

```

/* -----
(c) Riduan M ABID, eMail: R.Abid@auburn.edu
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 University - 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>

#define MP_COUNT 10
#define MPP_COUNT 5
#define MAP_COUNT 5
#define SHARED_MEMORY_SIZE sizeof(struct MP_Forwarding_Table)
#define R_SINRs_SHARED_MEMORY_SIZE (MP_COUNT+1) * sizeof(struct reverse_SINR)
#define KEY_R_SINR 1331131

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 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;
};
void print_R_SINRs(struct reverse_SINR *table);

__u8 local_mac_addr[6], src_mac_addr[6], dest_mac_addr[6];
__u8 broadcast_mac_addr[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}, NULL_addr[6] =
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, 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_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);
void print_MP_Forwarding_Table(struct MP_Forwarding_Table table);
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_AirTIME, *shm_RATE,
*shm_R_SINRs, eltID;
    struct sockaddr dest, name;
    int shmid, shmid_R_SINRs, shmid_RATE, dlen, cnt = 0, recievedBytes, len,
currentMPP, currentMAP;
    int raw_sock, raw_sock2, fd, i, PREQ_size = sizeof(struct PREQ), RREP_size =
sizeof(struct RREP), timeStamp, refStamp, n = 0;
    key_t key_MP_Table, key_R_SINRs;
    struct PREQ PREQ;
    struct RREP PREQ_reply;
    double link_metric;
    double avg_rate, avg_length, R_SINR, F_SINR;
    struct ifreq ifr;
    struct sockaddr_ll sll;
    struct MP_Forwarding_Table MP_forwarding_table;
    struct Proxy_Entry MPP_Entry, MAP_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 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 MP_Forwarding_Table Shared Memory,
    key_MP_Table = 131305;

    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 R_SINRs Shared Memory,
    key_R_SINRs = KEY_R_SINR;
    if ((shm_R_SINRs = shmget(key_R_SINRs, R_SINRs_SHARED_MEMORY_SIZE,
IPC_CREAT)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm_R_SINRs = shmat(shm_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);

    // 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;
    // Initializing the WMN_Neighborsm
    for (i = 0; i < MP_COUNT; i++)
        memcpy(WMN_neighbors[i].mac_addr, NULL_addr, 6);

    // ----- MAIN LOOP
    -----
    refStamp = time(NULL);
    while(1) {
        recievedBytes = recvfrom(raw_sock,buffer,4096,0,&dest,&dlen);
        // Checking if it is a Routing Frame, i.e., a PREQ_BROADCAST OR a RREP
frame,
        memcpy(&eltID, buffer+12, 1);
        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 local loops,

        // Checking Routing Frame Type,
        if (eltID == 13) { // a PREQ_Broadcast Frame,
            printf("\n ....%d..... PREQ Broadcast --- Received FROM: ----",

```

```

cnt++);

        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 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];

            // ===== 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);

        //
=====
        // 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);
            // Insert Local R_SINRs,
            memcpy(buffer2+12+PREQ_size, R_SINRs,
R_SINRs_SHARED_MEMORY_SIZE);
            if ((i = sendto(raw_sock2, buffer2, 12+PREQ_size +
R_SINRs_SHARED_MEMORY_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);
        // Insert Local R_SINRs,
        memcpy(buffer2+12+PREQ_size, R_SINRs,
R_SINRs_SHARED_MEMORY_SIZE);

        if ((i = sendto(raw_sock2, buffer2,
12+PREQ_size+R_SINRs_SHARED_MEMORY_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 Frame, Coming from a MAP,
    memcpy(dest_mac_addr, buffer, 6);
    if(!comp_mac_addr(local_mac_addr, dest_mac_addr))
        continue; // ath1 is on Monitor mode!
    printf("\n ++++++ RREP Frame Processing ----- FROM:
- ");

    print_mac_addr(src_mac_addr);
    // Getting the RREP Frame,
    memcpy(&PREQ_reply, buffer+12, RREP_size);

    // Getting the MAP sending the RREP,
    memcpy(MAP_mac_addr, PREQ_reply.MAP_mac_addr, 6);
    // Getting the destination MPP,
    memcpy(MPP_mac_addr, PREQ_reply.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 =
PREQ_reply.MAP_seq_no;
    MP_forwarding_table.MAP_Entries[currentMAP].hop_count =
PREQ_reply.hop_count+1;
    MP_forwarding_table.MAP_Entries[currentMAP].metric =
PREQ_reply.metric;
    printf("\n *** Route Metric: %.2f", PREQ_reply.metric);

```

```

        // Update the RREP metric,
        PREQ_reply.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];

        // Forwards the RREP,
        memcpy(buffer2, MPP_Entry.next_hop, 6);
        memcpy(buffer2+6, local_mac_addr, 6);
        memcpy(buffer2+12, &PREQ_reply, 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 ++ %d ++++++++ RREP forwarded TO ----> : ", n++);
print_mac_addr(MPP_Entry.next_hop);
    }
// Wrting the MP_Forwarding Table,
    timeStamp = time(NULL);
    if (timeStamp > refStamp+1) { // Avoiding writing temporary good routes,
        memcpy(shm_MP_Table, &MP_forwarding_table, SHARED_MEMORY_SIZE);
        refStamp = timeStamp;
        // Display the MP_Forwarding_table,
        print_MP_Forwarding_Table(MP_forwarding_table);
    }
}
}
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;
}
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,
}
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 ----- 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 ----- 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 ----- Whole Route Metric:
%.2f",table.MAP_Entries[i].metric);
    }
}
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_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);
}
```