Report 9 — Novmeber 7

Lecturer: Wu Qinghua *Completed by: 2022K8009929010 Zhang Jiawei*

9.1 实验内容

1. 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；
   - 运行实验
     - 运行网络拓扑 (topo.py)
     - 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
     - 运行./mospfd，使得各个节点生成一致的链路状态数据库。
2. 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；
   - 运行实验
     - 运行网络拓扑 (topo.py)
     - 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
     - 运行./mospfd，使得各个节点生成一致的链路状态数据库
     - 等待一段时间后，每个节点生成完整的路由表项
     - 在节点 h1 上 ping/traceroute 节点 h2
     - 关掉某节点或链路，等一段时间后，再次用 h1 去 traceroute 节点 h2。

9.2 实验过程

9.2.1 链路邻居发现

对于每一个链路上的路由器节点，都会从各个端口以 5 秒为周期发送 Hello 消息，以便发现邻居节点。Hello 消息包括了发送节点的 ID、端口的子网掩码等等信息。需要注意一点，Hello 消息的目的 IP 地址是 224.0.0.5，目的 MAC 地址是 01:00:5e:00:00:05，这是因为 Hello 消息是多播消息，需要发送给多播组。再接上各层头部，即可发送。具体代码如下：

```c
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) {
            int mospf_len = MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE;
            int packet_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + mospf_len;
            char *packet = (char *)malloc(packet_len);
            struct ether_header *ethdr = (struct ether_header *)packet;
            struct iphdr *iphdr = packet_to_ip_hdr(packet);
            struct mospf_hdr *mospfhdr = (struct mospf_hdr *)((char *)iphdr +
                IP_BASE_HDR_SIZE);
            struct mospf_hello *hello = (struct mospf_hello *)((char *)mospfhdr +
                MOSPF_HDR_SIZE);

            memset(packet, 0, packet_len);

            mospf_init_hello(hello, iface->mask);
            mospf_init_hdr(mospfhdr, MOSPF_TYPE_HELLO, mospf_len,
                instance->router_id, instance->area_id);
            mospfhdr->checksum = mospf_checksum(mospfhdr);

            ip_init_hdr(iphdr, iface->ip, MOSPF_ALLSPFRouters, packet_len -
                ETHER_HDR_SIZE, IPPROTO_MOSPF);

            memcpy(ethdr->ether_dhost, "\x01\x00\x5e\x00\x00\x05", ETH_ALEN);
            memcpy(ethdr->ether_shost, iface->mac, ETH_ALEN);
            ethdr->ether_type = htons(ETH_P_IP);

            iface_send_packet(iface, packet, packet_len);
        }

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}
```

当某个路由器节点接收到邻居节点发来的 Hello 消息时，会根据 Hello 消息中的信息更新邻居节点的信息。若源节点在自身邻居列表中，则更新对应存活时间等等数据项；若源节点不在自身邻居列表中，则将其加入邻居列表，导致邻居列表变动，所以还需要发送 LSU 消息通知其他节点。具体代码如下：

```
void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)
{
   // fprintf(stdout, "TODO: handle mOSPF Hello message.\n");
   struct iphdr *iphdr = (struct iphdr *)(packet + ETHER_HDR_SIZE);
   struct mospf_hdr *mospfhdr = (struct mospf_hdr *)((char *)iphdr +
       IP_HDR_SIZE(iphdr));
   struct mospf_hello *hello = (struct mospf_hello *)((char *)mospfhdr +
       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(mospfhdr->rid)) {
         nbr->alive = 0;
         nbr->nbr_ip = ntohl(iphdr->saddr);
         nbr->nbr_mask = ntohl(hello->mask);

         pthread_mutex_unlock(&mospf_lock);
         return;
      }
   }

   nbr = (mospf_nbr_t *)malloc(sizeof(mospf_nbr_t));
   nbr->nbr_id = ntohl(mospfhdr->rid);
   nbr->nbr_ip = ntohl(iphdr->saddr);
   nbr->nbr_mask = ntohl(hello->mask);
   nbr->alive = 0;
   init_list_head(&nbr->list);
   list_add_tail(&nbr->list, &iface->nbr_list);
   iface->num_nbr++;

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

此外，邻居列表还需要一个老化操作，如果列表中的节点在 3*hello-interval 时间内未更新，则将其删除，也导致邻居列表变动，需要发送 LSU 消息通知其他节点。具体代码如下：

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

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

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

        if (delete)
            send_mospf_lsu_packet();

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}
```

9.2.2 链路状态洪泛

路由器节点每隔一段时间就会发送一次 LSU 消息，以便通知其他节点自身的链路状态。我将发送过程封装成一个函数，发送完成后打印节点数据库信息。具体代码如下：

```
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_database();
        pthread_mutex_unlock(&mospf_lock);
    }
    return NULL;
}
```

发送函数的具体实现如下，首先生成要发送的 LSA 部分，LSA 消息的个数为所有邻居节点个数加上无邻居节点的接口个数，内容包括网络地址、子网掩码、邻居节点 ID（无邻居则为 0）。然后生成 LSU 消息，对于各个端口，复制其相应的 LSA 消息，添加头部信息，向邻居节点发送。具体代码如下：

```
void send_mospf_lsu_packet()
{
    int lsa_num = 0;
    iface_info_t *iface = NULL;
    list_for_each_entry(iface, &instance->iface_list, list) {
        lsa_num += iface->num_nbr ? iface->num_nbr : 1;
    }

    int mospf_len = MOSPF_HDR_SIZE + MOSPF_LSU_SIZE + lsa_num * MOSPF_LSA_SIZE;
    char *packet = (char *)malloc(mospf_len);
    memset(packet, 0, mospf_len);
    struct mospf_hdr *mospfhdr = (struct mospf_hdr *)packet;
    struct mospf_lsu *lsu = (struct mospf_lsu *)((char *)mospfhdr +
        MOSPF_HDR_SIZE);
    struct mospf_lsa *lsa = (struct mospf_lsa *)((char *)lsu + MOSPF_LSU_SIZE);

    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) {
                lsa->network = iface->ip & iface->mask;
                lsa->mask = iface->mask;
                lsa->rid = nbr->nbr_id;
                lsa++;
            }
        }
        else {
            lsa->network = iface->ip & iface->mask;
            lsa->mask = iface->mask;
            lsa->rid = 0;
```

```
            lsa++;
        }
    }

    mospf_init_lsu(lsu, lsa_num);
    instance->sequence_num++;

    mospf_init_hdr(mospfhdr, MOSPF_TYPE_LSU, mospf_len, instance->router_id,
        instance->area_id);
    mospfhdr->checksum = mospf_checksum(mospfhdr);

    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_send = (char *)malloc(ETHER_HDR_SIZE + IP_BASE_HDR_SIZE
                    + mospf_len);
                struct ether_header *ethdr = (struct ether_header *)packet_send;
                struct iphdr *iphdr = (struct iphdr *)(packet_send + ETHER_HDR_SIZE);
                char *mospf_message = packet_send + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE;

                memset(packet_send, 0, ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + mospf_len);
                memcpy(mospf_message, packet, mospf_len);
                ip_init_hdr(iphdr, iface->ip, nbr->nbr_ip, mospf_len +
                    IP_BASE_HDR_SIZE, IPPROTO_MOSPF);
                memcpy(ethdr->ether_shost, iface->mac, ETH_ALEN);
                ethdr->ether_type = htons(ETH_P_IP);

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

当某个路由器节点接收到 LSU 消息时，先检查是否是自身发送的，若是则丢弃。若不是，则根据 LSA 消息更新链路状态数据库，如果数据库内有相同 rid 的条目且新序列号大于数据库内的序列号，则更新，如果数据库内的序列号较大，则丢弃；若数据库内无相同 rid 的条目，则加入数据库。对于洪泛操作，LSU 消息也会转发到邻居节点，如果链路状态得到更新，则检查 LSU 消息的 ttl，若 ttl 大于 0，则继续转发给邻居节点并更新路由表。具体代码如下：

```c
void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)
{
   // fprintf(stdout, "TODO: handle mOSPF LSU message.\n");
   struct ether_header *ethdr = (struct ether_header *)packet;
   struct iphdr *iphdr = (struct iphdr *)(packet + ETHER_HDR_SIZE);
   struct mospf_hdr *mospfhdr = (struct mospf_hdr *)((char *)iphdr +
       IP_HDR_SIZE(iphdr));
   struct mospf_lsu *lsu = (struct mospf_lsu *)((char *)mospfhdr +
       MOSPF_HDR_SIZE);
   struct mospf_lsa *lsa = (struct mospf_lsa *)((char *)lsu + MOSPF_LSU_SIZE);

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

   int update = 0;
   int exist = 0;
   mospf_db_entry_t *db = NULL;
   list_for_each_entry(db, &mospf_db, list) {
      if (db->rid == ntohl(mospfhdr->rid)) {
         exist = 1;
         if (db->seq < ntohs(lsu->seq)) {
            db->seq = ntohs(lsu->seq);
            db->alive = 0;
            db->nadv = ntohl(lsu->nadv);
            for (int i=0; i<db->nadv; i++) {
               db->array[i].network = lsa[i].network;
               db->array[i].mask = lsa[i].mask;
               db->array[i].rid = lsa[i].rid;
            }
            update = 1;
         }
      }
   }

   if (!exist) {
      db = (mospf_db_entry_t *)malloc(sizeof(mospf_db_entry_t));
      db->rid = ntohl(mospfhdr->rid);
      db->seq = ntohs(lsu->seq);
      db->alive = 0;
      db->nadv = ntohl(lsu->nadv);
```

```
        db->array = (struct mospf_lsa *)malloc(MOSPF_LSA_SIZE * db->nadv);
        for (int i=0; i<db->nadv; i++) {
            db->array[i].network = lsa[i].network;
            db->array[i].mask = lsa[i].mask;
            db->array[i].rid = lsa[i].rid;
        }
        init_list_head(&db->list);
        list_add_tail(&db->list, &mospf_db);
        update = 1;
    }

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

    lsu->ttl--;
    if (lsu->ttl > 0) {
        mospfhdr->checksum = mospf_checksum(mospfhdr);
        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_send = (char *)malloc(len);
                    struct ether_header *ethdr_send = (struct ether_header
                        *)packet_send;
                    struct iphdr *iphdr_send = (struct iphdr *)(packet_send +
                        ETHER_HDR_SIZE);
                    memcpy(packet_send, packet, len);

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

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

                    iface_send_packet_by_arp(iface_out, nbr->nbr_ip, packet_send, len);
                }
            }
        }
    }
```

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

同样也需要处理节点失效问题，当数据库中一个节点的链路状态超过 40 秒未更新时，表明该节点已失效，将对应条目删除，再更新路由表。具体代码如下：

```c
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 delete = 0;
        mospf_db_entry_t *db = NULL;
        mospf_db_entry_t *db_q = NULL;
        list_for_each_entry_safe(db, db_q, &mospf_db, list) {
            if (db->alive > MOSPF_DATABASE_TIMEOUT){
                delete = 1;
                list_delete_entry(&db->list);
                free(db->array);
                free(db);
            }
            else
                db->alive++;
        }

        if (delete)
            update_rtable();

        pthread_mutex_unlock(&mospf_lock);
    }

    return NULL;
}
```

9.2.3 路由表更新计算

路由表更新计算在本实验中是最麻烦的一个步骤。我通过邻接矩阵记录图的拓扑结构，通过 Dijkstra 算法计算最短路径，再根据最短路径更新路由表。具体来说，先清除路由表，但是保留默认路由表项，然后根据节点的链路状态数据库，初始化邻接矩阵，再使用 Dijkstra 算法计算最短路径，最后根据最短路径更新路由表。具体

代码如下:

```
void update_rtable(void)
{
   // clear the routing table, but keep the default entry whose gw is 0
   rt_entry_t *rt_entry, *rt_q;
   list_for_each_entry_safe(rt_entry, rt_q, &rtable, list) {
      if (rt_entry->gw)
         remove_rt_entry(rt_entry);
   }

   // number all the nodes
   // 0 is the router itself
   // 1 ~ n are the other nodes
   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_exist(nbr->nbr_id))
               node_map[node_num++] = nbr->nbr_id;
         }
      }
   }

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

   // initialize the graph
   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] = 0x1fffffff; // seen as the maximum value
      }
```

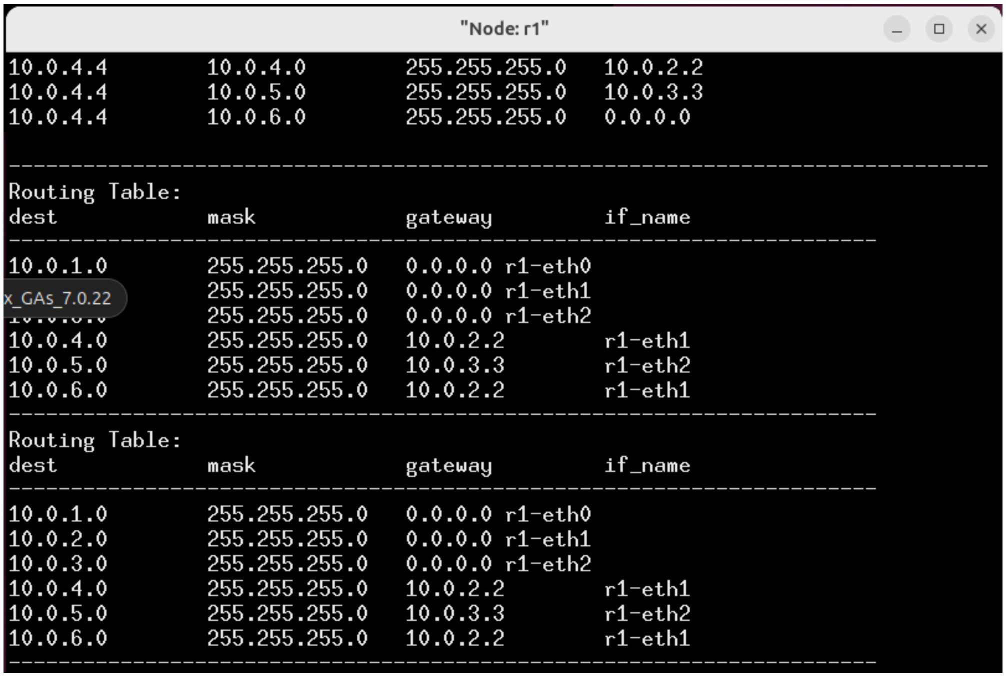
```
    }

    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][get_node_index(nbr->nbr_id)] = 1;
                graph[get_node_index(nbr->nbr_id)][0] = 1;
            }
        }
    }

    db = NULL;
    list_for_each_entry(db, &mospf_db, list) {
        int src = get_node_index(db->rid);
        for (int i=0; i<db->nadv; i++) {
            if (db->array[i].rid) {
                int dst = get_node_index(db->array[i].rid);
                graph[src][dst] = 1;
                graph[dst][src] = 1;
            }
        }
    }

    // Dijkstra algorithm
    int dist[MAX_NODE_NUM];
    int visited[MAX_NODE_NUM];
    for (int i=0; i<node_num; i++) {
        dist[i] = 0x1fffffff;
        visited[i] = 0;
        prev[i] = -1;
    }
    dist[0] = 0;
    stack_top = 0;

    for (int i=0; i<node_num; i++) {
        int min = 0x1fffffff;
        int u = -1;
        for (int j=0; j<node_num; j++) {
            if (!visited[j] && dist[j] < min) {
                min = dist[j];
                u = j;
```

```
            }
        }

        if (u == -1)
            break;

        visited[u] = 1;
        stack[stack_top++] = u;

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

    // update the routing table
    // use the prev array to find the next hop
    db = NULL;
    iface_info_t *iface_out = NULL;
    int current_node = 0;
    int find = 0;
    u32 gw;
    for (int i = 1; i < stack_top; i++) {
        current_node = stack[i];
        mospf_db_entry_t *db_tmp = NULL;
        list_for_each_entry(db_tmp, &mospf_db, list) {
            if (db_tmp->rid == node_map[current_node]) {
                db = db_tmp;
                break;
            }
        }

        if (db == NULL)
            continue;

        while (prev[current_node] != 0)
            current_node = prev[current_node];

        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[current_node]) {
                        iface_out = iface;
                        gw = nbr->nbr_ip;
                        break;
                    }
                }
            }
        }

        if (iface_out == NULL)
            continue;

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

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

    print_rtable();

    return;
}
```

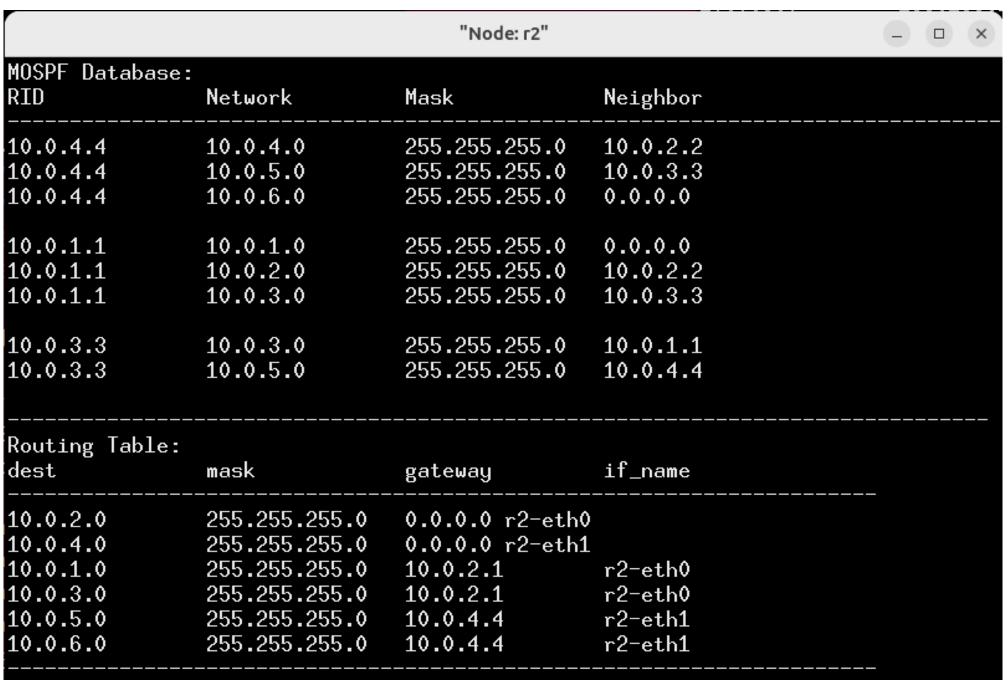
9.3 实验结果

在每个路由器节点上运行./mospfd，使得各个节点生成一致的链路状态数据库，并查看四个节点的路由表，如下：



```
"Node: r1"
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

--------------------------------------------------------------------------------
Routing Table:
dest            mask            gateway         if_name
------------------------------------------------------------------------
10.0.1.0        255.255.255.0   0.0.0.0 r1-eth0
x_GAs_7.0.22    255.255.255.0   0.0.0.0 r1-eth1
                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
------------------------------------------------------------------------
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
------------------------------------------------------------------------
```

图 **9.1.** r1 的数据库和路由表



```
"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

--------------------------------------------------------------------------------
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
------------------------------------------------------------------------
```

图 **9.2.** r2 的数据库和路由表

```
"Node: r3"
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

10.0.2.2        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

-------------------------------------------------------------------------------
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
----------------------------------------------------------------------
```

图 **9.3.** r3 的数据库和路由表

```
"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

-------------------------------------------------------------------------------
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
----------------------------------------------------------------------
```

图 **9.4.** r4 的数据库和路由表

可以看出，四个节点的数据库一致，路由表项均存在且正确。

在节点 h1 上 ping 节点 h2，如下：

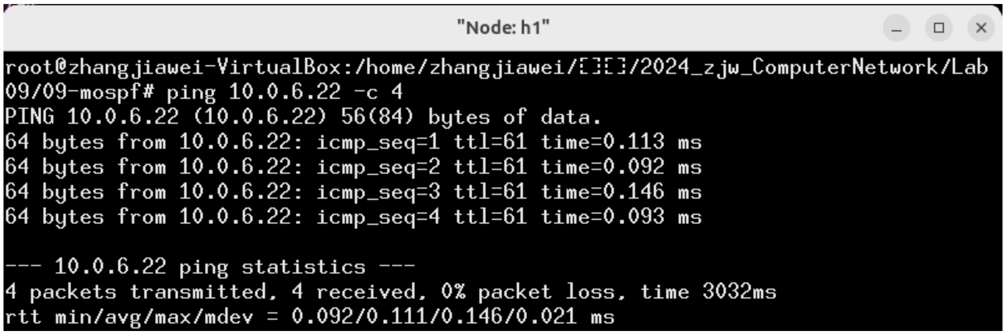


图 **9.5.** h1 ping h2

在节点 h1 上 traceroute 节点 h2，如下：

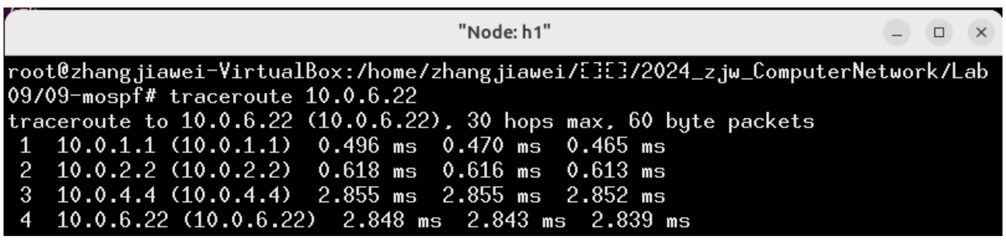


图 **9.6.** h1 traceroute h2

关闭 r1 和 r2 之间的链路，等待一段时间后，再次用 h1 去 traceroute 节点 h2，如下：

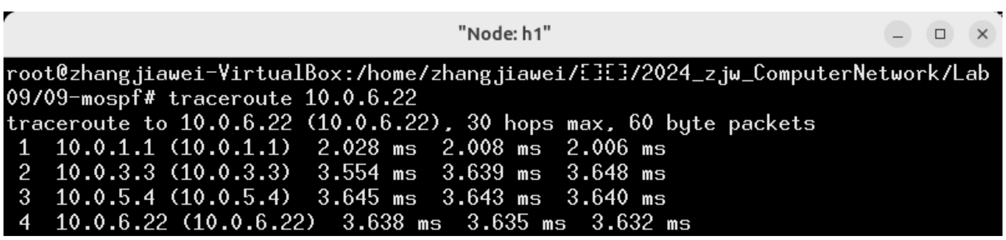


图 **9.7.** h1 traceroute h2 after link down

路径发生了变化，说明路由表更新成功，算法正确。

9.4 实验总结

本次实验主要是实现 mOSPF 协议的 Hello/LSU 消息的生成和处理，以及路由表的更新计算。实验中需要注意的是，链路状态数据库的一致性，需要在每个节点上都运行 mospfd，以便生成一致的数据库；路由表的更新计算，需要使用 Dijkstra 算法计算最短路径，再根据最短路径更新路由表。实验中还需要注意处理节点失效

问题，当链路状态超过 40 秒未更新时，表明该节点已失效，需要删除对应条目，再更新路由表。我学会了如何实现 mOSPF 协议的 Hello/LSU 消息的生成和处理，以及路由表的更新计算，对于理解路由器的工作原理有很大帮助。