

实验报告

网络路由实验

一、实验内容

了解 mOSPF 路由机制和协议格式，掌握如何处理 Hello、LSU 消息和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。

二、实验流程

1. 实现处理 mOSPF Hello/LSU 消息的相关操作，构建一致性链路状态数据库。
2. 实现路由器计算最短路径并根据最短路径生成路由表。
3. 验证构建一致性链路状态数据库、生成路由表的正确性。

三、实验结果及分析

（一）构建一致性链路状态数据库

1、处理 mOSPF Hello 消息

```
void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)
```

函数，负责处理接收到的 hello 消息。

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 mospf_send_lsu_packet() 函数洪泛链路状态。

最后 Hello 消息不需要转发。

```
void *sending_mospf_hello_thread(void *param)
```

线程函数，负责定时向周围发送 hello 消息

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。

```
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) {
        mospf_send_hello_packet(iface);
    }

    pthread_mutex_unlock(&mospf_lock);
}
```

```
void mospf_send_hello_packet(iface_info_t *iface)
```

函数，负责实际向端口发送 hello 消息

Hello 消息采用组播扩散，目的 IP 地址为 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。依次填充各段首部、数据，然后发送即可。

```
void *checking_nbr_thread(void *param)
```

线程函数，负责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，如果列表中的节点在 3*hello-interval 时间内未更新，则将其删除。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 mospf_send_lsu_packet()函数洪泛链路状态。

2、处理 mOSPF LSU 消息

```
void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)
```

函数，负责处理接收到的 lsu 消息。

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。

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

接着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 (saved=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)) {
        saved = 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;
            free(db_entry->array);
            db_entry->array = (struct mospf_lsa *)malloc(MOSPF_LSA_SIZE * db_entry->nadv);
            memcpy(db_entry->array, (char*)mospf_ls + MOSPF_LSU_SIZE, MOSPF_LSA_SIZE * db_entry->nadv);
            update = 1;
        }
    }
}

```

查询完数据库，检查 saved 标记，如果没有保存该条目，将其添加到数据库。

```

if (saved == 0) {
    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);
    memcpy(db_entry->array, (char*)mospf_ls + MOSPF_LSU_SIZE, MOSPF_LSA_SIZE * db_entry->nadv);
    init_list_head(&db_entry->list);
    list_add_tail(&db_entry->list, &mospf_db);
}

```

最后 LSU 消息可能需要转发，如果之前没有保存条目，或者条目更新，需要转发消息。检查其 ttl，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 (saved == 0 || update == 1) {
    mospf_ls->ttl = mospf_ls->ttl - 1;
    if (mospf_ls->ttl > 0) { ...
        build_route_table();
    }
}

```

最后如果数据库发生了更新，需要调用 build_route_table() 函数生成最新的路由表。

```
void *sending_mospf_lsu_thread(void *param)
```

线程函数，负责定时向邻居发送 lsu 消息。

生成并发送 lsu 消息采用单播发送，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。发送完后打印一下当前数据库内容。

```

while (1) {
    sleep(MOSPF_DEFAULT_LSUINT);
    pthread_mutex_lock(&mospf_lock);

    mospf_send_lsu_packet();

    print_database();

    pthread_mutex_unlock(&mospf_lock);
}

```

void mospf_send_lsu_packet(void)

函数，负责实际向所有邻居发送 hello 消息

首先根据邻居列表构建 mospf 报文，对于没有邻居节点的端口，也要占一个邻居位置，其 rid=0。

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 = (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, 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);
        }
    }
}

```

void *checking_database_thread(void *param)

线程函数，负责检查数据库每个条目的存活时间，将超时没有更新的条目删除。

设定为每秒执行一次，如果数据库中的条目在 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，说明数据库发生改变，因此需要调用 build_route_table()函数生成最新的路由表。

3、数据库结果验证

运行网络拓扑和路由器，一段时间后查看其链路状态数据库。

R1 数据库:

"Node: r1"			
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
10.0.4.4	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 数据库:

"Node: r2"			
MOSPF Database:			
RID	Network	Mask	Neighbor

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
10.0.4.4	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
10.0.3.3	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 数据库:

"Node: r3"			
MOSPF Database:			
RID	Network	Mask	Neighbor

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
10.0.4.4	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

R4 数据库:

"Node: r4"			
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
10.0.1.1	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

4 个节点的链路状态数据库一致且正确。

（二）生成路由表

1、计算最短路径并生成路由表

`void build_route_table(void)`

函数，计算并生成路由表

具体流程为：

```
void build_route_table(void)
{
    clear_route_table();
    get_node_mapped();

    init_graph();
    Dijkstra();

    gen_route_table();
    print_rtable();

    return;
}
```

`void clear_route_table(void)`

函数，负责清除旧路由表，保留内核中读取的默认路由项（gw=0），清除其他项。

```
void get_node_mapped(void)
```

函数，负责构建节点 rid 到图中 index 的映射。

将自己设为 0 号节点，将邻居列表和数据库中各节点依次编号，包括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。

```
void init_graph(void)
```

函数，负责初始化图，图采用邻接矩阵 graph 表示。

将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。

注意 INT_MAX 不要设为实际 int 的最大值，因为之后会有 $\text{dist}[i] + \text{graph}[i][j]$ 的计算，这两项默认值都为 INT_MAX，需要保证其和不会超过 int 最大值。

为了 Dijkstra 时计算正确，注意保证邻接矩阵是对称的。

```
void Dijkstra(void)
```

函数，负责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。

```
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();
    if (u == -1) {
        log(ERROR, "all node visited, stopped at: %d\n", i);
        break;
    }
    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;
        }
    }
}
}
```

```
void gen_route_table(void)
```

函数，负责根据最短路径，生成路由表。

按路径长度从小到大遍历节点，从 stack 中依次取出即可。stack[0]是该路由器本身，我们保留了其到邻居节点的路由表项，因此不用对其进行计算，从 stack[1]开始遍历。

对每个节点，首先尝试在数据库里找出其对应的条目。如果此时路由器刚启动，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，本路由器从邻居的 lsu 得知其存在，但其 lsu 还未到本路由器。这时跳过该节点即可。

```
node_now = stack[i];
find = 0;
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;
        find = 1;
    }
}

if (find == 0) {
    log(WARNING, "node database not find: %d "IP_FMT"\n", node_now, node_now);
    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，邻居的 lsu 比其 hello 先到达，本路由器从邻居的 lsu 知道邻居能到达自己，但邻居列表找不到该邻居节点。这种情况也跳过即可。

```
while (prev[node_now] != 0) {
    node_now = prev[node_now];
}
find = 0;
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;
                find = 1;
            }
        }
    }
}

if (find == 0) {
    log(WARNING, "nerghbor not find: %d "IP_FMT"\n", node_now, node_now);
    log(WARNING, "lsu came before hello\n");
    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。

对于 stack[i] 节点的连接的每个网络，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;
        }
    }

    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);
    }
}
```

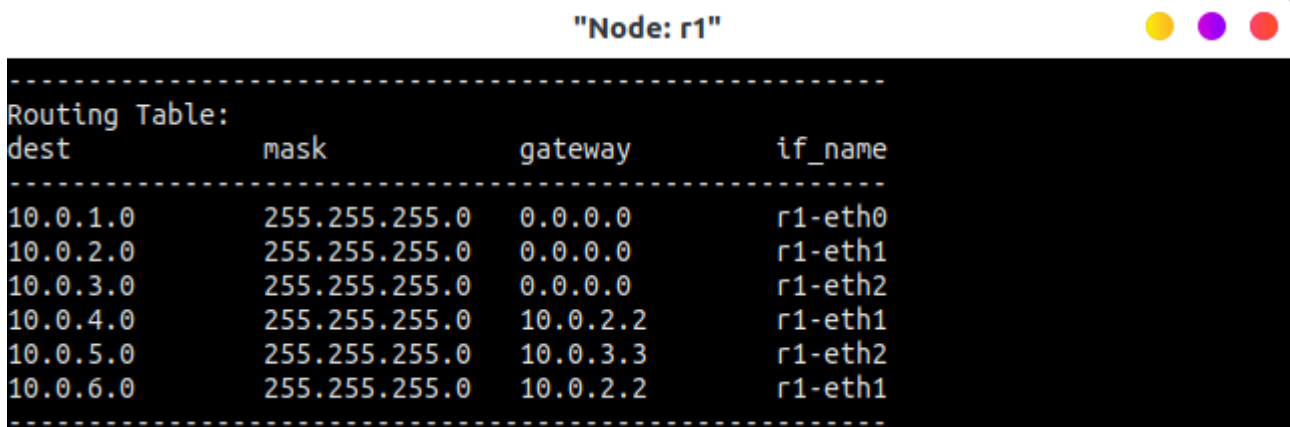
`void print_rtable()`

函数，负责打印路由表。

2、结果验证

运行网络拓扑和路由器，一段时间后查看其路由表。

R1 路由表：



The terminal window shows the routing table for node r1. The table has four columns: dest, mask, gateway, and if_name. The data rows are as follows:

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

各网络的路由表项存在且正确。

在 h1 主机上 ping h2 主机，可以 ping 通。

```
"Node: h1"
root@joker-linux:/mnt/hgfs/share/11-mopsf# 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=0.560 ms
64 bytes from 10.0.6.22: icmp_seq=2 ttl=61 time=0.485 ms
64 bytes from 10.0.6.22: icmp_seq=3 ttl=61 time=0.361 ms
64 bytes from 10.0.6.22: icmp_seq=4 ttl=61 time=0.351 ms

--- 10.0.6.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3068ms
rtt min/avg/max/mdev = 0.351/0.439/0.560/0.087 ms
```

在 h1 主机上 traceroute h2 主机，路径正确。

```
"Node: h1"
root@joker-linux:/mnt/hgfs/share/11-mopsf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.266 ms  0.209 ms  0.202 ms
 2  10.0.2.2 (10.0.2.2)  0.908 ms  0.936 ms  0.935 ms
 3  10.0.4.4 (10.0.4.4)  0.917 ms  0.913 ms  0.910 ms
 4  10.0.6.22 (10.0.6.22)  0.906 ms  0.966 ms  0.931 ms
```

关闭 r2 与 r4 的链路，再在 h1 主机上 traceroute h2 主机。

```
root@joker-linux:/mnt/hgfs/share/11-mopsf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.193 ms  0.157 ms  0.149 ms
 2  10.0.3.3 (10.0.3.3)  0.963 ms  0.977 ms  0.974 ms
 3  10.0.5.4 (10.0.5.4)  2.196 ms  2.196 ms  2.211 ms
 4  10.0.6.22 (10.0.6.22)  2.240 ms  2.237 ms  2.232 ms
```

路由表改变，路径正确。

因此路由表生成算法正确，且能在链路状态改变后正确构建新的路由表。

四、思考题

1. 在构建一致性链路状态数据库中，为什么邻居发现使用组播(Multicast)机制，链路状态扩散用单播(Unicast)机制？

在一般的网络中，路由器节点并不知道其各个端口上有没有邻居节点，更不知道其 IP 地址，没有办法使用单播，只能通过组播宣告自己的存在。

而在发送链路状态的 LSU 消息时，路由器已经知道自己有哪些邻居，也知道其 IP 地址，因此可以通过单播形式精准发送。单播形式可以避免对网络中其他设备的干扰，也避免其他设备获取路由信息，提高安全性。

2. 该实验的路由收敛时间大约为 20-30 秒，网络规模增大时收敛时间会进一步增加，如何改进路由算法的扩展性？

可以使用划分区域的方法，控制链路状态信息泛洪的范围，减小链路状态数据库大小，从而快速收敛。

基本思想是将网络分为不同区域，每个区域有不同的 area id，其中有一个 aid=0 的主干区域，其他为分支区域。主干区域与每个分支区域相连，而分支区域彼此不相连。链路状态信息的泛洪只在各区域内部进行，路由器不处理其他区域的链路状态信息，这样各区域可以分别快速收敛。

最后各分支区域从主干区域的路由器获取其他分支区域的路由信息，达到整个网络的收敛。

3. 路由查找的时间尺度为~ns，路由更新的时间尺度为~10s，如何设计路由查找更新数据结构，使得更新对查找的影响尽可能小？

目前的路由表更新方案为先清理旧路由表，再计算最短路径，最后生成新的路由表。耗时较长，这一段时间里路由查找都无法进行。

一个解决方案是，更新路由表时先维持旧路由表，使其不影响路由查找的使用。然后生成新路由表时使用一个临时的表来存放，生成完毕后，替换旧的路由表。这样路由更新的影响只有复制路由表的这段时间。

更进一步，我们可以用一个指针来指向当前使用的路由表，替换旧路由表时只需要更改一个指针，几乎不影响路由查找。

五、实验总结

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。