

```

/* -----
   (c) Riduan M ABID
   This code was implemented as a partial requirement for
   a Ph.D degree in Computer Science at Auburn University
   Supervisor: Dr. S. Biaz
   ----- */

#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 <netinet/in.h>
#include <asm/types.h>
#include <time.h>
#include <sys/ioctl.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MP_COUNT 10
#define MPP_COUNT 5
#define MAP_COUNT 5
#define KEY_MP_TABLE 131305
#define KEY_ETX 130508
#define ETX_SHARED_MEMORY_SIZE MP_COUNT * sizeof(struct ETX)
#define SHARED_MEMORY_SIZE sizeof(struct MP_Forwarding_Table)
#define BROADCAST_ADDR {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}
#define NULL_ADDR {0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

struct PREQ {
    __u8 eltID;
    __u8 hop_count;
    __u8 MPP_mac_addr[6];
    __u32 MPP_seq_no;
    double metric;
};

struct RREP {
    __u8 eltID;
    __u8 MPP_mac_addr[6];
    __u8 MAP_mac_addr[6];
    __u32 MAP_seq_no;
    __u8 hop_count;
    double metric;
};

struct Proxy_Entry {
    __u8 mac_addr[6];
    __u8 next_hop[6];
    __u32 seq_no;
    __u8 hop_count;
    double metric;
};

```

```

struct MP_Forwarding_Table {
    struct Proxy_Entry MPP_Entries[MPP_COUNT];
    struct Proxy_Entry MAP_Entries[MAP_COUNT];
    int visible_MPPs;
    int visible_MAPs;
};
struct ETX {
    __u8 mac_addr[6];
    double df, dr, etx;
};

__u8 local_mac_addr[6], src_mac_addr[6], broadcast_mac_addr[6] = BROADCAST_ADDR,
NULL_addr[6] = NULL_ADDR, MPP_mac_addr[6], MAP_mac_addr[6];

int get_MPP_Entry(struct MP_Forwarding_Table table, __u8 *MPP_mac_addr);
int get_MAP_Entry(struct MP_Forwarding_Table table, __u8 *MAP_mac_addr);
int get_MP_ETXi_entry(struct ETX *ETX, __u8 *mac_addr);
void print_MP_Forwarding_Table(struct MP_Forwarding_Table table);
void print_ETXi (struct ETX *ETXi);
void print_mac_addr(__u8 * mac);
void print_PREQ(struct PREQ PREQ);
int comp_mac_addr(__u8 * mac1, __u8 * mac2);

int main(int argc, char* argv[]) {
    __u8 buffer[4096], buffer2[4096], *shm_MP_Table, *shm_ETX, eltID;
    struct sockaddr dest, name;
    int shmid, shmid_ETX, dlen, cnt = 0, recievedBytes, len, currentMPP,
currentMAP;
    int raw_sock, raw_sock2, fd, i, PREQ_size = sizeof(struct PREQ), RREP_size =
sizeof(struct RREP);
    key_t key_MP_Table, key_ETX;
    struct PREQ PREQ;
    struct RREP RREP;
    double link_metric;

    struct ifreq ifr;
    struct sockaddr_ll sll;
    struct MP_Forwarding_Table MP_forwarding_table;
    struct Proxy_Entry MPP_Entry, MAP_Entry;
    struct ETX ETXi[MP_COUNT];

    // Checking for Arguments,
    if (argc < 2) {
        perror("\n Inussuficient Arguments ");
        perror("\n Usage: <executable> <Interface_Name>");
        exit(-1);
    }
    // Getting Local 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(local_mac_addr, ifr.ifr_hwaddr.sa_data, 6);

```

```

// Finding Local Wireless Interface Index,
if ((raw_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(raw_sock, SIOCGIFINDEX, &ifr) < 0) {
    perror("ioctl (SIOCGIFINDEX) ");
    exit(-1);
}
if ((raw_sock2 = 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(raw_sock2, SIOCGIFINDEX, &ifr) < 0) {
    perror("ioctl (SIOCGIFINDEX) ");
    exit(-1);
}

// Bind "sll" to the interface,
memset(&sll, 0, sizeof(sll));
sll.sll_family = AF_PACKET;
sll.sll_protocol = ETH_P_ALL;
sll.sll_ifindex = ifr.ifr_ifindex;
if (bind(raw_sock2, (struct sockaddr *)&sll, sizeof(sll)) != 0) {
    perror("Error Binding Raw_Sock");
    exit(-1);
}

// Creating MP_Forwarding_Table Shared Memory,
key_MP_Table = KEY_MP_TABLE;

if ((shmid = shmget(key_MP_Table, SHARED_MEMORY_SIZE, IPC_CREAT)) < 0) {
    perror("shmget");
    exit(1);
}

if ((shm_MP_Table = shmat(shmid, NULL, 0)) == NULL) {
    perror("shmat");
    exit(1);
}

// Locating Shared Memory for ETXi[] entries,
key_ETX = KEY_ETX;
if ((shmid_ETX = shmget(key_ETX, ETX_SHARED_MEMORY_SIZE, IPC_CREAT)) < 0) {
    perror("shmget");
    exit(1);
}

// Reading ETXi[] entries,

```

```

if ((shm_ETX = shmat(shmid_ETX, NULL, 0)) == NULL) {
    perror("shmat");
    exit(1);
}
memcpy(ETXi, shm_ETX, ETX_SHARED_MEMORY_SIZE);
// Feedback,
print_ETXi(ETXi);

// Initializing the MP_Forwarding_Table,
for (i = 0; i < MPP_COUNT; i++) {
    memcpy(MP_forwarding_table.MPP_Entries[i].mac_addr, NULL_addr, 6);
    memcpy(MP_forwarding_table.MPP_Entries[i].next_hop, NULL_addr, 6);
    MP_forwarding_table.MPP_Entries[i].seq_no = 0;
    MP_forwarding_table.MPP_Entries[i].hop_count = 0;
    MP_forwarding_table.MPP_Entries[i].metric = 0;
}
MP_forwarding_table.visible_MPPs = 0;
for (i = 0; i < MAP_COUNT; i++) {
    memcpy(MP_forwarding_table.MAP_Entries[i].mac_addr, NULL_addr, 6);
    memcpy(MP_forwarding_table.MAP_Entries[i].next_hop, NULL_addr, 6);
    MP_forwarding_table.MAP_Entries[i].seq_no = 0;
    MP_forwarding_table.MAP_Entries[i].hop_count = 0;
    MP_forwarding_table.MAP_Entries[i].metric = 0;
}
MP_forwarding_table.visible_MAPs = 0;

while(1) {
    recievedBytes = recvfrom(raw_sock,buffer,4096,0,&dest,&dlen);
    memcpy(&eltID, buffer+12, 1);
    // Checking if this is a Routing Frame: a PREQ_BROADCAST OR a RREP
frame,
    if (eltID != 13 && eltID != 5)
        continue;
    memcpy(src_mac_addr, buffer+6, 6);
    if (comp_mac_addr(src_mac_addr, local_mac_addr))
        continue; // Avoiding possible local loops,

    // GET LINK METRIC,
    // .....
    // 1. Get ETXi,
    if ((shm_ETX = shmat(shmid_ETX, NULL, 0)) == NULL) {
        perror("shmat");
        exit(1);
    }
    memcpy(ETXi, shm_ETX, ETX_SHARED_MEMORY_SIZE);
    // 2. Get ETXi Entry for the Link,
    i = get_MP_ETXi_entry(ETXi, src_mac_addr);
    if (i == -1) // Entry not found,
        link_metric = 1000; // Infinity,
    else
        link_metric = ETXi[i].etx;

    // Checking Routing Frame Type,
    if (eltID == 13) { // ----- a PREQ_Broadcast Frame - coming from

```

```

an MPP,
cnt++);

printf("\n ....%d..... PREQ Broadcast --- Received FROM: ----",

print_mac_addr(src_mac_addr);
// Getting the PREQ,
memcpy(&PREQ, buffer+12, PREQ_size);
// Getting the MPP sending the PREQ,
memcpy(MPP_mac_addr, PREQ.MPP_mac_addr, 6);

// Getting the corresponding entry in the MP_Forwarding_Table,
currentMPP = get_MPP_Entry(MP_forwarding_table, MPP_mac_addr);
if (currentMPP == -1) { // There is NO entry, create one
    currentMPP = MP_forwarding_table.visible_MPPs++;
    memcpy(MP_forwarding_table.MPP_Entries[currentMPP].mac_addr,
MPP_mac_addr, 6);
    memcpy(MP_forwarding_table.MPP_Entries[currentMPP].next_hop,
src_mac_addr, 6);
    MP_forwarding_table.MPP_Entries[currentMPP].seq_no = 0;
    MP_forwarding_table.MPP_Entries[currentMPP].metric = 0;
}
MPP_Entry = MP_forwarding_table.MPP_Entries[currentMPP];

// Checking the freshness of the PREQ message,
if (PREQ.MPP_seq_no < MPP_Entry.seq_no)
    continue; // Discard - Outdated PREQ,
else if (PREQ.MPP_seq_no > MPP_Entry.seq_no) { // a fresher PREQ,
    // Update freshness,
    MPP_Entry.seq_no = PREQ.MPP_seq_no;
    // Updating Routing Entry,
    memcpy(MPP_Entry.next_hop, src_mac_addr, 6);
    MPP_Entry.hop_count = PREQ.hop_count+1;
    MPP_Entry.metric = PREQ.metric + link_metric;

    // Update PREQ,
    PREQ.hop_count++;
    PREQ.metric += link_metric;

    // Re-Broadcast,
    memcpy(buffer2, broadcast_mac_addr, 6);
    memcpy(buffer2+6, local_mac_addr, 6);
    memcpy(buffer2+12, &PREQ, PREQ_size);
    if ((i = sendto(raw_sock2, buffer2, 12+PREQ_size, 0x00,
        (struct sockaddr *)&sll, sizeof(sll))) < 0 ) {
        perror("sendto");
        exit(-1);
    }
} else { // Same PREQ freshness,
    // Update Routing Entry ONLY if this corresponds to a better
route,

    if (PREQ.metric + link_metric < MPP_Entry.metric) {
        // Update PREQ,
        PREQ.hop_count++;
        PREQ.metric += link_metric;
        // Updating Routing Entry,

```

```

MPP_Entry.metric = PREQ.metric;
MPP_Entry.hop_count = PREQ.hop_count+1;
memcpy(MPP_Entry.next_hop, src_mac_addr, 6);
// Re-Broadcast,
memcpy(buffer2, broadcast_mac_addr, 6);
memcpy(buffer2+6, local_mac_addr, 6);
memcpy(buffer2+12, &PREQ, PREQ_size);
if ((i = sendto(raw_sock2, buffer2, 12+PREQ_size,
0x00,
(struct sockaddr *)&sll, sizeof(sll))) < 0 )
{
    perror("sendto");
    exit(-1);
}
}
// Updating the Routing Entry,
MP_forwarding_table.MPP_Entries[currentMPP] = MPP_Entry;
} else if (eltID == 5) { // a RREP_Reply Frame, Coming from a MAP,
printf("\n - %d --- RREP Frame Processing from: ", cnt++);
print_mac_addr(src_mac_addr);
// Getting the RREP Frame,
memcpy(&RREP, buffer+12, RREP_size);

// Getting the MAP sending the RREP,
memcpy(MAP_mac_addr, RREP.MAP_mac_addr, 6);
// Getting the destination MPP,
memcpy(MPP_mac_addr, RREP.MPP_mac_addr, 6);

// Getting the corresponding entry in the MP_Forwarding_Table,
currentMAP = get_MAP_Entry(MP_forwarding_table, MAP_mac_addr);
if (currentMAP == -1) { // There is NO entry, create one
    currentMAP = MP_forwarding_table.visible_MAPs++;
    memcpy(MP_forwarding_table.MAP_Entries[currentMAP].mac_addr,
MAP_mac_addr, 6);
    memcpy(MP_forwarding_table.MAP_Entries[currentMAP].next_hop,
src_mac_addr, 6);
    MP_forwarding_table.MAP_Entries[currentMAP].seq_no = 0;
    MP_forwarding_table.MAP_Entries[currentMAP].metric = 0;
}
// Update the MAP entry,
memcpy(MP_forwarding_table.MAP_Entries[currentMAP].next_hop,
src_mac_addr, 6);
MP_forwarding_table.MAP_Entries[currentMAP].seq_no =
RREP.MAP_seq_no;
MP_forwarding_table.MAP_Entries[currentMAP].hop_count =
RREP.hop_count+1;
MP_forwarding_table.MAP_Entries[currentMAP].metric = RREP.metric +
link_metric;

// Update the RREP metric,
RREP.metric += link_metric;
RREP.hop_count++;

```

```

        // Forward the RREP,
        if (comp_mac_addr(local_mac_addr, src_mac_addr))
            continue; // Avoiding Loops,

        // Get next_hop to MPP from MP_Forwarding Table,
        i = get_MPP_Entry(MP_forwarding_table, MPP_mac_addr);
        MPP_Entry = MP_forwarding_table.MPP_Entries[i];

        // Avoiding Loops, Avoiding Sending back to the sender,
        if (comp_mac_addr(MPP_Entry.next_hop, src_mac_addr))
            continue;
        // Forwarding the RREP,
        memcpy(buffer2, MPP_Entry.next_hop, 6);
        memcpy(buffer2+6, local_mac_addr, 6);
        memcpy(buffer2+12, &RREP, RREP_size);
        memcpy(sll.sll_addr, MPP_Entry.next_hop, 6);
        if ((i = sendto(raw_sock2, buffer2, 12+RREP_size, 0x00,
            (struct sockaddr *)&sll, sizeof(sll))) < 0 ) {
            perror("sendto");
            exit(-1);
        }
        printf("\n + RREP Forwarded to ---> ");
print_mac_addr(MPP_Entry.next_hop);
    }
    // Allowing a 1 Second safety to write only one route, Avoiding Route
Fluctuations,
    // Display the MP_Forwarding_table,
    print_MP_Forwarding_Table(MP_forwarding_table);
    // Wrting the MP_Forwarding Table,
    memcpy(shm_MP_Table, &MP_forwarding_table, SHARED_MEMORY_SIZE);
}
}
int get_MPP_Entry(struct MP_Forwarding_Table table, __u8 *MPP_mac_addr) {
    int i;
    for (i = 0; i < table.visible_MPPs; i++)
        if (comp_mac_addr(table.MPP_Entries[i].mac_addr, MPP_mac_addr))
            return i;
    return -1;
}
int get_MAP_Entry(struct MP_Forwarding_Table table, __u8 *MAP_mac_addr) {
    int i;
    for (i = 0; i < table.visible_MAPs; i++)
        if (comp_mac_addr(table.MAP_Entries[i].mac_addr, MAP_mac_addr))
            return i;
    return -1;
}
void print_MP_Forwarding_Table(struct MP_Forwarding_Table table) {
    int i;

    printf("\n ----- MP Forwarding Table ----- ");
    printf("\n - Visible MPPs: %d", table.visible_MPPs);
    printf("\n - Visible MAPs: %d", table.visible_MAPs);
    printf("\n ----- MPP Entries ----- ");
    for (i = 0; i < table.visible_MPPs; i++) {

```

```

        printf("\n          -----");
        printf("\n          - MPP: %d ", i+1);
        printf("\n          - MAC addr: ");
print_mac_addr(table.MPP_Entries[i].mac_addr);
        printf("\n          - Next hop: ");
print_mac_addr(table.MPP_Entries[i].next_hop);
        printf("\n          - Hop count: %d", table.MPP_Entries[i].hop_count);
        printf("\n          - Metric: %.2f",table.MPP_Entries[i].metric);
    }
    printf("\n          ----- MAP Entries ----- ");
    for (i = 0; i < table.visible_MAPs; i++) {
        printf("\n          -----");
        printf("\n          - MAP: %d ", i+1);
        printf("\n          - MAC addr: ");
print_mac_addr(table.MAP_Entries[i].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);
    }
    printf("\n ----- ");
}
void print_mac_addr(__u8 * mac) {
    int i;
    for (i = 0; i < 6; i++)
        printf("%2x:", mac[i]);
}
void print_PREQ(struct PREQ PREQ) {
    printf("\n ----- Received PREQ: ----- \n");
    printf("\n eltID: %d", PREQ.eltID);
    printf("\n hop_count: %d", PREQ.hop_count);
    printf("\n MPP_mac_addr: "); print_mac_addr((__u8 *)PREQ.MPP_mac_addr);
    printf("\n orig_seq_no: %d", PREQ.MPP_seq_no);
    printf("\n metric: %.2f", PREQ.metric);
}
int comp_mac_addr(__u8 * mac1, __u8 * mac2) {
    int i;
    for (i = 0; i < 6; i++)
        if (mac1[i] != mac2[i])
            return 1;
    return 0;
}
void print_ETXi (struct ETX *ETXi) {
    int i;
    for (i = 0; i < MP_COUNT; i++) {
        printf("\n-----
\n* ");
        print_mac_addr(ETXi[i].mac_addr);
        printf(" - df: %.2f", ETXi[i].df);
        printf(" - dr: %.2f", ETXi[i].dr);
        printf(" - ETX: %.2f", ETXi[i].etx);
    }
}
}

```

```
int get_MP_ETXi_entry(struct ETX *ETXi, __u8 *mac_addr) {
    int i;
    for (i = 0; i < MP_COUNT; i++)
        if (comp_mac_addr(ETXi[i].mac_addr, mac_addr))
            return i;
    return -1;
}
```