

```

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

#define MPP_COUNT 5
#define MP_COUNT 10
#define KEY 777777
#define KEY_AIRTIME 131313
#define PREQ_SIZE sizeof(struct PREQ)
#define RREP_SIZE sizeof(struct RREP)
#define AirTIME_SHARED_MEMORY_SIZE MP_COUNT * sizeof(struct AirTIME)
#define SHARED_MEMORY_SIZE sizeof(struct MAP_Proxying_Table)
#define MPP_BROADCAST_PERIOD 2 // 2 SECS,
#define ROUTE_ENTRY_LIFETIME 2 * MPP_BROADCAST_PERIOD // To account for two
consectutive broadcast periods,
#define BROADCAST_ADDR {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}
#define NULL_ADDR {0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

// ----- MP_MAC_ADDRESSES
----- //
// ----- For Controlling Mesh Visibility, ----- //
#define MP_4F {0x00, 0x11, 0x50, 0x55, 0x4A, 0x4F}

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 MPP_Proxy_Entry {
    __u8 MPP_mac_addr[6];
    __u8 next_hop[6];
    __u32 seq_no;
    __u32 timeStamp;
    __u8 hop_count;
    double metric;
};
struct MAP_Proxying_Table {
    struct MPP_Proxy_Entry MPP_Entries[MPP_COUNT];
    int active_MPP; // subscript in the Array,
    int visible_MPPs;
};

struct AirTIME {
    __u8 mac_addr[6];
    double df, airtime, etx;
};

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

// ----- For Controlling Mesh Visibility,
__u8 blocked_4F[6] = MP_4F;

int get_MPP_Entry(struct MAP_Proxying_Table table, __u8 *mac_addr);
int get_Best_MPP_Entry(struct MAP_Proxying_Table table, struct MPP_Proxy_Entry
MPP_Entry, int index);
int get_MP_AirTIMEi_entry(struct AirTIME *AirTIMEi, __u8 *mac_addr);
void print_mac_addr(__u8 * mac);
void print_PREQ(struct PREQ preq);
int comp_mac_addr(__u8 * mac1, __u8 * mac2);
void print_AirTIMEi (struct AirTIME *AirTIMEi);
void print_MAP_Proxying_Table(struct MAP_Proxying_Table table);

int main(int argc, char* argv[]) {
    __u8 buffer[4096], buffer2[4096], *shm, *shm_AirTIME;
    struct sockaddr dest, name;
    int shmid, shmid_AirTIME, dlen, recievedBytes, len, timeStamp, refStamp,
currentMPP;
    int raw_sock, raw_sock2, fd, i, j, flag = 0;
    __u64 cnt = 0, k = 0;
    double link_metric;
    key_t key, key_AirTIME;

    struct PREQ preq;
    struct RREP rrep;
    struct ifreq ifr;
    struct sockaddr_ll sll;

```

```

struct MAP_Proxying_Table MAP_proxying_table;
struct MPP_Proxy_Entry MPP_Entry;
struct AirTIME AirTIMEi[MP_COUNT];

// Checking for Arguments,
if (argc < 2) {
    perror("\n Inussufficient 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 the shared memory for MAP_entry,
key = KEY;
if ((shmid = shmget(key, SHARED_MEMORY_SIZE, IPC_CREAT)) < 0) {
    perror("shmget");
}

```

```

        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == NULL) {
        perror("shmat");
        exit(1);
    }
    // Locating Shared Memory for AirTIMEi[] entries,
    key_AirTIME = KEY_AIRTIME;
    if ((shm_AirTIME = shmget(key_AirTIME, AirTIME_SHARED_MEMORY_SIZE,
IPC_CREAT)) < 0) {
        perror("shmget");
        exit(1);
    }

    // Reading AirTIMEi[] entries,
    if ((shm_AirTIME = shmat(shmid_AirTIME, NULL, 0)) == NULL) {
        perror("shmat");
        exit(1);
    }
    memcpy(AirTIMEi, shm_AirTIME, AirTIME_SHARED_MEMORY_SIZE);
    print_AirTIMEi(AirTIMEi);

    // Initializing the MAP_Proxying_Table,
    for (i = 0; i < MPP_COUNT; i++) {
        memcpy(MAP_proxying_table.MPP_Entries[i].MPP_mac_addr, NULL_addr, 6);
        memcpy(MAP_proxying_table.MPP_Entries[i].next_hop, NULL_addr, 6);
        MAP_proxying_table.MPP_Entries[i].seq_no = 0;
        MAP_proxying_table.MPP_Entries[i].hop_count = 0;
        MAP_proxying_table.MPP_Entries[i].metric = 0;
    }
    MAP_proxying_table.active_MPP = -1;
    MAP_proxying_table.visible_MPPs = 0;

    // ----- MAIN LOOP
    -----
    while(1) {
        recievedBytes = recvfrom(raw_sock,buffer,4096,0,&dest,&dlen);
        memcpy(src_mac_addr, buffer+6, 6);
        if (recievedBytes != 32) // Size of a PREQ Frame,
            continue; // Checking for PREQ Frames,

        // ----- WMN Visibility Control -----//
        if (comp_mac_addr(src_mac_addr, blocked_4F))
            continue;
        // -----//

        if (comp_mac_addr(src_mac_addr, local_mac_addr))
            continue; // Avoiding local loops,

        // Getting the PREQ,
        memcpy(&preq, buffer+12, PREQ_SIZE);

        //Checking if it is a PREQ_BROADCAST frame,
        if (preq.eltID != 13)

```

```

        continue;
// Getting the Time Stamp,
timeStamp = time(NULL);
if (flag == 0) { // Stamping the first PREQ in the set (having the same
Seq No),
        refStamp = timeStamp;
        flag = 1;
    }

// Getting the MPP sending the PREQ,
memcpy(MPP_mac_addr, preq.MPP_mac_addr, 6);

// --- Feedback,
printf("\n - %d - PREQ relevant to MPP:",
cnt++);print_mac_addr(MPP_mac_addr); printf(",Received from: ");
print_mac_addr(src_mac_addr);
// Getting the corresponding entry in the MAP_Proxying_Tables,
currentMPP = get_MPP_Entry(MAP_proxying_table, MPP_mac_addr);
if (currentMPP == -1) { // There is NO entry, create one
        currentMPP = MAP_proxying_table.visible_MPPs++;
        memcpy(MAP_proxying_table.MPP_Entries[currentMPP].MPP_mac_addr,
MPP_mac_addr, 6);
        memcpy(MAP_proxying_table.MPP_Entries[currentMPP].next_hop,
src_mac_addr, 6);
        MAP_proxying_table.MPP_Entries[currentMPP].seq_no = 0;
        MAP_proxying_table.MPP_Entries[currentMPP].metric = 0;
    }
MPP_Entry = MAP_proxying_table.MPP_Entries[currentMPP];
// Time Stamping the Entry,
MPP_Entry.timeStamp = timeStamp;

// 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;
// Update Routing Entry regardless of whether it is a better route
or not,

        memcpy(MPP_Entry.next_hop, src_mac_addr, 6);
        MAP_proxying_table.active_MPP = currentMPP;
        // GET LINK METRIC,
        // .....
        // 1. Get AirTIMEi,
        if ((shm_AirTIME = shmat(shmid_AirTIME, NULL, 0)) == NULL) {
            perror("shmat");
            exit(1);
        }
        memcpy(AirTIMEi, shm_AirTIME, AirTIME_SHARED_MEMORY_SIZE);
        // 2. Get AirTIMEi Entry for the Link,
        i = get_MP_AirTIMEi_entry(AirTIMEi, src_mac_addr);
        if (i == -1) // Entry not found,
            link_metric = 100; // Infinity,
        else

```

```

        link_metric = AirTIMEi[i].airtime;
        // Update MPP_Entry,
        MPP_Entry.metric = preq.metric + link_metric;
        MPP_Entry.hop_count = preq.hop_count + 1;
    } else { // Same PREQ freshness,
        // Update Routing Entry ONLY if this corresponds to a better
route,

        // GET LINK METRIC,
        // .....
        // 1. Get AirTIMEi,
        if ((shm_AirTIME = shmat(shmid_AirTIME, NULL, 0)) == NULL) {
            perror("shmat");
            exit(1);
        }
        memcpy(AirTIMEi, shm_AirTIME, AirTIME_SHARED_MEMORY_SIZE);
        // 2. Get AirTIMEi Entry for the Link,
        i = get_MP_AirTIMEi_entry(AirTIMEi, src_mac_addr);
        if (i == -1) // Entry not found,
            link_metric = 100; // Infinity,
        else
            link_metric = AirTIMEi[i].airtime;
        preq.metric += link_metric;
        if (preq.metric < MPP_Entry.metric) {
            MPP_Entry.metric = preq.metric;
            memcpy(MPP_Entry.next_hop, src_mac_addr, 6);
            MPP_Entry.hop_count = preq.hop_count + 1;
        }
    }

    // Updating the MAP Proxying Table,
    MAP_proxying_table.MPP_Entries[currentMPP] = MPP_Entry;
    // Compare with other MPP entries and select the entry with the best
route and fresh timeStamp,
    currentMPP = get_Best_MPP_Entry(MAP_proxying_table, MPP_Entry,
currentMPP);
    MAP_proxying_table.active_MPP = currentMPP;
    if (timeStamp > refStamp+1) { // Avoiding The Writing of temporary good
routes, i.e., Avoiding Route Fluctuations - Assuming a minimum PREQ Broadcast Period
of 3 Secs,

        flag = 0;
        // Select the best route in this period,
        memcpy(shm, &MAP_proxying_table, SHARED_MEMORY_SIZE);
        refStamp = timeStamp;
        print_MAP_Proxying_Table(MAP_proxying_table);
        // Prepare a RREP,
        rrep.eltID = 05;
        memcpy(rrep.MPP_mac_addr, MPP_Entry.MPP_mac_addr, 6);
        memcpy(rrep.MAP_mac_addr, local_mac_addr, 6);
        rrep.MAP_seq_no = MPP_Entry.seq_no;
        rrep.metric = 0; // The metric will be set at the receiver,
        rrep.hop_count = 0;
        // Reply using the Routing Entry,
        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 ((j = sendto(raw_sock2, buffer2, 12+RREP_SIZE, 0x00,
            (struct sockaddr *)&sll, sizeof(sll))) < 0 ) {
            perror("sendto");
            exit(-1);
        }
        printf("\n = RREP N'%d sent back to --> ", k++);
        print_mac_addr(MPP_Entry.next_hop);
    }

}

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 0;
    return 1;
}

void print_AirTIMEi (struct AirTIME *AirTIMEi) {
    int i;
    for (i = 0; i < MP_COUNT; i++) {
        printf("\n-----");
        print_mac_addr(AirTIMEi[i].mac_addr);
        printf(" --- df: %.2f", AirTIMEi[i].df);
        printf(" --- Airtime: %.3f", AirTIMEi[i].airtime);
        printf(" --- etx: %.2f", AirTIMEi[i].etx);

    }
}

int get_MPP_Entry(struct MAP_Proxying_Table table, __u8 *mac_addr) {
    int i;
    for (i = 0; i < MPP_COUNT; i++)
        if (comp_mac_addr(table.MPP_Entries[i].MPP_mac_addr, mac_addr))
            return i;
    return -1;
}

int get_MP_AirTIMEi_entry(struct AirTIME *AirTIMEi, __u8 *mac_addr) {

```

```

    int i;
    for (i = 0; i < MP_COUNT; i++)
        if (comp_mac_addr(AirTIMEi[i].mac_addr, mac_addr))
            return i;
    return -1;
}
int get_Best_MPP_Entry(struct MAP_Proxying_Table table, struct MPP_Proxy_Entry
MPP_Entry, int index) {
    int i, best = index, c_time;

    c_time = time(NULL);
    for (i = 0; i < table.visible_MPPs; i++)
        if (table.MPP_Entries[i].metric < MPP_Entry.metric &&
table.MPP_Entries[i].timeStamp <= c_time - ROUTE_ENTRY_LIFETIME && i != index)
            best = i;
    return best;
}
void print_MAP_Proxying_Table(struct MAP_Proxying_Table table) {
    int i;
    printf("\n ----- MAP Proxying Table ----- ");
    printf("\n - Visible MPPs: %d", table.visible_MPPs);
    printf("\n - Active MPP: %d", table.active_MPP+1);
    for (i = 0; i < table.visible_MPPs; i++) {
        printf("\n ----- MPP: %d ", i+1);
        printf("\n ----- MAC addr: ");
print_mac_addr(table.MPP_Entries[i].MPP_mac_addr);
        printf("\n ----- Next hop: ");
print_mac_addr(table.MPP_Entries[i].next_hop);
        printf("\n ----- Time Stamp: %d",table.MPP_Entries[i].timeStamp);
        printf("\n ----- Metric: %.3f",table.MPP_Entries[i].metric);
    }
    printf("\n ----- ");
}

```