CS 39006: Assignment 8

Implementation of Traceroute using Raw Sockets

Assignment Date: 28-Mar-2019

Deadline: 11-April-2019 2:00 PM

Objective:

The objective of this assignment is to get familiar with raw sockets. You will implement `mytraceroute` -- your version of the Linux `traceroute` tool for identifying the number of layer 3 (IP layer) hops from your machine to a given destination. For this, get familiar with the `traceroute` tool in Linux. A typical output of `traceroute` www.iitkgp.ac.in is as follows:

```
traceroute to www.iitkgp.ac.in (10.3.100.102), 64 hops max, 52 byte packets
 1  router.asus.com (192.168.1.1)  1.928 ms  1.215 ms  1.267 ms
 2  10.124.88.3 (10.124.88.3)  2.741 ms  3.941 ms  2.700 ms
 3  10.118.1.169 (10.118.1.169)  16.205 ms  2.380 ms  4.539 ms
 4  10.3.100.102 (10.3.100.102)  2.040 ms  2.635 ms  1.758 ms
```

So there are 4 hops between my machine and the www.iitkgp.ac.in web server. `traceroute` sends three test packets (UDP packets with payload size of 52 bytes) at every hop -- so the three timestamps that you observe at every row are the minimum response time, maximum response time and the average response time, respectively.

Problem Statement:

You have to implement your version of `traceroute` called `mytraceroute`. The user runs the program using the following format:

```
mytraceroute <destination domain name>
```

The program uses two raw sockets, S1 to send UDP packets, and S2 to receive ICMP packets. It then runs as follows.

1. The program first finds out the IP address corresponding to the given domain name.
2. Create the two raw sockets with appropriate parameters and bind them to local IP.
3. Set the socket option on S1 to include IPHDR_INCL.
4. Sends a UDP packet for traceroute using S1. In more detail:
    a. Set the variable TTL = 1. This variable corresponds to the TTL field of the IP header.
    b. Create a UDP datagram with a payload size of 52 bytes. You can generate the payload with random bytes. Note that you need to follow the exact format of UDP header and set the header fields correctly. In the UDP packet, set the destination port as 32164. This port is likely to be a closed port at your destination server (the domain name specified by you) -- so, if the IP layer of the destination server

receives this UDP datagram, it is likely to send a ICMP Destination Unreachable message; therefore you can be sure that you have reached the target server.

c. Append an IP header with the UDP datagram. Set the fields of the IP headers properly, including the TTL field. The destination IP is the IP of your target server (IP corresponding to the domain name you provided).
d. Send the packet through the raw socket S1.

5. Make a select call to wait for a ICMP message to be received using the raw socket S2 or a timeout value of 1 sec.
6. If the select() call comes out with receive of an ICMP message in S2:
    a. If it is a ICMP Destination Unreachable Message (you need to check the IP protocol field to identify a ICMP packet which has a protocol number 1, check https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml for details; then check the ICMP header to check the `type` field. If `type = 3` then it is a ICMP Destination Unreachable Message), then you have reached to the target destination, and received the response from there (to be safe, check that the source IP address of the ICMP Destination Unreachable Message matches with your target server IP address). Print nicely (see format below) and exit after closing the sockets.
    b. Else if it is an ICMP Time Exceeded Message (check the ICMP header to check the `type` field. If `type = 11` then it is an ICMP Time Exceeded Message), then you have got the response from the Layer 3 device at an intermediate hop specified by the TTL value. Extract the IP address (source IP address of the ICMP Time Exceeded Message) which is the IP address of that hop. Thus, you have identified the device IP at a hop specified by the TTL value. Print nicely (see format below) and then proceed to Step 8.
    c. Else if you have received any other ICMP packet, it is a spurious packet not related to you. Ignore and go back to wait on the select call (Step 5) with the REMAINING value of timeout.
    d. For each ICMP Time Exceeded or Destination Unreachable received, measure the response time (time difference between the UDP packet sent, measured just after the `sendto()` function, and the reception of the ICMP Time Exceeded message or Destination Unreachable, measured after the `recvfrom()` call). Print it in the following format:

        ```
        Hop_Count(TTL_Value)    IP address      Response_Time
        ```

        IP address above is the source IP address of the ICMP Time Exceeded Message or the ICMP Destination Unreachable Message.
7. If the select call times out, repeat from step 4 again. Total number of repeats with the same TTL is 3. If the timeout occurs for all the three UDP packets and you do not receive a ICMP Time Exceeded Message or ICMP Destination Unreachable message in any of them, then print it as follows:

    ```
    Hop_Count(TTL_Value)     *      *
    ```

Proceed to the next step if 3 repeats are over.

8. Increment the TTL value by 1 and continue from Step 4 with this new TTL value.

Submission Instruction:

Submit a single C source file named mytraceroute_<roll_number>.c and upload it at Moodle course page by the deadline (11 April 2019 2:00 PM).

Sample Code for Raw Sockets:

File Name: raw.c /*Raw Sockets to look into the IP packets - this is a server code that captures packets from IP layer */

```
#include<stdio.h>
#include<stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <linux/ip.h> /* for ipv4 header */
#include <linux/udp.h> /* for upd header */
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>

#define MSG_SIZE 2048
#define LISTEN_PORT 8080
// #define LISTEN_IP "127.0.0.1""
#define LISTEN_IP "0.0.0.0"

int main(){
    int rawfd, udpfd;
    struct sockaddr_in saddr_raw, saddr_udp;
    struct sockaddr_in raddr;
    int saddr_raw_len, saddr_udp_len;
    int raddr_len;

    char msg[MSG_SIZE];
    int msglen;

    pid_t pid = fork();
    if(pid == 0){
        struct iphdr hdrip;
        struct udphdr hdrudp;

        int iphdrlen = sizeof(hdrip);
        int udphdrlen = sizeof(hdrudp);
```

```
        rawfd = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
        if(rawfd < 0){
            perror("raw socket");
            exit(__LINE__);
        }

        saddr_raw.sin_family = AF_INET;
        saddr_raw.sin_port = htons(LISTEN_PORT);
        saddr_raw.sin_addr.s_addr = INADDR_ANY; //inet_addr(LISTEN_IP);
        saddr_raw_len = sizeof(saddr_raw);

        if(bind(rawfd, (struct sockaddr*) &saddr_raw, saddr_raw_len) < 0){
            perror("raw bind");
            exit(__LINE__);
        }

        while(1){
            raddr_len = sizeof(raddr);
            msglen = recvfrom(rawfd, msg, MSG_SIZE, 0, (struct sockaddr *)
&raddr, &raddr_len);
            if (msglen <= 0) //ignoring all the errors is not a good idea.
                continue;
            hdrip = *((struct iphdr *) msg);
            hdrudp = *((struct udphdr *) (msg + iphdrlen));
            if(hdrudp.dest != saddr_raw.sin_port)
                continue;

            msg[msglen] = 0;

            printf("RAW socket: ");
            printf("hl: %d, version: %d, ttl: %d, protocol: %d",
hdrip.ihl, hdrip.version, hdrip.ttl, hdrip.protocol);
            printf(", src: %s", inet_ntoa(*((struct in_addr *)
&hdrip.saddr)));
            printf(", dst: %s", inet_ntoa(*((struct in_addr *)
&hdrip.daddr)));
            printf("\nRAW socket: \tUdp sport: %d, dport: %d",
ntohs(hdrudp.source), ntohs(hdrudp.dest));
            printf("\nRAW socket: \tfrom: %s:%d",
inet_ntoa(raddr.sin_addr), ntohs(raddr.sin_port));
            printf("\nRaw Socket: \tUDP payload: %s",
msg+iphdrlen+udphdrlen);
            printf("\n");
        }
        close(rawfd);
    }

    else{
        udpfd = socket(AF_INET, SOCK_DGRAM, 0);
        if(udpfd < 0){
            perror("udp socket");
            exit(__LINE__);
        }
```

```
        saddr_udp.sin_family = AF_INET;
        saddr_udp.sin_port = htons(LISTEN_PORT);
        saddr_udp.sin_addr.s_addr = INADDR_ANY; //inet_addr(LISTEN_IP);
        saddr_udp_len = sizeof(saddr_udp);

        if(bind(udpfd, (struct sockaddr*) &saddr_udp, saddr_udp_len) < 0){
            perror("raw bind");
            exit(__LINE__);
        }

        while(1){
            raddr_len = sizeof(raddr);
            msglen = recvfrom(udpfd, msg, MSG_SIZE, 0, (struct sockaddr *)
&raddr, &raddr_len);
            msg[msglen] = 0;
            printf("UDP: recv len: %d, recvfrom: %s:%d\n", msglen,
inet_ntoa(raddr.sin_addr), ntohs(raddr.sin_port));
            printf("UDP: payload: %s\n", msg);
        }
        close(udpfd);
    }


    return 0;
}
```

File Name: udp_cli.c /* Client Application to Send UDP Packets – Run the above server and then this client code to send packets to the server */

```
#include<stdio.h>
#include<stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <linux/ip.h> /* for ipv4 header */
#include <linux/udp.h> /* for upd header */
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>

#define MSG_SIZE 2048
#define LISTEN_PORT 8080
#define LISTEN_IP "127.0.0.1"

int main(){
    int udpfd;
    struct sockaddr_in saddr_udp;
    int saddr_udp_len;

    char msg[MSG_SIZE] = "Test";
```

```
    int msglen;
    int sendlen;

    udpfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(udpfd < 0){
        perror("udp socket");
        exit(__LINE__);
    }

    saddr_udp.sin_family = AF_INET;
    saddr_udp.sin_port = htons(LISTEN_PORT);
    saddr_udp.sin_addr.s_addr = inet_addr(LISTEN_IP);
    saddr_udp_len = sizeof(saddr_udp);
    msglen = strlen(msg);
    sendlen = sendto(udpfd, msg, msglen, 0, (struct sockaddr *)
&saddr_udp, saddr_udp_len);
    if (sendlen < 0)
        perror("sendto");
    else{
        printf("Successfully send\n");
    }
    close(udpfd);
    return 0;
}
```