CptS/EE 455
Project #3

Instructor: Adam Hahn

Due: 10/30/2019 at 11:59 pm

Assignment: This project will implement a program that send IP datagrams, extending the code developed for Project 2. The program should be run and validated within a Mininet emulation using the following configuration:

- `router.py` (*required*) –This file creates the Mininet environment necessary to test your router. The network includes six hosts (h1x1, h1x2, h2x1, h2x2, h3x1, h3x1), each within their subnet. The router (r0) is connect through each device through a switch. The Mininet environment can be run through the command (`$ python router.py`).

Deliverable: Submit all code to blackboard by the due date. Ensure instructions to build and run the program are provided.

1 Overview

In this project you must develop a program that can correctly send and receive IP frames, while using previously developed ARP requests to determine the correct destination. The program must have two modes (Send and Recv). The Send mode should take an interface name, destination IP, router IP, and message; and then send that message in an IP datagram to the destination IP. The receive mode should take an interface name and print out IP packets received on that interface.

Sending:

```
./455_proj3 Send <InterfaceName> <DestIP> <RouterIP> <Message>
```

Receiving:

```
./455_proj3 Recv <InterfaceName>
```

Examples:

```
./455_proj3 Send h1x1-eth0 10.0.0.1 192.168.1.100 'This is a test'
./455_proj3 Recv h3x2-eth0
```

2 Requirements

Develop a program that can successfully send IP packets from any host to any other host in a different subnet, to do this, the packets must also use ARP to find the correct destination MAC address. The program must perform the following tasks:

a) You'll need to send ARP requests (which can be either the router, from `<RouterIP>`, or destination host) and processes responses to get the correct destination MAC address, ARP requests should be automatically

generated/processed before sending an response, The ARP messages should be interoperable with both local hosts (running this same program) and the router (running the Linux APR implementation).

b) To perform the correct ARP requests, you'll need to determine whether the `DestIP` resides within the same network as the sending host. To do this you'll need to properly analyze the host's IP and netmask (Recommendation **b**), along with the `DestIP`.

c) For IP messages, ensure you properly specify all IP header parameters with the proper byte order. Choose reasonable values for the `TTL` and `Ident` fields. Ensure the `checksum` value is correctly computed (example below).

d) During the creation of Ethernet segments, you'll need to specify the correct value for `type`, depending on whether its an ARP (0x806) or and IP (0x0800) message.

3 Recommendations

The following list includes some recommended techniques that will likely be useful in the development of this router, but are not strictly required.

a) You need to generate and parse IP datagrams. The <`netinet/ip.h`> library specifies the `struct iphdr` which can be cast to a received buffer and allows you to view specific elements. You'll also need to convert IP addresses from string to binary format. The `inet_aton`() function can perform this. Furthermore, the `struct in_addr` is useful for storing addresses, which stores in an unsigned long `s_addr`.

```
struct in_addr addr
inet_aton("192.168.1.0", &addr);
```

b) Use the following `ioctl` commands/options to get the IP address and netmask associated with an interface.

```
unsigned int get_netmask(char *if_name, int sockfd){
   struct ifreq if_idx;
   memset(&if_idx, 0, sizeof(struct ifreq));
   strncpy(if_idx.ifr_name, if_name, IFNAMSIZ-1);
   if((ioctl(sockfd, SIOCGIFNETMASK, &if_idx)) == -1)
       perror("ioctl():");
   return ((struct sockaddr_in *)&if_idx.ifr_netmask)->sin_addr.s_addr;
 }

unsigned int get_ip_saddr(char *if_name, int sockfd){
   struct ifreq if_idx;
   memset(&if_idx, 0, sizeof(struct ifreq));
   strncpy(if_idx.ifr_name, if_name, IFNAMSIZ-1);
   if (ioctl(sockfd, SIOCGIFADDR, &if_idx) < 0)
       perror("SIOCGIFADDR");
    return ((struct sockaddr_in *)&if_idx.ifr_addr)->sin_addr.s_addr;
 }
```

c) You'll need to compute a IP checksum. The following code can be used to compute this, where `vdata` is pointer to the start of a packet, and `length` is the length of the header

(20 bytes). Note: the checksum should be computed over a completed packet, with the checksum set to `0x0000`.

```
int16_t ip_checksum(void* vdata,size_t length) {
    char* data=(char*)vdata;
    uint32_t acc=0xffff;

    for (size_t i=0;i+1<length;i+=2) {
        uint16_t word;
        memcpy(&word,data+i,2);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }

    if (length&1) {
        uint16_t word=0;
        memcpy(&word,data+length-1,1);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }
    return htons(~acc);
}
```

4 Grading

Your project will be graded by the TA looking for correct functionality. We will test whether each host h1x1 can successfully send an IP datagram to two other hosts (h1x2 and h3x2). Specifically, this will ensure Wireshark doesn't not raise any error, on the Ethernet and IP parsing (upper layer errors are ok). Correct ARP requests must be performed source HW addresses are correctly determined.