Build Your Own Router

田正祺　2020080095　软件 01

1 Environment

New Ubuntu 16.04.7 Desktop VM on Hyper-V. The only installed dependencies were from `setup.sh`. Various libraries related to I/O, data/string storage and manipulation from the STL were used. Please refer to the `includes` in each changed file.

2 Files changed

File	Changes
`arp-table.cpp`	Definition of `ArpCache::periodicCheckArpRequestsAndCacheEntries`
`routing-table.cpp`	Definition of `RoutingTable::lookup`
`simple-router.hpp`	Declaration of `SimpleRouter::send_or_queue`
`simple-router.cpp`	Definition of `SimpleRouter::send_or_queue` and `SimpleRouter::handlePacket`
`pdu.hpp`	Declaration and definition of various `PDU`s

3 Implementation

3.1 Object-oriented data manipulation

During this project, a difficulty I experienced was a significant portion of time was spent manipulating the data within packets/frames, and I would be writing the same code multiple times. As we are using C++, I took an object-oriented approach to manipulating the data. Within `pdu.hpp`, the classes `PDU`, `Ethernet`, `ARP`, `IP`, `ICMP` are defined. Each data unit of a layer inherits from the data unit that encapsulates it in the layer below. For example, `ICMP` inherits `IP`. Most member functions have a `protected` access specifier so its child classes can use the same functions. This increased readability, reduced code duplication, and allowed for easier debugging.

3.2 Routing table

Within the function `RoutingTable::lookup` in `routing-table.cpp`, the longest prefix match was implemented. The IP address with the longer prefix is also the one that is numerically greater, hence a copy of the routing table was sorted in decreasing order. Iterating the sorted table, if applying a routing table entry's mask to both the entry's destination IP address and the queried IP address creates identical results, then that entry's destination is the longest prefix match.

3.3 ARP table

Within the function `ArpCache::periodicCheckArpRequestsAndCacheEntries` in `arp-cache.cpp`, the ARP table was updated. For each ARP request, if it had been sent enough times without receiving a reply, then send an ICMP host unreachable message was sent to the source of each packet waiting on the ARP reply. Otherwise, resend the ARP request. Then, the cached ARP entries were iterated and the timed-out entries were deleted.

3.4 Handling received packets

The bulk of this project's work lies within the implementation of `SimpleRouter::handlePacket` in `simple_router`. Below is a summary of the flow control within the function:

```
return if ethernet header too short or MAC address doesn't match interface
if packet contains ARP packet
    return if packet too short
    if packet is ARP request
        return if destination IP address doesn't match
        send ARP reply
    else if packet is ARP reply
        cache ARP entry
        for each pending packet
            send packet
else if packet contains IP packet
    return if packet too short or IP checksum incorrect
    if packet destined to router
        if protocol is TCP or UDP
            send ICMP port unreachable
        else if protocol is ICMP
            return if packet too short or ICMP checksum incorrect
            if type is echo request
                send echo reply
    else
        decrement TTL
        if TTL is 0
            send ICMP time exceeded
        else
            find next hop IP address
            forward packet
```

One might note that "send packet" is missing a few steps. In most cases, it actually calls the function `send_or_queue`. It first checks if there exists an ARP entry for the destination IP address. If so, it fills in the destination MAC address and actually sends the packet. Otherwise, it queues the packet and sends an ARP request. As there are many instances of sending packets, this group of tasks warranted its own function.

4 Problems and Solutions

A major problem I faced was the difficulty of determining the contents of the packets both sent from and received by the router. For the majority of the debugging process I simply printed the packets using the functions in `utils.h`. However, due to the large number of packets being transmitted, this method was rather unwieldy. After realizing that Wireshark installed within the VM can capture traffic from the virtual interfaces, it greatly increased the ease of debugging.

Furthermore, a significant portion of the debugging also involved correcting mistakes in setting the data within the packet, whether it be accidentally switching the source and destination, or the verbosity of the data type conversions. However, as mentioned above, using an object-oriented approach greatly helped in this regard.