

```

/* -----
(c) Riduan M ABID, eMail: R.Abid@aui.ma
This code was implemented as a partial requirement for
a Ph.D degree in Computer Science at Auburn University
Supervisor: Dr. Biaz - eMail: biazsaa@auburn.edu,
Shelby Center for Engineering Technology- Auburn Unveristy - 2009,
----- */

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>
#include <linux/if.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <netinet/in.h>
#include <asm/types.h>
#include <sys/ioctl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>

#define NETWORK_STATIONS 255
#define MAP_COUNT 10
#define SHARED_MEMORY_SIZE sizeof(struct MPP_Proxying_Table)

#define BROADCAST_ADDR {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}
#define NULL_ADDR {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
#define NULL_IP {0x00, 0x00, 0x00, 0x00}
#define MONITOR_ADDR1 {0x00, 0x15, 0x58, 0xC3, 0xF3, 0x7B}
#define MONITOR_ADDR2 {0x00, 0x06, 0x5B, 0xB0, 0x6C, 0x35}

struct Association_Hash_Table_Entry {
    __u8 IP_addr[4];
    __u8 mac_addr[6];
    __u8 MAP_mac_addr[6];
    struct Association_Hash_Table_Entry *next;
};

struct Association_Hash_Table_Entry Hash_Table[NETWORK_STATIONS];

struct NAT_Entry {
    __u8 Src_IP[4];
    __u8 Dest_IP[4];
    __u16 Src_Port;
    __u16 Dest_Port;
    struct NAT_Entry *next;
};

struct NAT_Entry NAT_Table[NETWORK_STATIONS];

```

```

struct MAP_Proxy_Entry {
    __u8 MAP_mac_addr[6];
    __u8 next_hop[6];
    __u32 seq_no;
    __u8 hop_count;
    double metric;
};
struct MPP_Proxying_Table {
    struct MAP_Proxy_Entry MAP_Entries[MAP_COUNT];
    int visible_MAPs;
};
struct _802_11_s_frame_header {
    __u8 dest[6];
    __u8 src[6];
    __u8 type[2];
    __u8 eltID;
    __u8 dest_proxy[6];
    __u8 src_proxy[6];
    __u8 final_dest[6];
    __u8 originator[6];
};

struct Association_Hash_Table_Entry* get_Hash_Entry(struct
Association_Hash_Table_Entry *Hash_Table, __u8 *IP_addr);
int get_MAP_Entry(struct MPP_Proxying_Table table, __u8 *mac_addr);
struct NAT_Entry* get_NAT_Entry(struct NAT_Entry* NAT_Table, __u8 *IP_packet, __u16
Src_Port, __u16 Dest_Port);
struct NAT_Entry* get_Reverse_NAT_Entry(struct NAT_Entry* NAT_Table, __u8
*IP_packet, __u16 Src_Port, __u16 Dest_Port);
void print_MPP_Proxying_Table(struct MPP_Proxying_Table table);
void print_mac_addr(__u8 * mac);
void print_IP_addr(__u8 * IP);
int comp_mac_addr(__u8 * mac1, __u8 * mac2);
int comp_IP_addr(__u8 * IP1, __u8 * IP2);
void print_frame_hdr(__u8 * hdr);
struct _802_11_s_frame_header fill_header(__u8 * dest, __u8 * src, __u8 *
dest_proxy, __u8 * src_proxy, __u8 * final_dest, __u8 * originator, __u8 * buffer);
unsigned short in_cksum(unsigned short *, int);

__u8 LAN_ifr_mac_addr[6], adhoc_ifr_mac_addr[6], IP_addr[4], NULL_ip[4] = NULL_IP;
__u8 src_mac_addr[6], MAP_mac_addr[6], dest_mac[6], final_dest_mac[6], NULL_addr[6],
broadcast_addr[6] = BROADCAST_ADDR, originator[6], local_addr[4], client_ip[4],
monitor1[6] = MONITOR_ADDR1, monitor2[6] = MONITOR_ADDR2;

int main(int argc, char* argv[]) {
    __u8 buffer[4096], buffer2[4096], IP_packet[4096], TCP_pseudo[4096], eltID,
*shm;
    struct sockaddr dest, name;
    int shmid, dlen, recievedBytes, len;
    int LAN_sock, cnt = 0, adhoc_sock, open_sock, fd, sockfd, optval = 1, i,
frame_size = sizeof(struct _802_11_s_frame_header);

```

```

key_t key;
struct ifreq ifr, ifr2;
struct sockaddr_ll sll, sll0;
struct _802_11_s_frame_header frame_hdr;
struct MPP_Proxying_Table MPP_proxying_table;
struct MAP_Proxy_Entry MAP_Entry;
struct Association_Hash_Table_Entry *Hash_Entry;
struct NAT_Entry* NAT_entry;
struct sockaddr_in connection;
unsigned short check_sum, tcp_len, tcpLEN, Src_Port, Dest_Port;

struct iphdr *ip;
struct tcphdr *tcp;
struct udphdr* udp;

// Checking for Arguments,
if (argc < 4) {
    perror("\n Inussufficient Arguments ");
    perror("\n Usage: <executable> <LAN-ifr_Name> <Ad-Hoc-ifr_Name> <Local
Gateway IP>\n");
    exit(-1);
}
//
-----
// - Getting LAN-ifr MAC Address,
fd = socket(AF_INET, SOCK_DGRAM, 0);
ifr.ifr_addr.sa_family = AF_INET;
strncpy(ifr.ifr_name, argv[1], IFNAMSIZ-1);
ioctl(fd, SIOCGIFHWADDR, &ifr);
close(fd);
memcpy(LAN_ifr_mac_addr, ifr.ifr_hwaddr.sa_data, 6);
// Finding LAN-ifr Index,
if ((LAN_sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0 ) {
    perror("socket");
    exit(-1);
}
memset(&ifr, 0, sizeof(ifr));
strcpy(ifr.ifr_name, argv[1]);
if ( ioctl(LAN_sock, SIOCGIFINDEX, &ifr) < 0) {
    perror("ioctl (SIOCGIFINDEX) ");
    exit(-1);
}
// Bind "sll0" to the LAN ifr,
memset(&sll0, 0, sizeof(sll0));
sll0.sll_family = AF_PACKET;
sll0.sll_protocol = htons(ETH_P_ALL);
sll0.sll_ifindex = ifr.ifr_ifindex;
if (bind(LAN_sock, (struct sockaddr *)&sll0, sizeof(sll0)) != 0) {
    perror("Error Binding LAN_sock");
    exit(-1);
}
//
-----
// - Getting Ad-Hoc-ifr MAC Address,

```

```

fd = socket(AF_INET, SOCK_DGRAM, 0);
ifr2.ifr_addr.sa_family = AF_INET;
strncpy(ifr2.ifr_name, argv[2], IFNAMSIZ);
ioctl(fd, SIOCGIFHWADDR, &ifr2);
close(fd);
memcpy(adhoc_ifr_mac_addr, ifr2.ifr_hwaddr.sa_data, 6);
// Finding Ad-Hoc-ifr Index,
if ((adhoc_sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0 ) {
    perror("socket error");
    exit(-1);
}
memset(&ifr2, 0, sizeof(ifr2));
strncpy(ifr2.ifr_name, argv[2], IFNAMSIZ);
if ( ioctl(adhoc_sock, SIOCGIFINDEX, &ifr2) < 0 ) {
    perror("ioctl (SIOCGIFINDEX) ");
    exit(-1);
}
// Bind "sll" to the wireless ifr,
memset(&sll, 0, sizeof(sll));
sll.sll_family = AF_PACKET;
sll.sll_protocol = htons(ETH_P_ALL);
sll.sll_ifindex = ifr2.ifr_ifindex;
if (bind(adhoc_sock, (struct sockaddr *)&sll, sizeof(sll)) != 0) {
    perror("Error Binding LAN_sock");
    exit(-1);
}
// open_sock: Recieving both LAN and adhoc frames,
if ((open_sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0 ) {
    perror("socket error");
    exit(-1);
}

// Locating the MPP_Proxying_Table Shared Memory,
key = 12345;
if ((shmid = shmget(key, SHARED_MEMORY_SIZE, IPC_EXCL)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == NULL) {
    perror("shmat");
    exit(1);
}

// Initializing Association Hash Table,
for (i = 0; i < NETWORK_STATIONS; i++) {
    memcpy(Hash_Table[i].IP_addr, NULL_addr, 4);
    memcpy(Hash_Table[i].mac_addr, NULL_addr, 6);
    memcpy(Hash_Table[i].MAP_mac_addr, NULL_addr, 6);
    Hash_Table[i].next = NULL;
}
// Initializing the NAT Table,
for(i = 0; i < NETWORK_STATIONS; i++) {
    memcpy(NAT_Table[i].Src_IP, NULL_addr, 4);
    NAT_Table[i].next = NULL;
}

```

```

}

// Getting Local IP,
__u32 ip_addr = inet_addr(argv[3]);
memcpy(local_addr, &ip_addr, 4);

// IP RAW Socket,
if ((sockfd = socket(AF_INET, SOCK_RAW, htons(ETH_P_ALL))) == -1) {
    perror("socket");
    exit(EXIT_FAILURE);
}
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &optval, sizeof(int));

// ----- MAIN LOOP
-----
while(1) {
    recievedBytes = recvfrom(open_sock,buffer,4096,0,&dest,&dlen);

    if (recievedBytes > 2100) {
        printf("\n ==> Check! a LONG Packet --- ");
        continue;
    }

    memcpy(&eltID, buffer+12, 1);
    // Discarding Routing Frames,
    if (eltID == 13 || eltID == 5 || eltID == 77 || eltID == 11 || eltID ==
22 || eltID == 44 || eltID == 55)
        continue;
    memcpy(dest_mac, buffer, 6);
    memcpy(src_mac_addr, buffer+6, 6);
    if (comp_mac_addr(src_mac_addr, adhoc_ifr_mac_addr) ||
comp_mac_addr(src_mac_addr, LAN_ifr_mac_addr))
        continue; // Discard Local messages,
    // Checking Frame eltID,
    memcpy(&eltID, buffer+14, 1);
    if (eltID == 27) { // an Ad-hoc Frame,
        if(!comp_mac_addr(dest_mac, adhoc_ifr_mac_addr))
            continue;
        printf("\n - %d, Ad-hoc Packet from: ",
cnt++);print_mac_addr(src_mac_addr);
        // Getting Frame Header,
        memcpy(&frame_hdr, buffer, frame_size);

        // Getting IP_Packet,
        memcpy(IP_packet,buffer+frame_size, recievedBytes-frame_size);

        // ===== NAT Processing ===== //
        // 1.1 Get the IP Packet and Recording it in the NAT Table,
        memcpy(IP_addr, IP_packet+12, 4);

        // 1.2 Recording the Quadriplet (Src_IP, Src_Port, Dest_IP,
Dest_Port) in the NAT Table,
        // 1.2.1 Getting Src_Port and Dest_Port,

```

```

if (ip->protocol == 6) {
    memcpy(&Src_Port, IP_packet+20, 2);
    memcpy(&Dest_Port, IP_packet+22, 2);
} else if (ip->protocol == 17) {
    memcpy(&Src_Port, IP_packet+20, 2);
    memcpy(&Dest_Port, IP_packet+22, 2);
} else if (ip->protocol == 1)
    // Src_Port, Dest_Port are Irrelevant - Just get the
client_IP,

    memcpy(client_ip, IP_addr, 4);
else
    Src_Port = Dest_Port = 0;

// 1.2.2 Getting the corresponding NAT Table Entry,
NAT_entry = get_NAT_Entry(NAT_Table, IP_packet, Src_Port,
Dest_Port);
if (NAT_entry == NULL) { // No Entry, Create a new one,
    printf("\n ----- NO NAT Table Entry -----
");
    i = IP_packet[19]%255; // Hashing with Destination IP,
    if (comp_IP_addr(NAT_Table[i].Src_IP, NULL_ip)) { // Insert
in Link List Head,
        memcpy(NAT_Table[i].Src_IP, IP_packet+12, 4);
        memcpy(NAT_Table[i].Dest_IP, IP_packet+16, 4);
        NAT_Table[i].Src_Port = Src_Port;
        NAT_Table[i].Dest_Port = Dest_Port;
        NAT_Table[i].next = NULL;
    } else {
NAT_Entry));
        NAT_entry = (struct NAT_Entry*)malloc(sizeof(struct

        memcpy(NAT_entry->Src_IP, IP_packet+12, 4);
        memcpy(NAT_entry->Dest_IP, IP_packet+16, 4);
        NAT_entry->Src_Port = Src_Port;
        NAT_entry->Dest_Port = Dest_Port;
        NAT_entry->next = NULL;
        // Linking the entry to the appropriate chain,
        NAT_entry->next = NAT_Table[i].next;
        NAT_Table[i].next = NAT_entry;
    }
}
// 2. NAT: Replace the Src IP by IP of Local Station,
memcpy(IP_packet+12, local_addr, 4);

// 3. Update CheckSums,

ip = (struct iphdr *)IP_packet;
// 3.1 IP Checksum,
ip->check = 0;
check_sum = in_cksum((unsigned short *)ip, sizeof(struct iphdr));
ip->check = check_sum;

// 3.2 UDP Checksum,
if (ip->protocol == 17)
    IP_packet[26] = IP_packet[27] = 0; // Disabled,

```

```

// 3.3 TCP Checksum,
if (ip->protocol == 6) {
    tcp = (struct tcphdr *) (IP_packet+20);
    tcp_len = recievedBytes - frame_size - 20;
    /* Forming the Pseudo ----- */
    memcpy(TCP_pseudo, IP_packet+12, 8);
    TCP_pseudo[8] = 0;
    TCP_pseudo[9] = ip->protocol;
    tcpLEN = htons(tcp_len);
    memcpy(TCP_pseudo+10, &tcpLEN, 2);
    tcp->check = 0;
    memcpy(TCP_pseudo+12, IP_packet+20, tcp_len);
    check_sum = in_cksum((unsigned short *)TCP_pseudo, tcp_len +
12);

    tcp->check = check_sum;
}

// 4. Sending the Packet,
connection.sin_family = AF_INET;
memcpy(&connection.sin_addr.s_addr, IP_packet+16, 4);
i = sendto(sockfd, IP_packet, recievedBytes-frame_size, 0, (struct
sockaddr *)&connection, sizeof(struct sockaddr));
// ===== END of NAT Processing, =====

// Getting MAC addr of Originator,
memcpy(originator, frame_hdr.originator, 6);

// Cheking Corresponding Hash Entry - Recording the Mapping
between IP and MAC,
Hash_Entry = get_Hash_Entry(Hash_Table, IP_addr);
if (Hash_Entry == NULL) { // Insert a new Hash Entry,
    printf("\n ----- NO Hash Entry -----");
    i = IP_addr[3]%255;
    if (comp_IP_addr(Hash_Table[i].IP_addr, NULL_ip)) { //
Insert in Link List Head,
        memcpy(Hash_Table[i].IP_addr, IP_addr, 4);
        memcpy(Hash_Table[i].mac_addr, originator, 6);
        memcpy(Hash_Table[i].MAP_mac_addr,
frame_hdr.src_proxy, 6);
        Hash_Table[i].next = NULL;
    } else {
        // Allocating-filling the new entry,
        Hash_Entry = (struct
Association_Hash_Table_Entry*)malloc(sizeof(struct Association_Hash_Table_Entry));
        memcpy(Hash_Entry->IP_addr, IP_addr, 4);
        memcpy(Hash_Entry->mac_addr, originator, 6);
        memcpy(Hash_Entry->MAP_mac_addr, frame_hdr.src_proxy,
6);

        // Linking the entry to the appropriate chain,
        Hash_Entry->next = Hash_Table[i].next;
        Hash_Table[i].next = Hash_Entry;
    }
}

```

```

    }

    } else { // ===== Processing INTERNET Frames - 802.3
===== //
        //if (comp_mac_addr(src_mac_addr, monitor1) ||
comp_mac_addr(src_mac_addr, monitor2))
        // continue;
        if (comp_mac_addr(dest_mac, broadcast_addr) ||
comp_mac_addr(src_mac_addr, NULL_addr) || !comp_mac_addr(dest_mac,
LAN_ifr_mac_addr)) {
            continue; // Discard Broadcast Messages in the LAN,
        }
        printf("\n * %d - a LAN Frame from: ", cnt++);
print_mac_addr(src_mac_addr);
        // Getting IP_Packet,
        memcpy(&IP_packet, buffer+14, recievedBytes-14);

        // Reverse NATting,
        ip = (struct iphdr *)IP_packet;

        // Getting Src_Port and Dest_Port,
        if (ip->protocol == 6) {
            memcpy(&Src_Port, IP_packet+20, 2);
            memcpy(&Dest_Port, IP_packet+22, 2);

        } else if (ip->protocol == 17) {
            memcpy(&Src_Port, IP_packet+20, 2);
            memcpy(&Dest_Port, IP_packet+22, 2);
        } else
            Src_Port = Dest_Port = 0;

        // Getting the Corresponding NAT Table Entry,
        NAT_entry = get_Reverse_NAT_Entry(NAT_Table, IP_packet, Src_Port,
Dest_Port);

        if (NAT_entry == NULL) { // Irrelevant Frame - Do nothing;
            //printf("\n ----- NO Reverse NAT Entry - Irrelevant Frame
----- ");
        } else if (ip->protocol == 1) {
            // Replace with the Natted IP
            memcpy(IP_packet+16, client_ip, 4);
        } else
            memcpy(IP_packet+16, NAT_entry->Src_IP, 4);

        // Recomputing IP Checksum,
        ip->check = 0;
        check_sum = in_cksum((unsigned short *)ip, sizeof(struct iphdr));
        ip->check = check_sum;

        // Disabling UDP Checksum,
        if (ip->protocol == 17)
            IP_packet[26] = IP_packet[27] = 0;

        // TCP Checksum,

```

```

if (ip->protocol == 6) {
    tcp = (struct tcphdr *) (IP_packet+20);
    tcp_len = recievedBytes - 14 - 20;
    /* Forming the Pseudo ----- */
    memcpy(TCP_pseudo, IP_packet+12, 8);
    TCP_pseudo[8] = 0;
    TCP_pseudo[9] = ip->protocol;
    tcpLEN = htons(tcp_len);
    memcpy(TCP_pseudo+10, &tcpLEN, 2);
    tcp->check = 0;
    memcpy(TCP_pseudo+12, IP_packet+20, tcp_len);
    check_sum = in_cksum((unsigned short *)TCP_pseudo, tcp_len +
12);

    tcp->check = check_sum;
}

// ===== END of Reverse NATing =====

// Get the IP address of Destination,
memcpy(IP_addr, IP_packet+16, 4);

// Get its corresponding MAP and MAC addr from association hash
table

Hash_Entry = get_Hash_Entry(Hash_Table, IP_addr);
if (Hash_Entry == NULL)
    continue; // An irrelevant Frame,
memcpy(final_dest_mac, Hash_Entry->mac_addr, 6);
memcpy(MAP_mac_addr, Hash_Entry->MAP_mac_addr, 6);

// Getting the MPP_Forwarding_Table,
memcpy(&MPP_proxying_table, shm, SHARED_MEMORY_SIZE);
// print_MPP_Proxying_Table(MPP_proxying_table);
// Getting the MAP Entry from MPP_Forwarding_Table,
i = get_MAP_Entry(MPP_proxying_table, MAP_mac_addr);

MAP_Entry = MPP_proxying_table.MAP_Entries[i];

// Setting the 802.11.s Frame Header,
frame_hdr = fill_header(MAP_Entry.next_hop, adhoc_ifr_mac_addr,
MAP_mac_addr, adhoc_ifr_mac_addr, final_dest_mac, src_mac_addr, buffer);
memcpy(buffer2, &frame_hdr, frame_size);
/* Set the dest MAC addresses */
memcpy(sll.sll_addr, MAP_Entry.next_hop, 6);
// Setting IP_packet,
memcpy(buffer2+frame_size, IP_packet, recievedBytes-14);
// Ya HOU,
if ((i = sendto(adhoc_sock, buffer2, recievedBytes+frame_size-14,
0x00,
(struct sockaddr *)&sll, sizeof(sll))) < 0 ) {
    perror("sendto");
    exit(-1);
}
}
}

```

```

}
int get_MAP_Entry(struct MPP_Proxying_Table table, __u8 *mac_addr) {
    int i;
    for (i = 0; i < MAP_COUNT; i++)
        if (comp_mac_addr(table.MAP_Entries[i].MAP_mac_addr, mac_addr))
            return i;
    printf("\n ERROR: NO such entry for MAP in MPP_Forwarding_Table FOR: \n");
    print_mac_addr(mac_addr);
    return -1;
}

void print_frame_hdr(__u8 * hdr) {
    printf("\n ----- 802.11s Frame Header -----");
    printf("\n Dest: ");
    print_mac_addr(hdr);
    printf("\n Src: ");
    print_mac_addr(hdr+6);
    printf("\n Dest_Proxy: ");
    print_mac_addr(hdr+15);
    printf("\n Src_Proxy: ");
    print_mac_addr(hdr+21);
    printf("\n Final_Dest: ");
    print_mac_addr(hdr+27);
    printf("\n Originator: ");
    print_mac_addr(hdr+33);
}

void print_mac_addr(__u8 * mac) {
    int i;
    for (i = 0; i < 6; i++)
        printf("%2x:", mac[i]);
}

void print_IP_addr(__u8 * IP) {
    int i;
    for (i = 0; i < 4; i++)
        printf("%d:", IP[i]);
}

int comp_mac_addr(__u8 * mac1, __u8 * mac2) {
    int i;
    for (i = 0; i < 6; i++)
        if (mac1[i] != mac2[i])
            return 0;
    return 1;
}

int comp_IP_addr(__u8 * IP1, __u8 * IP2) {
    int i;
    for (i = 0; i < 4; i++)
        if (IP1[i] != IP2[i])
            return 0;
    return 1;
}

struct Association_Hash_Table_Entry* get_Hash_Entry(struct
Association_Hash_Table_Entry *Hash_Table, __u8 *IP_addr) {

```

```

struct Association_Hash_Table_Entry *walker;
int index;

index = IP_addr[3] % 255;
if (comp_IP_addr(Hash_Table[index].IP_addr, NULL_ip))
    return NULL;
else if (comp_IP_addr(Hash_Table[index].IP_addr, IP_addr))
    return &Hash_Table[index];
else { // Looking in the chain,
    walker = Hash_Table[index].next;
    while (walker != NULL) {
        if (comp_IP_addr(walker->IP_addr, IP_addr))
            return walker;
        walker = walker->next;
    }
}
return NULL;
}

struct _802_11_s_frame_header fill_header(__u8 * dest, __u8 * src, __u8 *
dest_proxy, __u8 * src_proxy, __u8 * final_dest, __u8 * originator, __u8 * buffer) {
    struct _802_11_s_frame_header hdr;

    memcpy(hdr.dest, dest, 6);
    memcpy(hdr.src, src, 6);
    memcpy(hdr.type, buffer+12, 2);
    hdr.eltid = 27;
    memcpy(hdr.dest_proxy, dest_proxy, 6);
    memcpy(hdr.src_proxy, src_proxy, 6);
    memcpy(hdr.final_dest, final_dest, 6);
    memcpy(hdr.originator, originator, 6);

    return hdr;
}

void print_MPP_Proxying_Table(struct MPP_Proxying_Table table) {
    int i;
    printf("\n ----- MPP Proxying Table ----- ");
    printf("\n - Visible MAPs: %d", table.visible_MAPs);
    for (i = 0; i < table.visible_MAPs; i++) {
        printf("\n ***** MAP: %d", i+1);
        printf("\n ----- MAC addr: ");
        print_mac_addr(table.MAP_Entries[i].MAP_mac_addr);
        printf("\n ----- Next hop: ");
        print_mac_addr(table.MAP_Entries[i].next_hop);
        printf("\n ----- Hop count: %d", table.MAP_Entries[i].hop_count);
        printf("\n ----- Metric: %.2f", table.MAP_Entries[i].metric+1);
    }
}

unsigned short in_cksum(unsigned short *addr, int len)
{
    register int sum = 0;
    u_short answer = 0;
    register u_short *w = addr;

```

```

register int nleft = len;
while (nleft > 1)
{
    sum += *w++;
    nleft -= 2;
}
if (nleft == 1)
{
    *(u_char *) (&answer) = *(u_char *) w;
    sum += answer;
}
sum = (sum >> 16) + (sum & 0xffff);
sum += (sum >> 16);
answer = ~sum;
return (answer);
}

struct NAT_Entry* get_NAT_Entry(struct NAT_Entry* NAT_Table, __u8 *IP_packet, __u16
Src_Port, __u16 Dest_Port) {
    struct NAT_Entry *walker;
    int index;

    index = IP_packet[19] % 255;
    if (comp_IP_addr(NAT_Table[index].Src_IP, NULL_ip))
        return NULL; // No Entry,
    else if (comp_IP_addr(NAT_Table[index].Src_IP, IP_packet+12) &&
comp_IP_addr(NAT_Table[index].Dest_IP, IP_packet+16) && NAT_Table[index].Src_Port ==
Src_Port && NAT_Table[index].Dest_Port == Dest_Port)
        return &NAT_Table[index];
    else { // Looking in the Chain,
        walker = NAT_Table[index].next;
        while (walker != NULL) {
            if (comp_IP_addr(walker->Src_IP, IP_packet+12) &&
comp_IP_addr(walker->Dest_IP, IP_packet+16) && walker->Src_Port == Src_Port &&
walker->Dest_Port == Dest_Port)
                return walker;
            walker = walker->next;
        }
    }
    return NULL;
}

struct NAT_Entry* get_Reverse_NAT_Entry(struct NAT_Entry* NAT_Table, __u8
*IP_packet, __u16 Src_Port, __u16 Dest_Port) {
    struct NAT_Entry *walker;
    int index;

    index = IP_packet[15] % 255;
    if (comp_IP_addr(NAT_Table[index].Src_IP, NULL_ip))
        return NULL; // No Entry,
    else if (comp_IP_addr(NAT_Table[index].Dest_IP, IP_packet+12) &&
NAT_Table[index].Src_Port == Dest_Port && NAT_Table[index].Dest_Port == Src_Port)
        return &NAT_Table[index];
    else { // Looking in the Chain,
        walker = NAT_Table[index].next;

```

```
        while (walker != NULL) {
            if (comp_IP_addr(walker->Dest_IP, IP_packet+12) && walker->Src_Port == Dest_Port && walker->Dest_Port == Src_Port)
                return walker;
            walker = walker->next;
        }
    }
    return NULL;
}
```