

```

/* -----
(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 PREQ_SIZE sizeof(struct PREQ)
#define RREP_SIZE sizeof(struct RREP)
#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 R_SINRs_SHARED_MEMORY_SIZE (MP_COUNT+1) * sizeof(struct reverse_SINR)
#define KEY_R_SINR 1331131
#define KEY 777777

#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 reverse_SINR {
    __u8 mac_addr[6];
    double R_SINR, avg_rssi, avg_rate, avg_length;
    double interference_RSSI_sum;
};
struct neighbor_entry {
    __u8 mac_addr[6];
    double R_SINR;
    double F_SINR;
    double CCI;
    double avg_rate, avg_length;
    double airtime;
    double ICE;
};

__u8 local_mac_addr[6], src_mac_addr[6];
__u8 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);
void print_mac_addr(__u8 * mac);
void print_PREQ(struct PREQ preq);
int comp_mac_addr(__u8 * mac1, __u8 * mac2);
void print_MAP_Proxying_Table(struct MAP_Proxying_Table table);
void print_R_SINRs(struct reverse_SINR *R_SINRs);
int get_WMN_Neighbor_Entry(struct neighbor_entry* table, __u8 *mac_addr);
double get_R_SINR(struct reverse_SINR *Piggybacked_R_SINRs, __u8 *mac_addr, double*
avg_rate, double* avg_length);

int main(int argc, char* argv[]) {

```

```

__u8 buffer[4096], buffer2[4096], *shm, *shm_R_SINRs;
struct sockaddr dest, name;
int shmid, shmid_R_SINRs, dlen, recievedBytes, len, timeStamp, refStamp,
currentMPP;
int raw_sock, raw_sock2, fd, i, j, flag = 0,PREQ_size = sizeof(struct PREQ),
RREP_size = sizeof(struct RREP);
__u64 cnt = 0, k = 0;
double link_metric, avg_rate, avg_length, R_SINR;
key_t key, key_AirTIME, key_R_SINRs;

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 reverse_SINR R_SINRs[MP_COUNT+1], Piggybacked_R_SINRs[MP_COUNT+1];
struct neighbor_entry WMN_neighbors[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 the shared memory for MAP_entry,
key = KEY;
if ((shmidx = shmget(key, SHARED_MEMORY_SIZE, IPC_CREAT)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmidx, NULL, 0)) == NULL) {
    perror("shmat");
    exit(1);
}
// 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;

// Locating R_SINRs Shared Memory,
key_R_SINRs = KEY_R_SINR;
if ((shmidx_R_SINRs = shmget(key_R_SINRs, R_SINRs_SHARED_MEMORY_SIZE,
IPC_CREAT)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm_R_SINRs = shmat(shmidx_R_SINRs, NULL, 0)) == NULL) {
    perror("shmat");
    exit(1);
}
memcpy(R_SINRs, shm_R_SINRs, R_SINRs_SHARED_MEMORY_SIZE);
print_R_SINRs(R_SINRs);

// ----- MAIN LOOP
-----
while(1) {
    recievedBytes = recvfrom(raw_sock,buffer,4096,0,&dest,&dlen);
    memcpy(src_mac_addr, buffer+6, 6);
    // ----- 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 piggybacked R_SINRs Values,
memcpy(Piggybacked_R_SINRs, buffer+12+PREQ_size,
R_SINRs_SHARED_MEMORY_SIZE);
// Update the corresponding entry in the neighbors table,
i = get_WMN_Neighbor_Entry(WMN_neighbors, src_mac_addr);
if (i == -1) // A new neighbor - Create entry for it,
    for(i = 0; i < MP_COUNT; i++)
        if(comp_mac_addr(WMN_neighbors[i].mac_addr, NULL_addr)) {
            memcpy(WMN_neighbors[i].mac_addr, src_mac_addr, 6);
            break;
        }
    WMN_neighbors[i].F_SINR = get_R_SINR(Piggybacked_R_SINRs,
local_mac_addr, &avg_rate, &avg_length);
    WMN_neighbors[i].avg_rate = avg_rate;
    WMN_neighbors[i].avg_length = avg_length;
    WMN_neighbors[i].CCI = Piggybacked_R_SINRs[MP_COUNT].R_SINR;
// Feedback,
printf("\n *** R_SINR of this station at:");
print_mac_addr(src_mac_addr); printf(" is: %.2f --- CCI: %.2f",
WMN_neighbors[i].F_SINR, WMN_neighbors[i].CCI);

// 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;
        // ===== Computing Link ICE metric =====
        // 1. Get R_SINRs Shared Memory,
        memcpy(R_SINRs, shm_R_SINRs, R_SINRs_SHARED_MEMORY_SIZE);
        //printf("\n ===== Feedback Here =====");
        //print_R_SINRs(R_SINRs);

        // 2. Get Reverse SINR for PREQ Sender from locally computed
SINRs,

        R_SINR = get_R_SINR(R_SINRs, src_mac_addr, &avg_rate,
&avg_length);

        // 3. Compute ICE,
        link_metric = WMN_neighbors[i].CCI * WMN_neighbors[i].avg_length /
WMN_neighbors[i].avg_rate / (WMN_neighbors[i].F_SINR * R_SINR);
        // ===== Feedback, =====
/*
        printf("\n CCI: %.2f", WMN_neighbors[i].CCI);
        printf(" - avg_length: %.2f", WMN_neighbors[i].avg_length);
        printf(" - avg_rate: %.2f", WMN_neighbors[i].avg_rate);
        printf(" - f_SINR: %.2f", WMN_neighbors[i].F_SINR);
        printf(" - r_SINR: %.2f", R_SINR);
        printf(" - *** ICE: %.2f", link_metric);
*/

        //
=====

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

        // Feedback -
        // ===== Computing Link ICE metric =====

```

```

// 1. Get R_SINRs Shared Memory,
memcpy(R_SINRs, shm_R_SINRs, R_SINRs_SHARED_MEMORY_SIZE);
//printf("\n ===== Feedback Here =====");
//print_R_SINRs(R_SINRs);

// 2. Get Reverse SINR for PREQ Sender from locally computed
SINRs,
R_SINR = get_R_SINR(R_SINRs, src_mac_addr, &avg_rate,
&avg_length);

// 3. Compute ICE,
link_metric = WMN_neighbors[i].CCI * WMN_neighbors[i].avg_length /
WMN_neighbors[i].avg_rate / (WMN_neighbors[i].F_SINR * R_SINR);
// ===== Feedback,=====
/*
printf("\n CCI: %.2f", WMN_neighbors[i].CCI);
printf(" - avg_length: %.2f", WMN_neighbors[i].avg_length);
printf(" - avg_rate: %.2f", WMN_neighbors[i].avg_rate);
printf(" - f_SINR: %.2f", WMN_neighbors[i].F_SINR);
printf(" - r_SINR: %.2f", R_SINR);
printf(" - *** ICE: %.2f", link_metric);
*/

//
=====
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 - This corresponds to the next PERIOD, Assuming PREQ Broadcast period is at
least 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 = preq.metric; // The metric of the whole path as
computed in PREQ,
        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;
}

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_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: %.2f",table.MPP_Entries[i].metric);
    }
    printf("\n ----- ");
}

void print_R_SINRs(struct reverse_SINR *R_SINRs) {
    int i;
    for (i = 0; i < MP_COUNT; i++) {
        printf("\n *"); print_mac_addr(R_SINRs[i].mac_addr);
        printf(" - R_SINR: %.2f", R_SINRs[i].R_SINR);
    }
    printf("\n ----- CCI: %.2f", R_SINRs[MP_COUNT].R_SINR);
}

int get_WMN_Neighbor_Entry(struct neighbor_entry* table, __u8 *mac_addr) {
    int i;
    for (i = 0; i < MP_COUNT; i++)
        if (comp_mac_addr(table[i].mac_addr, mac_addr))
            return i;
    return -1;
}

double get_R_SINR(struct reverse_SINR *Piggybacked_R_SINRs, __u8 *mac_addr, double*
avg_rate, double* avg_length) {
    int i;
    for(i = 0; i < MP_COUNT; i++)
        if(comp_mac_addr(Piggybacked_R_SINRs[i].mac_addr, mac_addr)) {
            *avg_rate = Piggybacked_R_SINRs[i].avg_rate;
            *avg_length = Piggybacked_R_SINRs[i].avg_length;
            return Piggybacked_R_SINRs[i].R_SINR;
        }
    return 0.01; // A very Weak SINR,
}

```