

# 网络路由器实验

学号： 2021K8009929010

姓名： 贾城昊

## 一、 实验题目：网络路由器实验

## 二、 实验任务：

了解 mOSPF 路由机制和协议格式，掌握如何处理 Hello， LSU 消息以及如何构建一致性链路状态数据库，最后使用 Dijkstra 算法计算最短路径，并根据最短路径生成路由表。

## 三、 实验流程

### 实验内容一：

基于已有代码框架，实现路由器生成和处理 mOSPF Hello/LSU 消息的相关操作，构建一致性链路状态数据库，具体流程如下：

1. 运行网络拓扑(topo.py)
2. 在各个路由器节点上执行 disable\_arp.sh, disable\_icmp.sh, disable\_ip\_forward.sh), 禁止协议栈的相应功能
3. 运行./mospfd, 使得各个节点生成一致的链路状态数据库

### 实验内容二：

基于实验一，实现路由器计算路由表项的相关操作，具体流程如下：

1. 运行实验运行网络拓扑(topo.py)
2. 在各个路由器节点上执行 `disable_arp.sh`, `disable_icmp.sh`, `disable_ip_forward.sh`), 禁止协议栈的相应功能
3. 运行 `./mospfd`, 使得各个节点生成一致的链路状态数据库
4. 等待一段时间后, 每个节点生成完整的路由表项
5. 在节点 h1 上 ping/traceroute 节点 h2
6. 关掉某节点或链路, 等一段时间后, 再次用 h1 去 traceroute 节点 h2

## 四、实验过程

### (一) 链路邻居的发现

#### 1. mOSPF Hello 消息发送

Hello 消息的收发处理都需要在 mOSPF 互斥锁环境下完成, 因为涉及多个线程对相关信息的可能修改。`sending_mospf_hello_thread()` 函数每 5 秒被唤醒一次, 发送 hello 消息时并不清楚哪个端口有邻居节点, 因此无差别向所有端口发送即可, 表明节点存在且有效 (下面代码中即对每个端口调用 `send_mospf_hello_packet()` 函数)。

```

void *sending_mospf_hello_thread(void *param)
{
    //fprintf(stdout, "TODO: send mOSPF Hello message periodically.\n");
    while (1) {
        sleep(MOSPF_DEFAULT_HELLOINT);
        pthread_mutex_lock(&mospf_lock);

        iface_info_t *iface = NULL;
        list_for_each_entry(iface, &instance->iface_list, list) {
            send_mospf_hello_packet(iface);
        }

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}

```

Hello 消息采用组播扩散， 目的 IP 地址 224.0.0.5， 目的 MAC 地址为 01: 00: 5E: 00: 00: 05， 因此不用调用 ARP 模块。依次生成 Hello 消息、mOSPF 头部、IP 头部和以太网头部消息，然后交由端口发送即可。

具体代码如下所示：

```

void send_mospf_hello_packet(iface_info_t *iface){
    int msg_len = MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE;
    int len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + msg_len;
    char * hello_pkt = malloc(len);
    struct ether_header *eh_hdr = (struct ether_header *)hello_pkt;
    struct iphdr *ip_hdr = packet_to_ip_hdr(hello_pkt);
    struct mospf_hdr *mospf = (struct mospf_hdr *)((char *)ip_hdr + IP_BASE_HDR_SIZE);
    struct mospf_hello *mospf_he = (struct mospf_hello *)((char *)mospf + MOSPF_HDR_SIZE);

    memset(hello_pkt, 0, len);

    mospf_init_hello(mospf_he, iface->mask);

    mospf_init_hdr(mospf, MOSPF_TYPE_HELLO, msg_len, instance->router_id, instance->area_id);
    mospf->checksum = mospf_checksum(mospf);

    ip_init_hdr(ip_hdr, iface->ip, MOSPF_ALLSPFRouters, IP_BASE_HDR_SIZE + msg_len, IPPROTO_MOSPF);

    memcpy(eh_hdr->ether_dhost, eth_allrouter_addr, ETH_ALEN);
    memcpy(eh_hdr->ether_shost, iface->mac, ETH_ALEN);
    eh_hdr->ether_type = htons(ETH_P_IP);

    iface_send_packet(iface, hello_pkt, ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + msg_len);
    free(hello_pkt);
}

```

## 2. mOSPF Hello 消息处理

处理 hello 消息的逻辑较为简单，提取消息中 rid，查找邻居列表是否存在该节点。如果已经存在，更新存活时间和其他数据项；如果不存在，那么添加该节点。添加节点后，本路由器的邻居列表改变，因此还需要调用 send\_mospf\_lsu\_packet() 函数发送 LSU 消息。

最后值得注意的是，Hello 消息并不需要转发。具体代码如下所示：

```
void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)
{
    struct iphdr *ip_hdr = (struct iphdr *) (packet + ETHER_HDR_SIZE);
    struct mospf_hdr *mospf = (struct mospf_hdr *) ((char *) ip_hdr + IP_HDR_SIZE(ip_hdr));
    struct mospf_hello *mospf_he = (struct mospf_hello *) ((char *) mospf + MOSPF_HDR_SIZE);

    pthread_mutex_lock(&mospf_lock);
    mospf_nbr_t *nbr = NULL;
    list_for_each_entry(nbr, &iface->nbr_list, list) {
        if (nbr->nbr_id == ntohl(mospf->rid)) {
            nbr->nbr_ip = ntohl(ip_hdr->saddr);
            nbr->nbr_mask = ntohl(mospf_he->mask);
            nbr->alive = 0;

            pthread_mutex_unlock(&mospf_lock);
            return;
        }
    }

    mospf_nbr_t *new_nbr = (mospf_nbr_t *) malloc(sizeof(mospf_nbr_t));
    new_nbr->nbr_id = ntohl(mospf->rid);
    new_nbr->nbr_ip = ntohl(ip_hdr->saddr);
    new_nbr->nbr_mask = ntohl(mospf_he->mask);
    new_nbr->alive = 0;
    init_list_head(&new_nbr->list);
    list_add_tail(&new_nbr->list, &iface->nbr_list);
    //iface->helloint = ntohs(mospf_he->helloint);
    iface->num_nbr++;

    send_mospf_lsu_packet();
    pthread_mutex_unlock(&mospf_lock);
}
```

### 3. 邻居信息更新检查

检查邻居列表的存活时间，将超时没有发送 hello 的邻居删除。设定为每秒执行一次，如果列表中的节点存活未更新时间超过了  $3 * \text{hello-interval}$ ，则将其删除；否则，则更新其未更新的时间。

需要注意一点是，如果进行了删除操作，说明本路由器的邻居列表改变，因此还需要调用 `send_mospf_lsu_packet()` 函数发送 LSU 消息。

具体代码如下所示：

```
void *checking_nbr_thread(void *param)
{
    //fprintf(stdout, "TODO: neighbor list timeout operation.\n");
    while (1) {
        sleep(1);
        pthread_mutex_lock(&mospf_lock);

        int update = 0;
        iface_info_t *iface = NULL;
        list_for_each_entry(iface, &instance->iface_list, list) {
            mospf_nbr_t *nbr = NULL, *nbr_q = NULL;
            list_for_each_entry_safe(nbr, nbr_q, &iface->nbr_list, list) {
                if (nbr->alive > 3 * iface->helloint) {
                    update = 1;
                    iface->num_nbr--;
                    list_delete_entry(&nbr->list);
                    free(nbr);
                }
                else{
                    nbr->alive++;
                }
            }
        }

        if (update) {
            send_mospf_lsu_packet();
        }

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}
```

## （二）链路状态扩散

### 1. mOSPF LSU 消息发送

路由节点每隔一定时间间隔发送一次 LSU 消息，LSU 消息发送较为复杂，并且其他地方也会复用，因此将其封装为一个函数。同时发送完后打印一下当前数据库内容，方便对实验结果进行验证。

```

void *sending_mospf_lsu_thread(void *param)
{
    //fprintf(stdout, "TODO: send mOSPF LSU message periodically.\n");
    while (1) {
        sleep(MOSPF_DEFAULT_LSUINT);
        pthread_mutex_lock(&mospf_lock);

        send_mospf_lsu_packet();

        print_mospf_database();

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}

```

每次向每个有邻居的端口发送的 LSU 消息中 LSA 部分都完全相同，考虑到每条消息在发送时都需要独立空间，所以可以先统一生成 LSA 信息，再拷贝进入各数据包并生成各头部信息。

其中 LSA 生成的过程中，首先需要确定空间大小，所以首先需要遍历各端口的邻居列表，每条邻居信息需要一个 LSA 表项。对于没有邻居信息的端口同样需要一个 LSA 表项来显示端口存在。开辟空间后将邻居信息逐条填入即可，无邻居的端口用自身的网络号和掩码填充对应项，路由号填为全 0。

```

iface = NULL;
int i = 0;
list_for_each_entry(iface, &instance->iface_list, list) {
    if (iface->num_nbr) {
        mospf_nbr_t *nbr = NULL;
        list_for_each_entry(nbr, &iface->nbr_list, list) {
            mospf_lsa_array[i].network = iface->ip & iface->mask;
            mospf_lsa_array[i].mask = iface->mask;
            mospf_lsa_array[i].rid = nbr->nbr_id;
            i++;
        }
    }
    else {
        mospf_lsa_array[i].network = iface->ip & iface->mask;
        mospf_lsa_array[i].mask = iface->mask;
        mospf_lsa_array[i].rid = 0;
        i++;
    }
}

```

发送时只向有邻居节点的端口发送，对每个邻居单播发送，此时需要使用 ARP 发

送，要给每个报文分配单独数据包，数据包的回收交由 ARP 模块处理。

```
iface = NULL;
list_for_each_entry(iface, &instance->iface_list, list) {
    if (iface->num_nbr) {
        mospf_nbr_t *nbr = NULL;
        list_for_each_entry(nbr, &iface->nbr_list, list) {
            char *packet = (char *)malloc(ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + mospf_packet_len);
            struct ether_header *eh = (struct ether_header *)packet;
            struct iphdr *ip_hdr = (struct iphdr *) (packet + ETHER_HDR_SIZE);
            char *mospf_message = packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE;

            memset(packet, 0, ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + mospf_packet_len);
            memcpy(mospf_message, mospf_packet, mospf_packet_len);

            ip_init_hdr(ip_hdr, iface->ip, nbr->nbr_ip, IP_BASE_HDR_SIZE + mospf_packet_len, IPPROTO_MOSPF);

            memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
            eh->ether_type = htons(ETH_P_IP);

            iface_send_packet_by_arp(iface, nbr->nbr_ip, packet, ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + mospf_packet_len);
        }
    }
}
```

## 2. mOSPF LSU 消息处理

首先查看 lsu 消息的 rid，如果是自己发送的消息又被传回来了，丢弃：

```
pthread_mutex_lock(&mospf_lock);
if (instance->router_id == ntohl(mospf->rid)) {
    pthread_mutex_unlock(&mospf_lock);
    return;
}
```

然后根据数据包中的信息， 可以将 LSU 消息分为三种：新的链路状态信息、原有状态信息的更新、已记录或过期的状态信息。对于第三种情况，可以直接中止处理并丢弃本条消息；前两种情况需要通过遍历已有链路装填数据库来区分，并以参数 exist 和 update 来标识。

具体操作便是查找数据库是否已经有相同 rid 的条目。如果存在，标记已保存 (exist=1)，接着检查序列号，如果新收到的序列号大于数据库的，更新条目，标记已更新(update=1)；如果序列号不大于数据库的，丢弃。

```

mospf_db_entry_t *db_entry = NULL;
list_for_each_entry(db_entry, &mospf_db, list) {
    if (db_entry->rid == ntohl(mospf->rid)) {
        exist = 1;
        if (db_entry->seq < ntohs(mospf_ls->seq)) {
            db_entry->seq = ntohs(mospf_ls->seq);
            db_entry->nadv = ntohl(mospf_ls->nadv);
            db_entry->alive = 0;
            for (int i = 0; i < db_entry->nadv; i++) {
                db_entry->array[i].mask = lsa[i].mask;
                db_entry->array[i].network = lsa[i].network;
                db_entry->array[i].rid = lsa[i].rid;
                /*fprintf(stdout, "Update db entry "IP_FMT" "IP_FMT" "IP_FMT" "IP_FMT"\n",
                    HOST_IP_FMT_STR(db_entry->rid),
                    HOST_IP_FMT_STR(db_entry->array[i].network),
                    HOST_IP_FMT_STR(db_entry->array[i].mask),
                    HOST_IP_FMT_STR(db_entry->array[i].rid));*/
            }
            update = 1;
        }
    }
}

```

对于新的链路状态信息，需要开辟新的存储空间，并根据 LSU 消息的内容添加到一致性链路状态数据库中即可。

```

if (!exist) {
    db_entry = (mospf_db_entry_t *)malloc(sizeof(mospf_db_entry_t));
    db_entry->rid = ntohl(mospf->rid);
    db_entry->seq = ntohs(mospf_ls->seq);
    db_entry->nadv = ntohl(mospf_ls->nadv);
    db_entry->alive = 0;
    db_entry->array = (struct mospf_lsa *)malloc(MOSPF_LSA_SIZE * db_entry->nadv);
    for (int i = 0; i < db_entry->nadv; i++) {
        db_entry->array[i].mask = lsa[i].mask;
        db_entry->array[i].network = lsa[i].network;
        db_entry->array[i].rid = lsa[i].rid;
        /*fprintf(stdout, "Update db entry "IP_FMT" "IP_FMT" "IP_FMT" "IP_FMT"\n",
            HOST_IP_FMT_STR(db_entry->rid),
            HOST_IP_FMT_STR(db_entry->array[i].network),
            HOST_IP_FMT_STR(db_entry->array[i].mask),
            HOST_IP_FMT_STR(db_entry->array[i].rid));*/
    }
    init_list_head(&db_entry->list);
    list_add_tail(&db_entry->list, &mospf_db);

    update = 1;
}

```

最后 LSU 消息可能需要转发，如果之前没有保存条目，或者条目得到了更新，需要转发消息。检查其 tt1，如果未减为 0，向所有邻居单播转发该消息。注意此时需要调用 ARP 模块发送，要给每个邻居分配一个数据包，数据包的回收交由 ARP 模块负责。



```

if(!update){
    pthread_mutex_unlock(&mospf_lock);
    return;
}

mospf_ls->tll -= 1;
if (mospf_ls->tll > 0) {
    mospf->checksum = mospf_checksum(mospf);
    iface_info_t *iface_out = NULL;

    list_for_each_entry(iface_out, &instance->iface_list, list) {
        if (iface_out->num_nbr && iface_out != iface) {
            mospf_nbr_t *nbr = NULL;
            list_for_each_entry(nbr, &iface_out->nbr_list, list) {
                char *packet_out = (char *)malloc(len);
                struct ether_header *eh_out = (struct ether_header *)packet_out;
                struct iphdr *ip_out = (struct iphdr *)(packet_out + ETHER_HDR_SIZE);
                memcpy(packet_out, packet, len);

                ip_out->daddr = htonl(nbr->nbr_ip);
                ip_out->checksum = ip_checksum(ip_out);

                memcpy(eh_out->ether_shost, iface_out->mac, ETH_ALEN);
                eh_out->ether_type = htons(ETH_P_IP);

                iface_send_packet_by_arp(iface_out, nbr->nbr_ip, packet_out, len);
            }
        }
    }
}
update_route_table();

pthread_mutex_unlock(&mospf_lock);

```

最后如果数据库发生了更新，需要调用 `update_route_table()` 函数更新路由表。

### 3. 链路状态数据库老化处理

本设计中，链路状态数据库每 1 秒进行一次老化检查，如果 3. 链路状态数据库中的条目在 40s 时间内未更新，则将其删除。需要注意，如果进行了删除操作，说明 3.链路状态数据库发生改变，因此需要调用 `update_route_table()` 函数生成最新的路由表。

具体代码如下所示：

```

void *checking_database_thread(void *param)
{
    //fprintf(stdout, "TODO: link state database timeout operation.\n");
    while (1) {
        sleep(1);
        pthread_mutex_lock(&mospf_lock);

        int update = 0;
        mospf_db_entry_t *db_entry = NULL, *db_q = NULL;
        list_for_each_entry_safe(db_entry, db_q, &mospf_db, list) {
            if (db_entry->alive > MOSPF_DATABASE_TIMEOUT) {
                update = 1;
                list_delete_entry(&db_entry->list);
                free(db_entry->array);
                free(db_entry);
            }
            else {
                db_entry->alive++;
            }
        }

        if (update) {
            update_route_table();
        }

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}

```

### （三）最短路径计算与路由表的生成

#### 1. 总体流程

使用 Dijkstra 算法计算最短路径，这里选用一个邻接矩阵来记录节点间的连通关系，从而实现链路数据库到路由表项的计算。而更新（生成）路由表的过程首先是清除现有的旧路由表（但保留内核中读取的默认路由项），然后根据链路数据库构建图的邻接矩阵，接着调用 Dijkstra 算法计算最短路径，最后生成新的路由表。

具体如下所示：

```

void update_route_table(void)
{
    clear_route_table();
    build_rid_map();

    init_graph();
    Dijkstra();

    build_route_table();
    print_rtable();

    return;
}

```

下面针对每个函数进行更详细的介绍。

## 2. 清除旧的路由表

根据 gw 来清除旧的路由表，即保留内核中读取的默认路由项(gw=0)，清除其他项。

```

void clear_route_table(void)
{
    rt_entry_t *rt_entry, *rt_q;
    list_for_each_entry_safe(rt_entry, rt_q, &rttable, list) {
        if (rt_entry->gw) {
            remove_rt_entry(rt_entry);
        }
    }
}

```

## 3. 构建节点 rid 到图 index 的映射

将自己设为 0 号节点，将邻居列表和链路数据库中各节点依次编号，包括链路数据库中节点的邻居。注意在遍历数据库节点的邻居时，rid=0 的邻居表示实际不存在邻居节点，编号时去掉该项即可。

```

void build_rid_map(void)
{
    node_map[0] = instance->router_id;
    node_num = 1;

    iface_info_t *iface = NULL;
    list_for_each_entry(iface, &instance->iface_list, list) {
        if (iface->num_nbr) {
            mospf_nbr_t *nbr = NULL;
            list_for_each_entry(nbr, &iface->nbr_list, list) {
                if (!rid_map_existed(nbr->nbr_id)) {
                    node_map[node_num++] = nbr->nbr_id;
                }
            }
        }
    }

    mospf_db_entry_t *db_entry = NULL;
    list_for_each_entry(db_entry, &mospf_db, list) {
        if (!rid_map_existed(db_entry->rid)) {
            node_map[node_num++] = db_entry->rid;
        }
        for (int i=0; i<db_entry->nadv; i++) {
            if (db_entry->array[i].rid && !rid_map_existed(db_entry->array[i].rid))
                node_map[node_num++] = db_entry->array[i].rid;
        }
    }
}

```

#### 4. 图的邻接矩阵的初始化

对邻接矩阵进行初始化时，将直接可达的节点间距离设为 1，不可达节点设为 INT\_MAX，节点自己到自己设为 0。但是值得注意的是，INT\_MAX 不是设为实际 int 的最大值，因为之后会有  $\text{dist}[i] + \text{graph}[i][i]$  的计算，这两项默认值都为 INT\_MAX，需要保证其和不会溢出。

具体如下所示：

```

void init_graph(void)
{
    for (int i=0; i<node_num; i++) {
        for (int j=0; j<node_num; j++) {
            if (i==j)
                graph[i][j] = 0;
            else
                graph[i][j] = INT_MAX;
        }
    }

    iface_info_t *iface = NULL;
    list_for_each_entry(iface, &instance->iface_list, list) {
        if (iface->num_nbr) {
            mospf_nbr_t *nbr = NULL;
            list_for_each_entry(nbr, &iface->nbr_list, list) {
                graph[0][rid_to_index(nbr->nbr_id)] = 1;
                graph[rid_to_index(nbr->nbr_id)][0] = 1;
            }
        }
    }

    mospf_db_entry_t *db_entry = NULL;
    list_for_each_entry(db_entry, &mospf_db, list) {
        int x = rid_to_index(db_entry->rid);
        for (int i=0; i<db_entry->nadv; i++) {
            if (db_entry->array[i].rid) {
                graph[x][rid_to_index(db_entry->array[i].rid)] = 1;
                graph[rid_to_index(db_entry->array[i].rid)][x] = 1;
            }
        }
    }
}

```

## 5. Dijkstra 算法

以该路由器为根节点，计算最短路径。算法与讲义上一致，不过需要增加一个 stack 用于保存选取节点的顺序。生成路由表时需要按路径长度从小到大遍历节点，这一顺序与 min\_dist 选出的顺序一致(Dijkstra 算法每次选择当前距离起点最近的顶点进行处理)。

具体代码如下图所示：

```

void Dijkstra(void)
{
    int dist[MAX_NODE_NUM];
    int visit[MAX_NODE_NUM];

    for (int i = 0; i < node_num; i++) {
        dist[i] = INT_MAX;
        visit[i] = 0;
        prev[i] = -1;
    }
    dist[0] = 0;
    stack_top = 0;

    for (int i = 0; i < node_num; i++) {
        int u = min_dist(dist, visit);
        visit[u] = 1;
        stack[stack_top++] = u;

        for (int v = 0; v < node_num; v++) {
            if (!visit[v] && (graph[u][v] + dist[u] < dist[v])) {
                dist[v] = graph[u][v] + dist[u];
                prev[v] = u;
            }
        }
    }
}

```

## 6. 生成新的路由表

按路径长度从小到大遍历节点来生成路由表项（实际只用从 stack 中依次取出即可）。

由于 stack[0] 是该路由器本身，而我们清楚旧路由表项的时候保留了其到邻居节点的路由表项（默认路由项），因此不用对其进行计算，所以我们从 stack[1] 开始遍历即可。

对每个节点，首先尝试在链路数据库里找出其对应的条目。如果此时路由器刚启动，链路数据库还未收敛，那么可能查找失败。这是因为该节点的 hello 先到了其邻居，然后其邻居的 1su 消息到了本路由器，所以本路由器知道了其存在，但与此同时其 1su 还未到本路由器，所以导致链路数据库没有其对应的条目，导致查找失败。这时跳过该节点即可。

（注：当链路数据库和每个节点的邻居列表收敛后不会出现次情况）

```

node_now = stack[i];
mospf_db_entry_t *db_tmp = NULL;
list_for_each_entry(db_tmp, &mospf_db, list) {
    if (db_tmp->rid == node_map[node_now]) {
        db_entry = db_tmp;
        break;
    }
}

if (!db_entry)
    continue;

```

然后从不断通过 prev 查找这个节点的前一跳节点，直到其前一跳是本路由器节点，这时我们就知道了该通过哪个邻居作为到达那个节点的下一跳节点。

但是在未收敛时，也有可能找不到对应 rid 的邻居。这是因为本路由器的 hello 先到达邻居，因而邻居的邻居列表有本路由器，但是邻居的 lsu 消息比其 hello 先到达，导致本路由器邻居列表没有该邻居节点，但初始化图的时候从邻居的 lsu 知道邻居能到达自己，对于这种情况也跳过即可。

(注：当链路数据库和每个节点的邻居列表收敛后不会出现次情况)

```

while (prev[node_now] != 0) {
    node_now = prev[node_now];
}
iface_info_t *iface = NULL;
list_for_each_entry(iface, &instance->iface_list, list) {
    if (iface->num_nbr) {
        mospf_nbr_t *nbr = NULL;
        list_for_each_entry(nbr, &iface->nbr_list, list) {
            if (nbr->nbr_id == node_map[node_now]) {
                iface_out = iface;
                gw = nbr->nbr_ip;
                break;
            }
        }
    }
}

if (!iface_out) {
    continue;
}

```

如果找到了邻居，那我们就知道了下一跳节点的 ip、端口，就可以添加路由表项了，

但需要首先查看已经部分构建的路由表，检查如果还没有添加其路由表项，就增加其路由表项。

```
for (int j=0; j<db_entry->nadv; j++) {
    find = 0;
    rt_entry_t *rt_entry = NULL;
    list_for_each_entry(rt_entry, &rtable, list) {
        if (rt_entry->dest == db_entry->array[j].network) {
            find = 1;
            break;
        }
    }

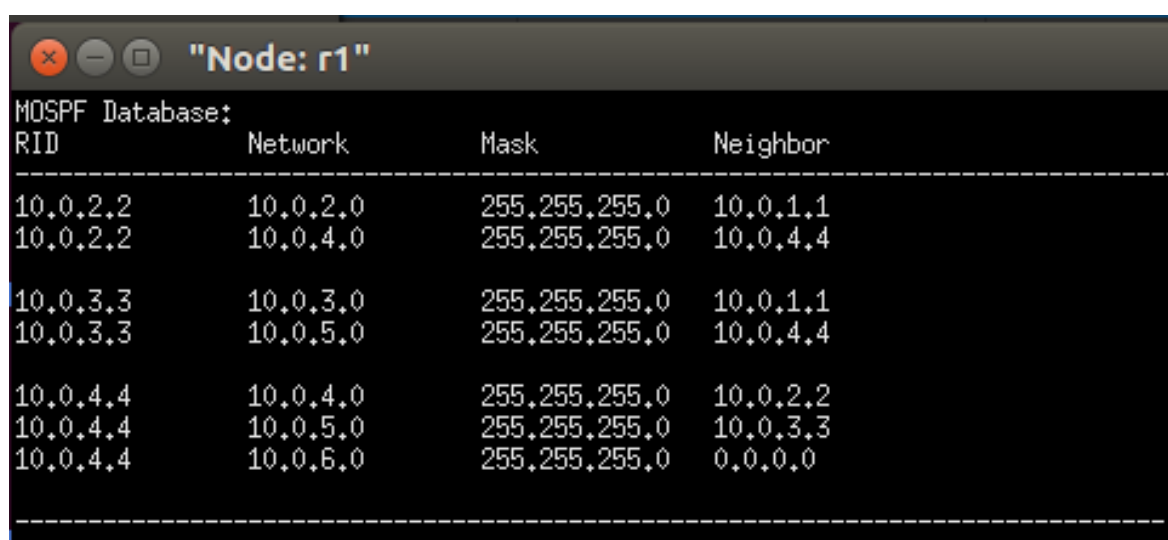
    if (find == 0) {
        rt_entry = new_rt_entry(db_entry->array[j].network, db_entry->array[j].mask, gw, iface_out);
        add_rt_entry(rt_entry);
    }
}
```

## 五、实验结果与分析

### （一）链路数据库一致性验证

执行 topo.py 脚本，在 r1, r2, r3, r4 结点上启 mospfd 程序，一段时间后查看其链路状态数据库，结果如下：

r1 链路数据库：



MOSPF Database:			
RID	Network	Mask	Neighbor
10.0.2.2	10.0.2.0	255.255.255.0	10.0.1.1
10.0.2.2	10.0.4.0	255.255.255.0	10.0.4.4
10.0.3.3	10.0.3.0	255.255.255.0	10.0.1.1
10.0.3.3	10.0.5.0	255.255.255.0	10.0.4.4
10.0.4.4	10.0.4.0	255.255.255.0	10.0.2.2
10.0.4.4	10.0.5.0	255.255.255.0	10.0.3.3
10.0.4.4	10.0.6.0	255.255.255.0	0.0.0.0



r2 链路数据库:

"Node: r2"			
MOSPF Database:			
RID	Network	Mask	Neighbor
10.0.4.4	10.0.4.0	255.255.255.0	10.0.2.2
10.0.4.4	10.0.5.0	255.255.255.0	10.0.3.3
10.0.4.4	10.0.6.0	255.255.255.0	0.0.0.0
10.0.1.1	10.0.1.0	255.255.255.0	0.0.0.0
10.0.1.1	10.0.2.0	255.255.255.0	10.0.2.2
10.0.1.1	10.0.3.0	255.255.255.0	10.0.3.3
10.0.3.3	10.0.3.0	255.255.255.0	10.0.1.1
10.0.3.3	10.0.5.0	255.255.255.0	10.0.4.4

r3 链路数据库:

"Node: r3"			
10.0.2.0	255.255.255.0	10.0.3.1	r3-eth0
10.0.4.0	255.255.255.0	10.0.5.4	r3-eth1
10.0.6.0	255.255.255.0	10.0.5.4	r3-eth1
MOSPF Database:			
RID	Network	Mask	Neighbor
10.0.4.4	10.0.4.0	255.255.255.0	10.0.2.2
10.0.4.4	10.0.5.0	255.255.255.0	10.0.3.3
10.0.4.4	10.0.6.0	255.255.255.0	0.0.0.0
10.0.2.2	10.0.2.0	255.255.255.0	10.0.1.1
10.0.2.2	10.0.4.0	255.255.255.0	10.0.4.4
10.0.1.1	10.0.1.0	255.255.255.0	0.0.0.0
10.0.1.1	10.0.2.0	255.255.255.0	10.0.2.2
10.0.1.1	10.0.3.0	255.255.255.0	10.0.3.3

r4 链路数据库:

"Node: r4"			
10.0.2.0	255.255.255.0	10.0.4.2	r4-eth0
10.0.3.0	255.255.255.0	10.0.5.3	r4-eth1
10.0.1.0	255.255.255.0	10.0.4.2	r4-eth0
-----			
MOSPF Database:			
RID	Network	Mask	Neighbor
-----			
10.0.3.3	10.0.3.0	255.255.255.0	10.0.1.1
10.0.3.3	10.0.5.0	255.255.255.0	10.0.4.4
10.0.2.2	10.0.2.0	255.255.255.0	10.0.1.1
10.0.2.2	10.0.4.0	255.255.255.0	10.0.4.4
10.0.1.1	10.0.1.0	255.255.255.0	0.0.0.0
10.0.1.1	10.0.2.0	255.255.255.0	10.0.2.2
10.0.1.1	10.0.3.0	255.255.255.0	10.0.3.3
-----			

可以看出 4 个节点的链路状态数据库一致且正确。

## (二) 路由表计算结果验证

执行 topo.py 脚本，在 r1, r2, r3, r4 结点上启 mospfd 程序，一段时间后查看四个路由器的路由表，结果如下：

"Node: r1"			
Routing Table:			
dest	mask	gateway	if_name
-----			
10.0.1.0	255.255.255.0	0.0.0.0	r1-eth0
10.0.2.0	255.255.255.0	0.0.0.0	r1-eth1
10.0.3.0	255.255.255.0	0.0.0.0	r1-eth2
10.0.4.0	255.255.255.0	10.0.2.2	r1-eth1
10.0.5.0	255.255.255.0	10.0.3.3	r1-eth2
10.0.6.0	255.255.255.0	10.0.2.2	r1-eth1
-----			

"Node: r2"			
Routing Table:			
dest	mask	gateway	if_name
-----			
10.0.2.0	255.255.255.0	0.0.0.0	r2-eth0
10.0.4.0	255.255.255.0	0.0.0.0	r2-eth1
10.0.1.0	255.255.255.0	10.0.2.1	r2-eth0
10.0.3.0	255.255.255.0	10.0.2.1	r2-eth0
10.0.5.0	255.255.255.0	10.0.4.4	r2-eth1
10.0.6.0	255.255.255.0	10.0.4.4	r2-eth1
-----			

```
"Node: r3"
-
Routing Table:
dest      mask            gateway         if_name
-----
10.0.3.0   255.255.255.0   0.0.0.0         r3-eth0
10.0.5.0   255.255.255.0   0.0.0.0         r3-eth1
10.0.1.0   255.255.255.0   10.0.3.1        r3-eth0
10.0.2.0   255.255.255.0   10.0.3.1        r3-eth0
10.0.4.0   255.255.255.0   10.0.5.4        r3-eth1
10.0.6.0   255.255.255.0   10.0.5.4        r3-eth1
-----
```

```
"Node: r4"
-----
Routing Table:
dest      mask            gateway         if_name
-----
10.0.4.0   255.255.255.0   0.0.0.0         r4-eth0
10.0.5.0   255.255.255.0   0.0.0.0         r4-eth1
10.0.6.0   255.255.255.0   0.0.0.0         r4-eth2
10.0.2.0   255.255.255.0   10.0.4.2        r4-eth0
10.0.3.0   255.255.255.0   10.0.5.3        r4-eth1
10.0.1.0   255.255.255.0   10.0.4.2        r4-eth0
-----
```

可以看出各网络的路由表项存在且正确。

同时，在 h1 主机上 ping h2 主机，可以 ping 通：

```
"Node: h1"
root@Computer:~/workspace/Network/lab8/09-mospf# ping -c 4 10.0.6.22
PING 10.0.6.22 (10.0.6.22) 56(84) bytes of data.
64 bytes from 10.0.6.22: icmp_seq=1 ttl=61 time=1.35 ms
64 bytes from 10.0.6.22: icmp_seq=2 ttl=61 time=0.150 ms
64 bytes from 10.0.6.22: icmp_seq=3 ttl=61 time=0.158 ms
64 bytes from 10.0.6.22: icmp_seq=4 ttl=61 time=1.04 ms

--- 10.0.6.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 0.150/0.676/1.358/0.535 ms
root@Computer:~/workspace/Network/lab8/09-mospf#
```

在 h1 主机上 traceroute h2 主机，路径正确：

```
root@Computer:~/workspace/Network/lab8/09-mospf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 64 hops max
 1  10.0.1.1  0.168ms  0.035ms  0.034ms
 2  10.0.2.2  0.048ms  0.161ms  0.068ms
 3  10.0.4.4  0.090ms  0.058ms  0.299ms
 4  10.0.6.22 0.347ms  0.192ms  0.136ms
```

关闭 r2 与 r4 的链路，再在 h1 主机上 traceroute h2 主机，结果如下：

```

4 10.0.6.22 0.347ms 0.132ms 0.130ms
root@Computer:~/workspace/Network/lab8/09-mospf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 64 hops max
 1 10.0.1.1 0.171ms 0.017ms 0.048ms
 2 10.0.3.3 0.464ms 0.364ms 0.259ms
 3 10.0.5.4 2.375ms 1.130ms 1.534ms
 4 10.0.6.22 1.881ms 1.130ms 1.896ms

```

路由表改变，路径正确。因此路由表生成算法正确，能在链路状态改变后正确构建新的路由表。

## 六、实验总结

本次实验中，我实现了路由器从构建并处理 mOSPF 信息到维护一致性链路状态数据库，再到映射为邻接矩阵并通过计算最短路径来生成路由表的功能。通过本次实验，我了解了 mOSPF 协议内容，掌握了如何使用 Dijkstra 算法计算最短路径以及生成路由表，这让我对理论课上讲过的 OSPF 路由协议有了更深的理解。