

# 网络路由实验报告

2020K8009906008 周富杰

## 一、实验内容

1. 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，并进行验证。
2. 运行实验，测试连通性，关闭某一个节点后，等待一定时间，检查网络功能是否正常。

## 二、实验流程

1. 从路由器转发实验复制所需的 c 文件
2. 阅读代码（果然给我发现了，课程给出的代码缺少一个函数，功能不全）
3. 完善 mospf\_daemon.c 文件
  - void \*sending\_mospf\_hello\_thread(void \*param)  
这个函数是用于定期发送 hello 包的线程函数，思路比较直白，每隔 5s，按照相应格式填充 hello 包，然后从每个端口发送出去即可。注意对于 mospf 头，以及 hello, lsu 等内容都有专门的初始化函数，可直接使用。

```
iface_info_t *iface;
while (1)
{
    sleep(MOSPF_DEFAULT_HELLOINT);

    pthread_mutex_lock(&mospf_lock);

    iface = NULL;
    list_for_each_entry(iface, &instance->iface_list, list){
        int len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE;
        char *p = (char*)malloc(len);
        struct ether_header *eh = (struct ether_header *)p;
        struct iphdr *ih = packet_to_ip_hdr(p);
        struct mospf_hdr *mh = (struct mospf_hdr *)((char *)ih+IP_BASE_HDR_SIZE);
        struct mospf_hello *hello = (struct mospf_hello *)((char *)mh+MOSPF_HDR_SIZE);

        u8 dhost[ETH_ALEN] = {0x01,0x00,0x5e,0x00,0x00,0x05};
        memcpy(eh->ether_dhost,dhost,ETH_ALEN);
        eh->ether_type = htons(ETH_P_IP);
        memcpy(eh->ether_shost,iface->mac,ETH_ALEN);
        ip_init_hdr(ih,iface->ip,MOSPF_ALLSPFRouters,len-ETHER_HDR_SIZE,IPPROTO_MOSPF);
        mospf_init_hdr(mh,MOSPF_TYPE_HELLO,MOSPF_HDR_SIZE+MOSPF_HELLO_SIZE,instance->router_id,0);
        mospf_init_hello(hello,iface->mask);
        mh->checksum = mospf_checksum(mh);
        //fprintf(stdout, "SEND HELLO.\n");
        iface_send_packet(iface,p,len);
    }
    pthread_mutex_unlock(&mospf_lock);
}
```

- void \*checking\_nbr\_thread(void \*param)  
该函数用于老化邻居列表。遍历邻居列表，如果列表中的节点在 3\*hello-interval 时间内未更新，则将其删除

```

while (1)
{
    sleep(1);
    pthread_mutex_lock(&mospf_lock);
    iface_info_t *iface;
    list_for_each_entry(iface, &instance->iface_list, list){
        mospf_nbr_t *nbr;
        list_for_each_entry(nbr, &iface->nbr_list, list){
            if(++nbr->alive > 3 * MOSPF_DEFAULT_HELLOINT){
                list_delete_entry(&nbr->list);
                free(nbr);
                iface->num_nbr--;
                send_lsu();
            }
        }
    }
    pthread_mutex_unlock(&mospf_lock);
}

```

- void \*checking\_database\_thread(void \*param)  
该函数用于处理节点失效问题，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

```

mospf_db_entry_t *db;
while(1){
    sleep(1);
    pthread_mutex_lock(&mospf_lock);
    list_for_each_entry(db, &mospf_db, list){
        if((++db->alive) > MOSPF_DATABASE_TIMEOUT){
            list_delete_entry(&db->list);
            free(db);
        }
    }
    // update rtable
    //shortest_rtable();
    pthread_mutex_unlock(&mospf_lock);
}

```

- void handle\_mospf\_hello(iface\_info\_t \*iface, const char \*packet, int len)  
该函数用于处理 hello 包。对传入的包进行解析，如果在邻居列表中已有，则更新 alive，如果没有，则初始化一个邻居项，根据包的内容填充相应数据加入邻居列表，注意此时表明自身拓扑结构发生变化，需要发送 lsu 包给其他节点。

```

struct iphdr * ih = packet_to_ip_hdr(packet);
struct mospf_hdr * mh = (struct mospf_hdr *)((char *)ih+IP_HDR_SIZE(ih));
struct mospf_hello *hello = (struct mospf_hello *)((char *)mh+MOSPF_HDR_SIZE);
pthread_mutex_lock(&mospf_lock);
mospf_nbr_t *nbr;
int match = 0;
list_for_each_entry(nbr,&iface->nbr_list,list){
    if(nbr->nbr_ip == ntohl(ih->saddr)){
        match = 1;
        nbr->alive = 0;
        break;
    }
}
if(match == 0){
    nbr = malloc(sizeof(mospf_nbr_t));
    nbr->alive = 0;
    nbr->nbr_id = ntohl(mh->rid);
    nbr->nbr_ip = ntohl(ih->saddr);
    nbr->nbr_mask = ntohl(hello->mask);
    list_add_tail(&nbr->list,&iface->nbr_list);
    iface->num_nbr++;
    send_lsu();
}
pthread_mutex_unlock(&mospf_lock);

```

- void \*sending\_mospf\_lsu\_thread(void \*param)  
该函数用于周期性发送 lsu 数据包，发送数据包功能由 send\_lsu()实现，这里是周期性调用该函数。

```

while (1)
{
    sleep(MOSPF_DEFAULT_LSUINT);
    pthread_mutex_lock(&mospf_lock);
    send_lsu();
    pthread_mutex_unlock(&mospf_lock);
}

```

- void send\_lsu()  
该函数用于发送 lsu 包。注意对于邻居列表发生变化时以及超过 lsu interval (30 秒)未发送过链路状态信息时，都需要发送 lsu 包。lsu 数据包除了相应的报头以外，还有自身的内容，也就是 lsa，报告自己的邻居信息，由于邻居数量不定，所以这个长度是动态变化的，需要计算。函数第一步就是计算邻居数量，对于邻居为 0，也就是连接主机的也要算为 1 个，其余正常计算，最后计算出包含以太网头、IP 头、mospf 头、lsu 头、邻居信息的数据包的长度。然后分配空间，填充 lsa 信息。遍历所有接口的邻居列表，对于每个邻居创建一个新的数据包，填充以太网头部、IP 头部、mospf 头部和 LSU 消息。将数据包通过邻居对应的接口发送出去。最后更新本节点的 sequence number。

```

void send_lsu(){
    int nbr_num = 0;
    iface_info_t *iface;
    list_for_each_entry (iface, &instance->iface_list, list) {
        if (!iface->num_nbr) {
            nbr_num++;
        } else {
            nbr_num += iface->num_nbr;
        }
    }
    int len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_SIZE + MOSPF_LSU_SIZE + nbr_num*MOSPF_LSA_SIZE;

    struct mospf_lsa *lsa = (struct mospf_lsa *)malloc(nbr_num * MOSPF_LSA_SIZE);

    // fill lsa
    int pos = 0;
    mospf_nbr_t *nbr;
    list_for_each_entry(iface, &instance->iface_list, list) {
        if (!iface->num_nbr) {
            lsa[pos].network = htonl(iface->ip & iface->mask);
            lsa[pos].mask = htonl(iface->mask);
            lsa[pos].rid = 0;
            pos++;
        } else {
            list_for_each_entry(nbr, &iface->nbr_list, list) {
                lsa[pos].network = htonl(nbr->nbr_mask & nbr->nbr_ip);
                lsa[pos].mask = htonl(nbr->nbr_mask);
                lsa[pos].rid = htonl(nbr->nbr_id);
                pos++;
            }
        }
    }
}

```

```

//all
list_for_each_entry(iface, &instance->iface_list, list){
    list_for_each_entry(nbr, &iface->nbr_list, list){
        char *p = (char*)malloc(len);
        struct ether_header *eh = (struct ether_header *)p;
        struct iphdr *ih = packet_to_ip_hdr(p);
        struct mospf_hdr *mh = (struct mospf_hdr *)((char *)ih+IP_BASE_HDR_SIZE);
        struct mospf_lsu *lsu = (struct mospf_lsu *)((char *)mh+MOSPF_HDR_SIZE);
        struct mospf_lsa *lsa1 = (struct mospf_lsa *)((char *)lsu+MOSPF_LSU_SIZE);

        memcpy(lsa1,lsa,nbr_num * MOSPF_LSA_SIZE);
        eh->ether_type = htons(ETH_P_IP);
        memset(eh->ether_dhost,0x00,ETH_ALEN);
        memcpy(eh->ether_shost,iface->mac,ETH_ALEN);
        ip_init_hdr(ih, iface->ip, nbr->nbr_ip, len - ETHER_HDR_SIZE, IPPROTO_MOSPF);
        mospf_init_hdr(mh, MOSPF_TYPE_LSU, len - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE, instance->router_id, instance->area_id);
        mospf_init_lsu(lsu, nbr_num);
        mh->checksum = mospf_checksum(mh);
        //fprintf(stdout, "SEND LSU\n");
        ip_send_packet(p,len);
    }
}
instance->sequence_num++;

```

- void handle\_mospf\_lsu(iface\_info\_t \*iface, char \*packet, int len)**  
 该函数用于处理 lsu 包。首先就是解析得到 LSU 消息头部和内容，并遍历当前节点的数据库表查找是否有与该消息路由 ID 相同的项。如果找到了，则比较新接收到的 LSU 消息序列号和已有记录中的序列号大小，如果新消息序列号更大则更新对应的记录；否则直接丢弃该消息。如果没找到，则创建一个新的记录并添加到数据库表中。自身处理完后，要根据寿命判断是否转发数据包。也就是 TTL 减 1，如果 TTL 值大于 0，则向除该端口以外的端口转发该消息。

```

pthread_mutex_lock(&mospf_lock);
struct iphdr * ih = packet_to_ip_hdr(packet);
struct mospf_hdr * mh = (struct mospf_hdr *)((char *)ih+IP_HDR_SIZE(ih));
struct mospf_lsu * lsu = (struct mospf_lsu *)((char *)mh+MOSPF_HDR_SIZE);
struct mospf_lsa * lsa = (struct mospf_lsa *)((char *)lsu+MOSPF_LSU_SIZE);
//update db
mospf_db_entry_t *db;
int match = 0;
list_for_each_entry(db,&mospf_db,list){
    if(db->rid == ntohl(mh->rid)){
        match = 1;
        if(db->seq < ntohs(lsu->seq)){ // update
            db->alive = 0;
            for (int i = 0; i < db->nadv; i++)
            {
                db->array[i].mask = ntohl(lsa[i].mask);
                db->array[i].network = ntohl(lsa[i].network);
                db->array[i].rid = ntohl(lsa[i].rid);
            }
            db->nadv = ntohl(lsu->nadv);
            db->seq = ntohs(lsu->seq);
        }
    }
}
}

```

```

// new id
if(!match){
    db = malloc(sizeof(mospf_db_entry_t));
    db->alive = 0;
    db->array = (struct mospf_las*)malloc(ntohl(lsu->nadv)*MOSPF_LSA_SIZE);
    for (int i = 0; i < db->nadv; i++)
    {
        db->array[i].mask = ntohl(lsa[i].mask);
        db->array[i].network = ntohl(lsa[i].network);
        db->array[i].rid = ntohl(lsa[i].rid);
    }
    db->nadv = ntohl(lsu->nadv);
    db->rid = ntohl(mh->rid);
    db->seq = ntohs(lsu->seq);
    list_add_tail(&db->list,&mospf_db);
}
pthread_mutex_unlock(&mospf_lock);
//send if alive
if(-- lsu->ttl > 0){
    iface_info_t *ifa;
    list_for_each_entry(ifa, &instance->iface_list, list){
        if(ifa->num_nbr > 0 && ifa->index != ifa->index){
            mospf_nbr_t *nbr;
            list_for_each_entry(nbr, &ifa->nbr_list, list){
                if(nbr->nbr_id != ntohl(mh->rid)){
                    char *p = (char *)malloc(Len);
                    memcpy(p,packet,Len);
                    struct ether_header *eh = (struct ether_header *)p;
                    struct iphdr * ih1 = packet_to_ip_hdr(p);
                    struct mospf_hdr * mh1 = (struct mospf_hdr *)((char *)ih1+IP_HDR_SIZE(ih1));
                    memcpy(eh->ether_shost,ifa->mac,ETH_ALEN);
                    ih1->saddr = htonl(ifa->ip);
                    ih1->daddr = htonl(nbr->nbr_ip);
                    ih1->checksum = ip_checksum(ih1);
                    mh1->checksum = mospf_checksum(mh1);
                    ip_send_packet(p,Len);
                }
            }
        }
    }
}
}

```

- void dijkstra(int prev[NUM\_ROUTER], int dist[NUM\_ROUTER])  
 该函数就是 ppt 里描述的 dijkstra 算法的 C 语言描述，实现思路基本一样。其中 graph 数组为节点临界矩阵，known\_num 为当前路由已知路由数量（拓扑节点数量），均为全局变量。



使用Dijkstra算法计算源节点到其它节点的最短路径和相应前一跳节点


```
for i in range(num):
    dist[i] = INT_MAX
    visited[i] = false
    prev[i] = -1

dist[0] = 0

for i in range(num):
    u = min_dist(dist, visited, num)
    visited[u] = true

    for v in range(num):
        if visited[v] == false && graph[u][v] > 0 && \
            dist[u] + graph[u][v] < dist[v]:
            dist[v] = dist[u] + graph[u][v]
            prev[v] = u
```

在未访问的节点中，选取离已访问节点最近的那个



```
void dijkstra(int prev[NUM_ROUTER], int dist[NUM_ROUTER])
{
    int visit[NUM_ROUTER];
    for (int i = 0; i < NUM_ROUTER; i++) {
        dist[i] = INT8_MAX;
        visit[i] = 0;
        prev[i] = -1;
    }

    dist[0] = 0;

    for (int i = 0; i < known_num; i++) {
        int j = -1;
        for (int m = 0; m < known_num; m++) {
            if(visit[m] == 0) {
                if(j == -1 || dist[m] < dist[j])
                    j = m;
            }
        }
        int u = j;
        visit[u] = 1;
        for(int v = 0; v < known_num; v++) {
            if(!visit[v] && graph[u][v]>0 &&dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                prev[v]= u;
            }
        }
    }
}
```

■ void shortest\_rtable()

这个函数用于构建当前节点的最短路径路由表。我在另外一个部分使用一个线

程 while 循环调用该函数来实现更新的功能。首先是初始化邻接表，也就是使用最大数表示无穷远来填充邻接表。使用 router 数组来记录邻接表每一项对应的路由 id。然后遍历数据库，先根据 id 找对应邻接表的那一行列，然后检查其邻居，在 graph 中的对应位置设置为 1。然后，使用 Dijkstra 算法计算出从当前节点到其他每个节点的最短路径，得到 prev 和 dist 数组。然后为了不和结构冲突，这里选择删除原有的路由表中所有与网关有关的表项。然后从最短路径开始，逐步访问剩余节点。对于每个已经访问过的节点，遍历该节点的 LSA 信息，为其中未添加的目的网络添加新的路由表项。每次添加新的路由表项时，需要根据该目的网络的最短路径上的下一个节点在邻居列表中查找对应的网关 IP，并创建一个新的路由表项添加到路由表中。

```
memset(graph,INT8_MAX,sizeof(graph));
known_num = 0;
router[known_num++] = instance->router_id;
// record id
mospf_db_entry_t *db;
list_for_each_entry(db,&mospf_db,list){
    router[known_num++] = db->rid;
}
// finish graph
list_for_each_entry(db,&mospf_db,list){
    int m,n;
    for (int i = 0; i < known_num; i++)
    {
        if(router[i] == db->rid){
            m = i;
            break;
        }
    }
    for (int i = 0; i < db->nadv; i++)
    {
        if(!db->array[i].rid) continue; // node
        for (int j = 0; j < known_num; j++)
        {
            if(router[j]== db->array[i].rid)
            {
                n = j;
                break;
            }
        }
        graph[m][n] = 1;
        graph[n][m] = 1;
    }
}
```



```

int prev[NUM_ROUTER];
int dist[NUM_ROUTER];
dijkstra(prev,dist);

//delete all entry
rt_entry_t *entry;
list_for_each_entry(entry,&rttable,list){
    if(entry->gw)
        remove_rt_entry(entry);
}

int visit[NUM_ROUTER] = {0};
visit[0] = 1;

```

```

// visit from shortest to biggest
for (int i = 0; i < known_num; i++)
{
    int m = -1;
    for (int j = 0; j < known_num; j++)
    {
        if(!visit[j]){
            if(m == -1){
                m = j;
            }else if(dist[j]<dist[m]){
                m = j;
            }
        }
    }
    visit[m] = 1;
    list_for_each_entry(db, &mospf_db, list)
    {
        if(router[m] != db->rid) continue;
        int next = m;
        while (prev[next] != 0)
        {
            next = prev[next];
        }
        int match =0;
        u32 gw;
        iface_info_t *iface;
        list_for_each_entry(iface, &instance->iface_list, list) {
            mospf_nbr_t *nbr;
            list_for_each_entry(nbr, &iface->nbr_list, list) {
                if(nbr->nbr_id == router[next]){
                    gw = nbr->nbr_ip;
                    match = 1;
                    break;
                }
            }
            if(match) break;
        }
    }
}

```

```

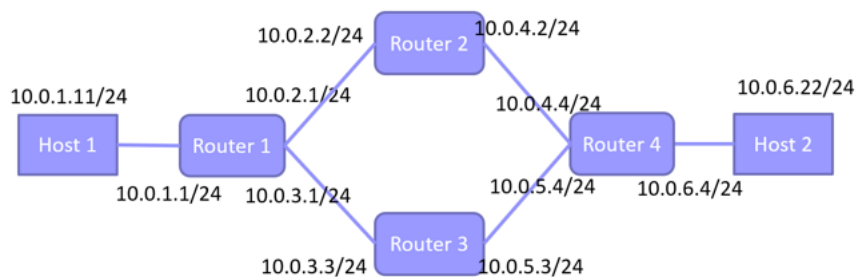
if(!match) break; //should do not have ?
for (int j = 0; j < db->nadv; j++)
{
    match = 0;
    list_for_each_entry(entry,&rtable,list){
        if(entry->dest == db->array[j].network && entry->mask == db->array[j].mask){
            match = 1;
            break;
        }
    }
    if(match) continue;
    rt_entry_t *new = new_rt_entry(db->array[j].network,db->array[j].mask,gw,iface);
    add_rt_entry(new);
}

```

4. make 编译，然后运行 topo.py 文件进行测试。

### 三、实验结果与分析

拓扑结构，与相应 ip 地址如下



在设计文件中添加一个循环，调用 print\_rtable 函数（已由课程代码实现），打印路由表信息，结果如下，符合预期

Terminal windows showing router configurations and routing tables:

- Node: r1**: Shows configuration for interfaces r1-eth0, r1-eth1, r1-eth2 and the resulting routing table with 6 entries.
- Node: r2**: Shows configuration for interfaces r2-eth0, r2-eth1 and the resulting routing table with 6 entries.
- Node: r3**: Shows configuration for interfaces r3-eth0, r3-eth1 and the resulting routing table with 6 entries.
- Node: r4**: Shows configuration for interfaces r4-eth0, r4-eth1, r4-eth2 and the resulting routing table with 6 entries.

在运行一段时间后，使用 h1 traceroute h2，然后关闭 r3 节点，等待一段时间后，再次

traceroute, 结果如下, 可以看到链路正常。

```
"Node: h1"
root@ucas-cod-2022:/home/ucas/08-mospf# 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)  0.201 ms  0.192 ms  0.189 ms
 2  10.0.2.2 (10.0.2.2)  0.328 ms  0.333 ms  0.336 ms
 3  10.0.5.4 (10.0.5.4)  0.807 ms  0.805 ms  0.802 ms
 4  10.0.6.22 (10.0.6.22)  0.966 ms  0.973 ms  0.764 ms
root@ucas-cod-2022:/home/ucas/08-mospf# 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)  0.488 ms  1.281 ms  1.354 ms
 2  10.0.2.2 (10.0.2.2)  1.376 ms  1.514 ms  1.550 ms
 3  10.0.4.4 (10.0.4.4)  2.533 ms  2.576 ms  2.576 ms
 4  10.0.6.22 (10.0.6.22)  2.575 ms  2.573 ms  2.572 ms
```

#### 四、思考

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?

因为 hello 包的发送较为频繁, 使用组播机制进行邻居发现可以减少网络中的广播风暴, 降低网络负载并提高效率。而使用单播机制进行链路状态扩散可以保证信息传输的可靠性和完整性, 同时避免因广播导致的冗余信息传输和处理。

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可扩展性?

修改数据结构, 使用更高效的算法, 还可以结合硬件加速、多线程等技术提高计算速度

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?

优化数据结构, 加速更新算法, 尽量减少路由更新频率。