计算机网络实验 8 报告

学号　2017K8009929059
姓名　於修远

网络路由实验

一、实验内容

- 实现路由器生成和处理 mOSPF 消息的相关操作，构建一致性链路状态数据库。
- 实现路由器计算路由表项的相关操作。
- 运行给定网络拓扑` topo.py `，验证 ping 和 traceroute 功能在各种情况下均处理正常。

二、实验流程

（一）链路邻居发现

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，向邻居节点发送状态信息，表明节点存在且有效。

```
void *sending_mospf_hello_thread(void *param) {
    // TODO: send mOSPF Hello message periodically.

    int length = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE;
    iface_info_t *iface;
    while (1) {
        pthread_mutex_lock(&mospf_lock);
        list_for_each_entry(iface, head: &instance->iface_list, list) {...}
        pthread_mutex_unlock(&mospf_lock);
        sleep(MOSPF_DEFAULT_HELLOINT);
    }

    return NULL;
}
```

具体而言，需要依次生成 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。

```
mospf_init_hello(mhello, iface->mask);

mospf_init_hdr(mhdr, MOSPF_TYPE_HELLO, len: MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE, instance->router_id, aid: 0);
mhdr->checksum = mospf_checksum(mhdr);

ip_init_hdr(ihdr, iface->ip, MOSPF_ALLSPFRouters, len: length - ETHER_HDR_SIZE, IPPROTO_MOSPF);
ihdr->checksum = ip_checksum(ihdr);

ehdr->ether_dhost[0] = 0x01;
ehdr->ether_dhost[2] = 0x5e;
ehdr->ether_dhost[5] = 0x05;
memcpy(ehdr->ether_shost, iface->mac, ETH_ALEN);
ehdr->ether_type = htons(ETH_P_IP);

iface_send_packet(iface, packet, length);
```

2、mOSPF HELLO 处理

接收到 HELLO 消息后首先检查该邻居节点是否已有记录。若是，直接更新存活时间并释放锁即可。

```
struct mospf_hdr *mhdr = (struct mospf_hdr *) (packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
u32 srid = ntohl(mhdr->rid);
mospf_nbr_t *nbr;
pthread_mutex_lock(&mospf_lock);
list_for_each_entry(nbr, head: &iface->nbr_list, list) {
    if (nbr->nbr_id == srid) {
        nbr->alive = 3 * iface->helloint;
        pthread_mutex_unlock(&mospf_lock);
        return;
    }
}
```

若不然，则在对应端口增加新的邻居节点条目，并记录相关信息。

```
struct iphdr *ihdr = packet_to_ip_hdr(packet);
struct mospf_hello *mhello = (struct mospf_hello *) ((char *) mhdr + MOSPF_HDR_SIZE);
nbr = (mospf_nbr_t *) malloc(sizeof(mospf_nbr_t));
nbr->nbr_id = srid;
nbr->nbr_ip = ntohl(ihdr->saddr);
nbr->nbr_mask = ntohl(mhello->mask);
nbr->alive = 3 * iface->helloint;
list_add_tail(&nbr->list, &iface->nbr_list);
iface->num_nbr++;
pthread_mutex_unlock(&mospf_lock);
```

3、邻居信息更新检查

` checking_nbr_thread `线程每 3 秒唤醒一次，主体部分如下图所示。遍历每个端口的邻居列表，并将存活时间减 3 秒。当存活时间归零时认为节点失效，删除本条记录。

```
list_for_each_entry(iface, head: &instance->iface_list, list) {
    mospf_nbr_t *nbr, *q;
    //printf("%s : ", iface->name);
    list_for_each_entry_safe(nbr, q, head: &iface->nbr_list, list) {
        //printf("id : %d ; ip : %d ; mask : %d ", nbr->nbr_id, nbr-
        nbr->alive -= 3;
        if (nbr->alive <= 0) {
            iface->num_nbr--;
            list_delete_entry(&nbr->list);
            free(nbr);
```

（二）链路状态扩散

1、mOSPF LSU 消息发送

路由节点每隔 lsuint 的时间间隔发送一次 LSU 消息，每次发送的所有 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。

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。

```
list_for_each_entry(iface, head: &instance->iface_list, list) {
    nadv += iface->num_nbr;
    if (!iface->num_nbr) nadv++;
}
u32 len_lsa = MOSPF_LSA_SIZE * nadv;
struct mospf_lsa *lsa_all = (struct mospf_lsa *) malloc(len_lsa);
struct mospf_lsa *temp = lsa_all;
list_for_each_entry(iface, head: &instance->iface_list, list) {
    if (!iface->num_nbr) {
        temp->network = htonl( hostlong: iface->ip & iface->mask);
        temp->mask = htonl(iface->mask);
        temp->rid = htonl( hostlong: 0);
        temp++;
    } else
        list_for_each_entry(nbr, head: &iface->nbr_list, list) {
            temp->network = htonl( hostlong: nbr->nbr_ip & nbr->nbr_mask);
            temp->mask = htonl(nbr->nbr_mask);
            temp->rid = htonl(nbr->nbr_id);
            temp++;
        }
}
```

此后，针对各个端口分别组装 mOSPF 头部、IP 头部和以太网头部信息后作为 IP 数据包发送即可。

```
memcpy(mlsa, lsa_all, len_lsa);

mospf_init_lsu(mlsu, nadv);

mospf_init_hdr(mhdr, MOSPF_TYPE_LSU, len: MOSPF_HDR_SIZE + MOSPF_LSU_SIZE
                                    + len_lsa, instance->router_id, aid: 0);
mhdr->checksum = mospf_checksum(mhdr);

ip_init_hdr(ihdr, iface->ip, nbr->nbr_ip, len: len_pkt - ETHER_HDR_SIZE, IPPROTO_MOSPF);
ihdr->checksum = ip_checksum(ihdr);

ip_send_packet(packet, len_pkt);
```

2、mOSPF LSU 消息处理

根据数据包中的信息，LSU 消息可以分为三种：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` new_entry `来标识。

```
list_for_each_entry(entry, head: &mospf_db, list) {
    if (ntohl(mhdr->rid) == entry->rid) {
        if (ntohs(mlsu->seq) <= entry->seq) {
            pthread_mutex_unlock(&db_lock);
            return;
        }
        new_entry = 0;
        break;
    }
}
```

对于新的链路状态信息，需要开辟新的存储空间，并接入一致性链路状态数据库中。初始化该条目的序号为路由器标号，并将条目的矩阵映射项置为 ADD 标识，即后续需要在路径矩阵中添加该条目信息。其他内容同一写入数据库即可。

```
if (new_entry) {
    entry = (mospf_db_entry_t *) malloc(sizeof(mospf_db_entry_t));
    entry->rid = ntohl(mhdr->rid);
    //printf("recv ckpt 2.3\n");
    entry->map2mtx = NODE_ADD;
    entry->array = (struct mospf_lsa *) malloc(sizeof(struct mospf_lsa));
    list_add_tail(&entry->list, &mospf_db);
}
//printf("recv ckpt 2.5\n");
entry->seq = ntohs(mlsu->seq);
entry->nadv = ntohl(mlsu->nadv);
entry->alive = MOSPF_DATABASE_TIMEOUT;
u32 len_lsa = MOSPF_LSA_SIZE * entry->nadv;
entry->array = (struct mospf_lsa *) realloc(entry->array, len_lsa);
```

遍历 LSU 消息中的 LSA 表项，逐条进行字节序转换后写入数据库表项。

```
for (int i = 0; i < entry->nadv; ++i) {
    dblsa->rid = ntohl(mlsa->rid);
    dblsa->mask = ntohl(mlsa->mask);
    dblsa->network = ntohl(mlsa->network);
    dblsa++;
    mlsa++;
}
```

最后，将 TTL 减 1 后仍大于 0 的消息进行广播转发，发往所有非接收端口。每次发送前，需要重新设置下一跳 IP 地址，并更新 IP 头部的校验和。

```
list_for_each_entry(ientry, head: &instance->iface_list, list) {
    if (ientry->index == iface->index) continue;
    ihdr->saddr = htonl(ientry->ip);
    if (ientry->num_nbr)
        list_for_each_entry(nbr, head: &ientry->nbr_list, list) {
            char *cpacket = (char *) malloc(len);
            memcpy(cpacket, packet, len);
            struct iphdr *cihdr = packet_to_ip_hdr(cpacket);
            cihdr->daddr = htonl(nbr->nbr_ip);
            cihdr->checksum = ip_checksum(cihdr);
            ip_send_packet(cpacket, len);
        }
}
```

3、链路状态数据库老化处理

本设计中，链路状态数据库每 5 秒进行一次老化检查，将现存条目存活时间减 5，并标记存活时间归零的条目。这些条目会在路径矩阵更新时被删除，此处只进行标记是为了方便路径矩阵删除映射条目。

```
while (1) {
    pthread_mutex_lock(&db_lock);
    if (!list_empty( list: &mospf_db))
        list_for_each_entry_safe(dbEntry, q, head: &mospf_db, list) {
            dbEntry->alive -= 5;
            if (dbEntry->alive <= 0) {
                dbEntry->map2mtx = NODE_DELETE - dbEntry->map2mtx;
            }
        }
    pthread_mutex_unlock(&db_lock);
    sleep( seconds: 5);
}
```

（三）路径计算与路由表项生成

1、数据结构与初始化

为了使用 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。` record `数组作为表项索引，需要保存其映射的数据库条目指针、有效位、路径长度、下一跳 IP 及相应转出端口等信息。数据库表项的` map2mtx `参数保存了其映射的 record 标号，形成互相映射的索引关系。

```
typedef struct {
    mospf_db_entry_t* dbEntry;
    int valid;
    int dist;
    u32 gw;
    iface_info_t* iface;
} mtx_map_t;
```

路径矩阵的每一项都是一个记录网络号和掩码号和连通性的结构体，若第 i 行 j 列的` value `属性为 1，则表示存在从第 i 条记录的路由到第 j 条记录的路由的链路。

```
typedef struct {
    u32 network;
    u32 mask;
    int value;
} net_info_t;
```

针对后续实验的需要，链路状态数据库的初始化函数中，额外增加了 record 数组的初始化信息。同时记录固有路由表项数目 start，对于这些从内核读入的条目，后续处理中不会简单删除，而是直接修改其中的值，这样可以避免其他线程查找表项时发生错误。

```
void init_mospf_db() {
    init_list_head(&mospf_db);
    for (int i = 0; i < MAX_NODES; ++i) {
        record[i].valid = 0;
    }
    rt_entry_t *rtEntry;
    list_for_each_entry(rtEntry, head: &rtable, list) {
        start++;
    }
    pthread_mutex_init(&db_lock, mutexattr: NULL);
    pthread_t rt;
    pthread_create(&rt, attr: NULL, setting_rt_list_thread, arg: NULL);
}
```

2、路径矩阵维护

维护路径矩阵的第一步是更新行列映射。遍历链路状态数据库，根据接收 LSU 数据包时改动的信息，对于需要删除的表项，还原其 record 下标后将条目有效位置 0，再将其从一致性链路状态数据库中删除；对于新增加的表项，遍历 record 条目表并找到一个无效表项，将其置为有效后为二者相互建立映射关系。此后遍历所有有效行，根据链路状态数据库保存的链路关系即可实现对路径矩阵的修改更新。

```
void renew_matrix() {
    mospf_db_entry_t *dbEntry, *q;
    int count = 0;
    pthread_mutex_lock(&db_lock);
    if (!list_empty( list: &mospf_db))
        list_for_each_entry_safe(dbEntry, q,  head: &mospf_db, list) {
            if (dbEntry->map2mtx <= NODE_DELETE) {...}
            else if (dbEntry->map2mtx == NODE_ADD) {
                for (int i = 0; i < MAX_NODES; i++) {...}
            }
            count++;
        }
    pthread_mutex_unlock(&db_lock);
    init_matrix();
    int i = 0;
    while (count) {
        if (!record[i++].valid) continue;
        count--;
        //printf("editting line %d ||", i-1);
        edit_mtx( line: i - 1);
    }
    sup = i;
}
```

更新填写路径矩阵的函数` edit_mtx `的核心部分为下图所示的 2 层循环。索引第 line 行的 record 条目，遍历其映射的链路状态条目，对每一条目记录的路由器号，查找在第 line 行中是否存在映射同一路由器的列条目。若存在，置矩阵在该位置的 value 值为 1，并记录条目中的网络号和掩码。其含义为：第 line 行的条目对应路由器通过这一组网络信息连接到第 j 行的条目。

```
for (i = 0; i < dbEntry->nadv; ++i) {
    lsa_item = dbEntry->array + i;
    for (int j = 0; j < sup; ++j) {
        if (record[j].valid && (record[j].dbEntry->rid == lsa_item->rid)) {
            path_map[line][j].value = 1;
            path_map[line][j].network = lsa_item->network;
            path_map[line][j].mask = lsa_item->mask;
            //printf("  "IP_FMT, HOST_IP_FMT_STR(lsa_item->rid));
        }
    }
}
```

3、最短路径计算与路由表项生成

每次重新生成路由表时，直接清除所有非默认表项，便于统一处理失效表项。

```
list_for_each_entry_safe(item, q,  head: &rtable, list) {
    if (check++ < start) continue;
    list_delete_entry(&item->list);
    free(item);
}
```

由于路径矩阵中没有保存本节点信息，所以首先需要通过遍历各节点的邻居列表，筛选出所有距离为 1 的节点集合。具体而言，对于每个邻居节点，在路径矩阵条目信息中检索其位置，将距离信息设置为 1，并记录转发端口和下一跳 IP 等信息。同时，根据这些信息更新固有条目或写入新路由条目。

```
list_for_each_entry(iface, head: &instance->iface_list, list) {
    list_for_each_entry(nbr, head: &iface->nbr_list, list) {
        for (int i = 0; i < sup; ++i) {
            if (record[i].valid && (record[i].dbEntry->rid == nbr->nbr_id)) {
                record[i].dist = 1;
                record[i].gw = nbr->nbr_ip;
                record[i].iface = iface;
                rt_entry_t *test = chk_init( subnet: nbr->nbr_ip & nbr->nbr_mask);
                if (test) {
                    farther = 1;
                    test->gw = nbr->nbr_ip;
                    break;
                }
                rt_entry_t *temp = new_rt_entry( dest: nbr->nbr_ip & nbr->nbr_mask, nbr->nbr_mask,
                                                 nbr->nbr_ip, iface);
                add_rt_entry(temp);
                farther = 1;
                break;
            }
        }
    }
}
```

最后，只需使用 Dijkstra 算法并让距离每次增加 1，并对每次扩散后得到的点集填写路由条目即可。具体而言，farther 参数标识本层中是否可以搜索到距离更远的节点，若是，则重复进行 Dijkstra 算法；否则结束路径计算和路由表填写。搜索同样分为两部分，假设到本节点距离为 dist 的点集非空，遍历这些点，一方面搜索是否存在距离为 dist+1 的点，另一方面检查这些点中是否有未填写进入路由表的子网信息。

```
while (farther) {
    farther = 0;
    dist++;
    for (int i = 0; i < sup; ++i) {
        if (record[i].dist == dist) {
            for (int j = 0; j < sup; ++j) {...}
            mospf_db_entry_t *dbEntry = record[i].dbEntry;
            struct mospf_lsa *lsa_item;
            pthread_mutex_lock(&db_lock);
            //printf("line %d :", i);
            for (int k = 0; k < dbEntry->nadv; ++k) {...}
            pthread_mutex_unlock(&db_lock);
        }
    }
}
```

点集扩散的逻辑位于第一个内部 for 循环中。距离为 dist 的节点试图搜索与之相连的、且此前未被访问并建立路径过的节点。对于这样的节点，设置距离后复制自身的下一跳 IP 和转出端口，因为这意味着新旧节点将共享之前的所有传播路径。

```
if (path_map[i][j].value == 1 && record[j].dist == 0) {
    record[j].dist = dist + 1;
    record[j].gw = record[i].gw;
    record[j].iface = record[i].iface;
    rt_entry_t *temp = new_rt_entry( dest: path_map[i][j].network & path_map[i][j].mask,
                                     path_map[i][j].mask, record[j].gw, record[j].iface);
    add_rt_entry(temp);
    farther = 1;
    break;
}
```

子网信息补充逻辑位于第二个内部 for 循环中，该补充的原因在于此前建立路径矩阵时，只为独立发出过 LSU 信息的路由器节点建立了行列索引，而其连接的子网则不会拥有索引。所以，需要对这些子网的路由条目进行一次补充。对于每个可达的节点，检查其下属 LSA 条目中是否存在子网不在矩阵索引中且不同于本机的条目。对于这些条目，根据已有记录补充路由条目。

```
lsa_item = dbEntry->array + k;
if (lsa_item->rid == instance->router_id) {
    //printf(IP_FMT" get from router\n", HOST_IP_FMT_STR(lsa_item->rid));
    continue;
}
int host = 1;
for (int l = 0; l < sup; ++l) {
    if (record[l].valid && (record[l].dbEntry->rid == lsa_item->rid)) {
        host = 0;
        //printf(IP_FMT" get from %d\n", HOST_IP_FMT_STR(lsa_item->rid), l);
        break;
    }
}
if (host) {
    //printf(IP_FMT" get from host\n", HOST_IP_FMT_STR(lsa_item->rid));
    rt_entry_t *temp = new_rt_entry( dest: lsa_item->mask & lsa_item->network, lsa_item->mask,
                                     record[i].gw, record[i].iface);
    add_rt_entry(temp);
}
```

三、结果分析

（一）ping 功能测试

为了测试方便，直接修改实验脚本，使 4 个路由节点自启动 mospfd 程序。

执行微调的`topo.py`脚本后，等待约 40 秒，用 h1 节点对 h2 节点执行 ping 操作，得到结果如下图所示，可见节点间连通性正常。

```
mininet> h1 ping h2
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=0.381 ms
64 bytes from 10.0.6.22: icmp_seq=2 ttl=61 time=0.398 ms
64 bytes from 10.0.6.22: icmp_seq=3 ttl=61 time=0.343 ms
^C
--- 10.0.6.22 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
```

（二）traceout 测试及节点变动测试

用 h1 节点对 h2 节点执行 traceroute 操作，可见路径查找正常。执行` link r2 r4 down `命令，取消 r2 和 r4 节点间的链路。继续等待约 40s，重新执行 h1 到 h2 的 traceroute 操作，可见路径发生改变且符合理论情况，如下图所示。

```
mininet> h1 traceroute h2
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)  0.225 ms  0.195 ms  0.187 ms
 2  10.0.2.2 (10.0.2.2)  0.883 ms  0.891 ms  0.892 ms
 3  10.0.4.4 (10.0.4.4)  0.887 ms  0.881 ms  0.848 ms
 4  10.0.6.22 (10.0.6.22)  0.838 ms  0.832 ms  0.827 ms
mininet> link r2 r4 down
mininet> h1 traceroute h2
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)  0.268 ms  0.191 ms  0.169 ms
 2  10.0.3.3 (10.0.3.3)  0.495 ms  0.483 ms  0.473 ms
 3  10.0.5.4 (10.0.5.4)  8.658 ms  8.668 ms  8.662 ms
 4  10.0.6.22 (10.0.6.22)  8.656 ms  8.649 ms  8.643 ms
```

（三）实验总结

本次实验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，再到映射为路径矩阵并通过计算最短路径来生成路由表的功能。本次设计的主要优化点在于路径矩阵与链路状态数据库的合理映射以降低反复遍历数据库的开销。同时，路径矩阵只保存路由器节点对应条目，可以大大减小内存占用和维护开销。非路由器节点的条目恰好可以在后续检索中补全，总体而言增加的时间开销可以接受。