 and destination IP would be 192.168.2.2. After the ethernet frame arrives at the router, the router extracts the IP packet from it, checks if it is large enough to have an IP header and validates the checksum. After that my router logic decrements TTL, recomputes the checksum with new header (with changed TTL value) and sees the destination as 192.168.2.2 (which is not one of its interfaces). It then consults the routing table with the above logic and finds the destination IP 192.168.2.2 ANDed with subnet mask 255.255.255.255 gives prefix 192.168.2.2 which matches the entry in the 2nd row and therefore the interface onto which the packet has to be forwarded is the last column i.e., eth1 (192.168.2.1). In order to construct an ethernet frame which contains this IP packet it checks whether it has entry for server 1's MAC address in its ARP cache, if yes then it uses that as the destination MAC address and source MAC address as eth1, if not then it sends an ARP request to server1 which responds with its MAC address and my router logic uses it to construct the frame as well as cache's it for later use. So essentially there are 2 hops from client 1 to router and then from router to server 1 each having its own frame but the IP packet contained within it is the same. On each hop the TTL field (Time to Live) is also decremented by 1. Internet Control Message Protocol is also used to send diagnostic information (Network unreachable, host unreachable) and also send echo reply packets to echo request packets enabling the ping command.

The primary logic for processing an IP packet is written within the function `process_ip_packet()`. It first checks whether the length of the packet is large enough to hold an IP header+Ethernet header. Then it extracts a pointer to IP header which is present after the ethernet header and is of type `sr_ip_hdr_t`. It then also extracts a pointer to ICMP header which will be present after ethernet header and then after IP header since ICMP packets are encapsulated within IP Packets which is then encapsulated into an Ethernet frame. It saves the original checksum in a variable and then computes the checksum using `cksum()` function and checks the currently computed checksum to be equal to the one sent in the header. It then gets the interface instance (`sr_if`) corresponding to the destination IP address, it does this by iterating through the interface list `sr->if_list` and matches the interface IP address with destination IP. In the IP header there is a field which indicates what are the contents inside it and the protocol as well. Based on that check I determine whether it is an IP packet or ICMP packet.

If it is an IP packet then I reduce TTL (if it becomes 0, I send an ICMP type 11 to indicate TTL expiry). Then I calculate longest prefix match. If no match is found I send an ICMP type 3 indicating no match or else I compute the checksum again with changed header (decremented TTL). Then I lookup ARP cache using method `sr_arpcache_lookup()` to check if an entry for next hop is present, if yes, I acquire the next MAC address and put it in the ethernet header and send out the packet. If no entry is present within ARP cache then I queue up an ARP request to get its MAC address using the method `sr_arpcache_queuereq()` and then call `process_arpreq()` to send an ARP request.

If it is an ICMP packet then I calculate its checksum to ensure its validity and check whether ICMP type is 8 (echo request). If it is, then I send an echo reply packet using the function `send_fabricated_icmp_packet()`. In this function I dynamically malloc a new memory space to construct

the ICMP packet. I construct the various headers required to send an ICMP packet beginning with Ethernet followed by IP followed by ICMP and send it on the same interface I got the packet on. I also put in appropriate values for all fields in these 3 headers: source IP, dest IP, TTL, ICMP protocol, ICMP type, source MAC, dest MAC and then send it off using `sr_send_packet()`. After that I use `free()` to free the memory which was occupied by the ICMP packet.

The functionality to send an ARP request is present within the sr_arpcache.c file in the function `process_arpreq()`. This function checks the time difference between now and last ARP request sent and if it is more than 1 second then it sends another if 5 ARP request have not been sent, if 5 ARP requests have already been sent with no reply then it sends an ICMP type 3 to notify ARP request has timed out. To send the ICMP packet it uses the same function `send_fabricated_icmp_packet()`. If 5 ARP requests have not been sent before then it sends an ARP request after constructing it. In order to construct it dynamically allocates memory using `malloc()`. Then it gets a pointer to the ARP header to populate the necessary fields in the ARP request namely: ARP operation, header length, target IP address (tip), sender IP address (sip), length of protocol address (pln), format of protocol address (pro), format of Hardware address (hrd) and sender hardware address (sha). It also then constructs the ethernet header and sends the ethernet frame containing the ARP request packet. In order to process the received ARP request, the method `process_arp_packet()` is used. In this method I first check if the ARP packet if of enough length to contain Ethernet header as well as ARP header. If yes, I extract the interface on which I received the ARP packet as well as pointers to ARP header and ethernet header. I then check the target IP requested within the ARP packet, if it is mine (router's), then I construct an ARP reply with `malloc()` and fill in the required fields in the ethernet and ARP header namely source MAC, dest MAC , ARP operation, ARP source MAC, ARP HW address format, ARP source and target IP and then I send the ARP reply packet off using `sr_send_packet()`. If an ARP reply is received instead of ARP request as per the ARP operation code then the particular ARP reply is extracted from the queue and inserted into the ARP cache using the metho d`sr_arpcache_insert()`.