Lab Report

Lab Name Respond to ICMP

Course	Computer Network
Major	Computer Science and Technology
Id	191220129
Name	Shangyu.Xing
Email	191220129@smail.nju.edu.cn
Date	2021.05

Contents

1 Objective

- Learn ICMP protocol and how to implement it;
- learn to implement hardware logic using the Switchyard framework;
- learn to capture network package using wireshark.

2 Requirements

This lab requires to implement a router who can respond to ICMP and generate ICMP error messages, in addition to forwarding packets and handling arp requests which was implemented in the last 2 labs. Specifically, the router should do the following:

- Respond to ICMP echo requests;
- generate ICMP error messages:
  - destination network unreachable
  - time exceeded
  - destination host unreachable
  - destination port unreachable

3 Procedure

I completed all the tasks as required. In this section, I will explain how I did my work in detail.

3.1 Modify forwarding logic

To complete the task, router's forwarding logic implemented in the last lab must be modified. The logic is below:

1. Check if the packet is arp request targeted at the router;
2. decrease ttl by 1 and generate time exceeded error if reduced to 0 (except that the packet is echo request targeted at the router itself);
3. if the packet is targeted at the router, check if it is echo request;
4. if it is, do echo reply; else generate destination port unreachable error;

5. if the packet is not targeted at the router, forward it.

```
def handle_packet(self, recv: switchyard.llnetbase.
    ReceivedPacket):
    timestamp, ifaceName, packet = recv
    # arp reply ...

    # forward & icmp handle
    hip_idx = packet.get_header_index(IPv4)
    if(hip_idx != -1):
        hip = packet[hip_idx]
        try:
            self.net.interface_by_ipaddr(hip.dst)
            # target self
            hicmp_idx = packet.get_header_index(ICMP)
            if hicmp_idx != -1 and packet[hicmp_idx].icmptype ==
                ICMPType.EchoRequest:
                # ping reply ...

            else:
                # check ttl ...

                # destination port unreachable ...

        except KeyError: # not self
            # check ttl ...

            self.forward_packet(packet, ifaceName)
```

During forwarding destination unreachable error can also be generated, so we should do some judgment.

```
def forward_packet(self, packet, ifaceName):
    hip = packet.get_header(IPv4)
    ip_intf = self.forwardTable.query(hip.dst)
    if ip_intf == None:
        # destination network unreachable ...

    next_ip, next_intf = tuple(ip_intf)
    intf = self.net.interface_by_name(next_intf)
    if intf.ipaddr != next_ip:
        # create Ethernet header and send out
        if next_ip == IPv4Address('0.0.0.0'):
            next_ip = hip.dst
        mac = self.arpTable.query(next_ip)
        if mac == None:
            # construct arp request ...

        else:
            # forward directly ...
```

Lastly, if an arp request is not replied after 5 retries, it should generate destination host unreachable error. This error occurs only when packets in arpQueue (waiting for arp reply) attempt to send arp requests again.

```
def send_arp_request(self, index):
        '''
        increase retry, remove if exceed, return new index
        '''
        entry = self.data[index]
        if entry[4] >= self.max_retry:
                # host unreachable ...

        # send request ...

        return index + 1
```

Now we have successfully constructed the framework for handling ICMP messages.

3.2 Implementation for ICMP messages

ICMP message packet consists of 3 headers: Ethernet, IPv4, ICMP. So I created a universal ICMP constructor:

```
def create_icmp(type, code, packet, srcip):
    # create ip
    hip = IPv4()
    hip.src = srcip
    hip.dst = packet.get_header(IPv4).src
    hip.ttl = 100
    hip.protocol = IPProtocol.ICMP
    # create icmp
    hicmp = ICMP()
    hicmp.icmptype = type
    hicmp.icmpcode = code
    return Ethernet() + hip + hicmp
```

Next, we should fill in the corresponding data field. We must pay special attention to srcip.
For destination unreachable, srcip should be that of the interface which received the packet.

```
# an example: host unreachable
hip = entry[0].get_header(IPv4)
if not (str(hip.src), str(hip.dst)) in self.unreach:
        self.unreach.add((str(hip.src), str(hip.dst)))
        reply = create_icmp(ICMPType.DestinationUnreachable, 1,
            entry[0], self.net.interface_by_name(entry[5]).ipaddr
            )
```

```
        rhicmp_idx = reply.get_header_index(ICMP)
        reply[rhicmp_idx].icmpdata.data = packet_data(entry[0])
        reply[rhicmp_idx].icmpdata.origdgramlen = len(entry[0])
        global router
        router.forward_packet(reply, "")
del self.data[index]
return index
```

Note that destination host unreachable is special, because it shouldn't send multiple error messages with the same srcip and dstip even if there are multiple error packets. So I used a set to store all (srcip, dstip) pairs. If it has already sent, just delete the packet from arpQueue without sending error message.
For ttl expire, srcip should also be that of the interface which received the packet.

```
reply = create_icmp(ICMPType.TimeExceeded, 0, packet, self.net.
    interface_by_name(ifaceName).ipaddr)
rhicmp_idx = reply.get_header_index(ICMP)
reply[rhicmp_idx].icmpdata.data = packet_data(packet)
reply[rhicmp_idx].icmpdata.origdgramlen = len(packet)
self.forward_packet(reply, "")
```

But for echo reply, srcip should also be the destination specified in the received packet.

```
reply = create_icmp(ICMPType.EchoReply, 0, packet, self.net.
    interface_by_name(ifaceName).ipaddr)
rhicmp_idx = reply.get_header_index(ICMP)
reply[rhicmp_idx].icmpdata.data = packet[hicmp_idx].icmpdata.
    data
reply[rhicmp_idx].icmpdata.identifier = packet[hicmp_idx].
    icmpdata.identifier
reply[rhicmp_idx].icmpdata.sequence = packet[hicmp_idx].icmpdata
    .sequence
self.forward_packet(reply, "")
```

4 Test & Result

4.1 Testcase

Firstly I tested my code with switchyard testcases:

```
Results for test scenario IP forwarding and ARP requester tests: 28 passed, 0 failed, 0 pending

Passed:
1   ICMP echo request (PING) for the router IP address
    192.168.1.1 should arrive on router-eth0.  This PING is
    directed at the router, and the router should respond with
    an ICMP echo reply.
2   Router should send an ARP request for 10.10.1.254 out
    router-eth1.
3   Router should receive ARP reply for 10.10.1.254 on router-
    eth1.
4   Router should send ICMP echo reply (PING) to 172.16.111.222
    out router-eth1 (that's right: ping reply goes out a
    different interface than the request).
5   ICMP echo request (PING) for the router IP address 10.10.0.1
    should arrive on router-eth1.
6   Router should send ICMP echo reply (PING) to 172.16.111.222
    out router-eth1.
7   ICMP echo request (PING) for 10.100.1.1 with a TTL of 1
    should arrive on router-eth1.  The router should decrement
    the TTL to 0 then see that the packet has "expired" and
    generate an ICMP time exceeded error.
```

Figure 1: Switchyard test result

4.2 Deployment

I designed my test in mininet in the following ways. Note that destination port unreachable error and destination host unreachable error are hard to generate in mininet with ping request, so they are omitted.

4.2.1 Echo reply

command: server1# ping -c 1 router
I ran wireshark on router's corresponding interface and got the following result (the wireshark capture result is saved in report/):

```
mininet> server1 ping -c 1 router
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
64 bytes from 192.168.100.2: icmp_seq=1 ttl=100 time=116 ms

--- 192.168.100.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 116.263/116.263/116.263/0.000 ms
mininet>
```

Figure 2: ping result

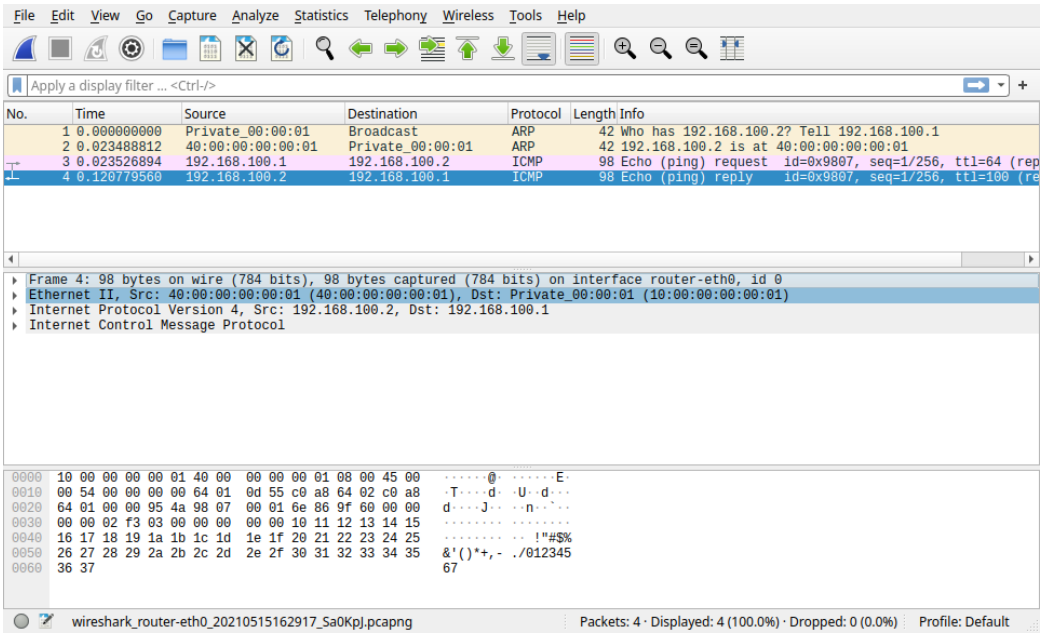

Figure 3: wireshark capture result

4.2.2 TTL expire

command: server1# ping -c 1 -t 1 router
result:

```
mininet> server1 ping -c 1 -t 1 router
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
From 192.168.100.2 icmp_seq=1 Time to live exceeded

--- 192.168.100.2 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

mininet>
```

Figure 4: ping result

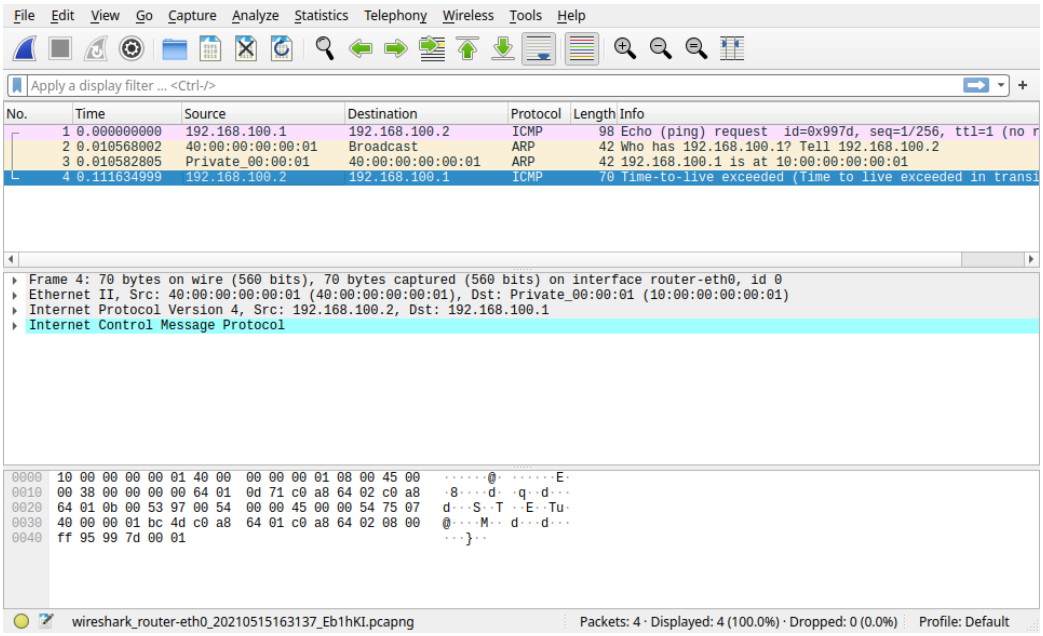

Figure 5: wireshark capture result

4.2.3 Destination network unreachable

In order to generate destination network unreachable error with a ping request from server1, I added an entry to start_mininet.py:

```
1 set_route(net, 'server1', '172.16.0.0/16', '192.168.100.2')
```

command: server1# ping -c 1 172.16.1.1
result:

```
mininet> server1 ping -c 1 172.16.1.1
PING 172.16.1.1 (172.16.1.1) 56(84) bytes of data.
From 192.168.100.2 icmp_seq=1 Destination Net Unreachable

--- 172.16.1.1 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

mininet>
```

Figure 6: ping result

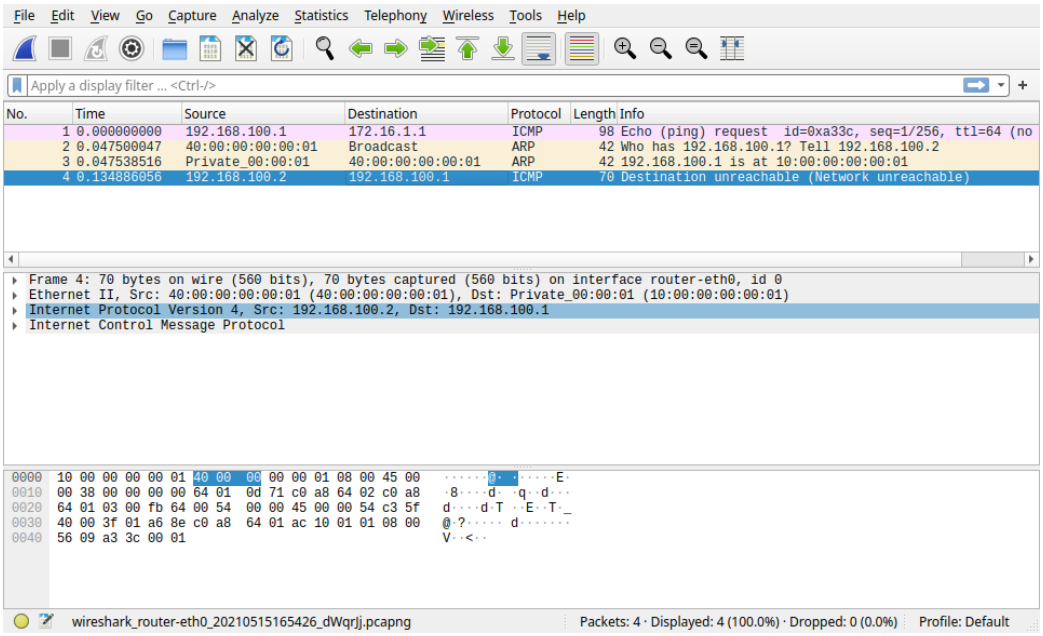

Figure 7: wireshark capture result

4.2.4 Traceroute

Lastly I tested my implementation with traceroute command:

```
mininet> server1 traceroute client
traceroute to 10.1.1.1 (10.1.1.1), 30 hops max, 60 byte packets
 1  192.168.100.2 (192.168.100.2)  97.005 ms  99.183 ms  101.456 ms
 2  10.1.1.1 (10.1.1.1)  304.790 ms  306.247 ms  307.383 ms
mininet>
```

Figure 8: traceroute test result

It turned out fine! Besides, it is worth mentioning that I didn't use -N option, which means my program can efficiently handle a large number of packets in a very short time.

5 Summary

- Knowing how to use tools effectively will greatly enhance working efficiency;
- always keep your code simple and clear, or you have to reconstruct it before you can add new functions;

- English reading and writing skills are important.