网络路由实验

中国科学院大学
袁欣怡 2018K8009929021
2021.5.25

实验内容

使用 mOSPF 协议构建网络结点的路由表，并测试是否构建成功。每个结点在这个过程中需要完成的操作包括：

1. 创建自己的链路状态数据包 LSU；
2. 发送 LSU，向其他结点通告自己的链路状态信息；
3. 根据收到其他结点的 LSU 构建整个网络的拓扑结构；
4. 根据网络拓扑结构计算到网络中其他结点的最短路径，生成路由表；
5. 当链路状态发生变化时，重复以上步骤。

我们的测试分为两个部分：

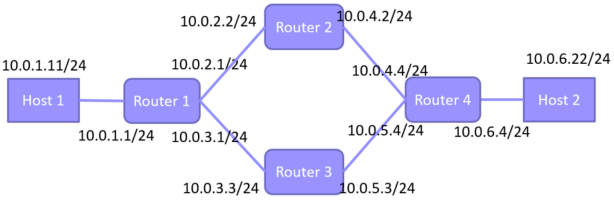
1. 测试路由器能否正常生成路由表并打印；
2. 测试链路状态发生变化时路由器能否重新生成路由表。

实验流程

1. 搭建实验环境

本实验中涉及到的文件主要有：

- main.c：编译后生成 mospf 可执行文件，在路由器结点上运行。
- mospf_daemon.c：本次实验需要完成其中 sending_mospf_hello_thread 、 checking_nbr_thread 等函数，完成的函数功能包括：周期发送 HELLO 消息，处理收到的 HELLO 消息，发送 LSU 报文，处理收到的 LSU 报文等等。
- Makefile：处理终端 make all 和 make clean 命令。
- topo.py：构建六结点网络拓扑结构的 topo 文件，网络拓扑结构如下图所示：



2. 实验代码设计

设计思路

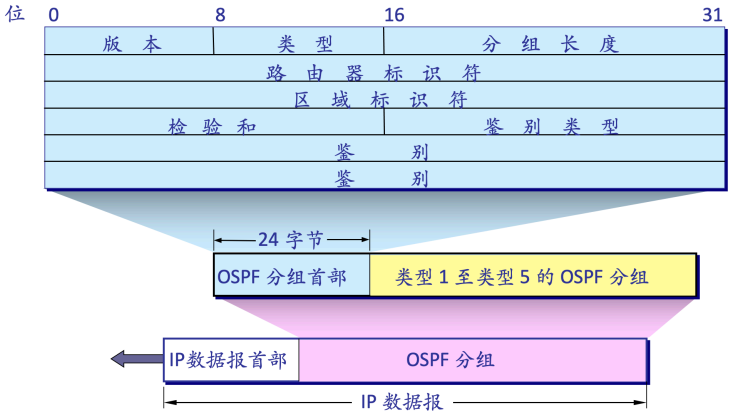
路由是我们进行网络通信时非常重要的步骤。常见的路由协议可以分为静态和动态两种。之前我们实现的都是静态路由协议，这需要我们充当管理员的角色，在网络通信开始之前手动配置路由表。本次实验中，我们需要实现的是动态路由协议 OSPF 协议。和静态路由相比，动态路由可以自动建立自己的路由表，并且在网络拓扑结构发生变化时自动进行调整。因此，在网络规模较大的情况下，静态路由会给管理员带来陡增的压力，此时选择动态路由更加明智。

OSPF 协议， Open Shortest Path First，是一种基于链路状态的路由协议。 OSPF 工作原理可以分为三步：1. 邻居建立：路由器之间发现并建立邻居关系；2. 同步链路状态数据库：每台路由器产生并向邻居洪泛链路状态信息，同时手机来自其他路由器的链路状态信息；3. 计算最优路由：根据 Dijkstra 算法，计算自己到网络中每个结点的最短路径。

本次实验用到的协议为组播扩展 OSPF，即 MOSPF 。它在原 OSPFv2 的基础上作了增强，使之支持 IP 组播路由。它与 OSPFv2 的区别在于：

1. OSPFv2 的 protocol number 为89，而 mOSPF 为90；
2. mOSPF 对数据包格式进行了适当简化；
3. OSPFv2 基于可靠洪泛：收到 LSU 数据包后需要回复 ACK；
4. OSPFv2 有更多的消息类型。例如，链路状态数据库 Summary；
5. OSPFv2 有安全认证机制（鉴别）；

mOSPF 数据包格式和 OSPFv2 相同：



版本号：2。

类型：5种，类型1： HELLO 分组，类型2：数据库描述(Database Description)分组，类型3：链路状态请求(Link State Request)分组，类型4：链路状态更新(Link State Update)分组，类型5：链路状态确认(Link State Acknowledgment)分组。

分组长度： mOSPF 消息的长度（首部+内容）。

路由器标识符：发送本消息的路由器 ID 。

区域标识符：划分消息传播的区域，本实验中设置为 0.0.0.0 。

检验和：检验是否出错。

本次实验中，需要用到类型1（ HELLO ）和类型4（ LSU ）两种分组。

- HELLO 分组：

mask	
hello interval	padding

mask ： 生成本消息的端口的掩码。

hello interval ： 两次发送 hello 消息之间间隔的时间。

padding ： 对齐，填充为0。

- LSU 分组：

sequence number	ttl	unused
#(advertisement)		
network		
mask		
router id		
… …		

sequence number ： LSU 的序列号。

ttl ： 剩余生存时间。

#(advertisement ： 邻居的个数。

network ： 邻居所在的网段。

mask ： 邻居所在网段的掩码。

router id：邻居路由器 ID 。

实验代码

由于本次实验代码量较大，故不贴出全部代码，仅在需要时添加代码方便理解。以下代码，如无特别声明，均在 mospf_daemon.c 中实现。

1. 函数 void *sending_mospf_hello_thread(void *param)：

   功能：周期性发送 HELLO 报文。

   此函数由一个线程单独运行，在开始全需要获得 mospf 互斥锁。

   结点通过周期性发送 mOSPF Hello 消息，向外界宣告自己的存在。HELLO 消息中包含路由器 ID，端口掩码等信息。

   首先计算报文需要占用空间大小 len，然后分配相应空间，并进行初始化操作。初始化操作包括设置以太网首部、IP 首部、mospf 首部、hello 段和校验和。设置结束后释放互斥锁。

```
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 * packet = (char*)malloc(len);
    bzero(packet, len);

    struct ether_header *eh = (struct ether_header *)packet;
    struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
    struct mospf_hdr * mospf_header = (struct mospf_hdr *)((char *)ip_hdr +
IP_BASE_HDR_SIZE);
    struct mospf_hello * hello = (struct mospf_hello *)((char *)mospf_header +
MOSPF_HDR_SIZE);

    // 设置以太网首部
    eh->ether_type = htons(ETH_P_IP);
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    u8 dhost[ETH_ALEN] = {0x01,0x00,0x5e,0x00,0x00,0x05};
    memcpy(eh->ether_dhost, dhost, ETH_ALEN);

    // 设置IP首部
    ip_init_hdr(ip_hdr,
            iface->ip,
            MOSPF_ALLSPFRouters,
            len - ETHER_HDR_SIZE,
            IPPROTO_MOSPF);

    // 设置mospf首部
    mospf_init_hdr(mospf_header,
              MOSPF_TYPE_HELLO,
              len - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE,
              instance->router_id,
              0);

```

```
31      mospf_init_hello(hello, iface->mask);
32
33      // 计算校验和
34      mospf_header->checksum = mospf_checksum(mospf_header);
35
36      // 发包
37      iface_send_packet(iface, packet, len);
38  }
```

2. 函数 void *checking_nbr_thread(void *param)：

   功能：检查邻居列表中老化的结点。

   判断邻居结点老化的标准是： 3*hello_interval 时间内没有更新。

```
list_for_each_entry_safe(nbr_pos, nbr_q, &iface->nbr_list, list) {
    nbr_pos->alive++;
    if (nbr_pos->alive > 3 * iface->helloint) {
        list_delete_entry(&nbr_pos->list);
        free(nbr_pos);
        iface->num_nbr--;
        sending_mospf_lsu();
    }
}
```

3. 函数 void *checking_database_thread(void *param)：

   功能：周期性检查数据库中老化的结点。

   判断结点老化的标准： MOSPF_DATABASE_TIMEOUT 时间内没有更新链路状态。

```
list_for_each_entry_safe(db_pos, db_q, &mospf_db, list) {
    db_pos->alive ++ ;
    if (db_pos->alive > MOSPF_DATABASE_TIMEOUT) {
        list_delete_entry(&db_pos->list);
        free(db_pos);
    }
}
```

   在删除老化结点后，还需要更新路由表。本实验中用 update_rtable 函数完成这一功能。

```
void update_rtable() {
    int prev[ROUTER_NUM];
    int dist[ROUTER_NUM];
    init_graph();
    Dijkstra(prev, dist);
    update_router(prev, dist);
}
```

4. 函数 void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)：

   功能：处理收到的 HELLO 报文。

处理步骤包括：提取路由器 ID 、 IP 、 MASK 等信息。若这些信息不在 nbr_list 中，则存入，否则找到对应条目并更新到达时间。注意在操作 nbr_list 时需要获得 mospf 互斥锁。

```
pthread_mutex_lock(&mospf_lock);
mospf_nbr_t * nbr_pos = NULL;

list_for_each_entry(nbr_pos, &iface->nbr_list, list) {
    if (nbr_pos->nbr_ip == ip) { // 已有条目
        nbr_pos->alive = 0;
        isFound = 1;
        break;
    }
}

if (!isFound) { // 没有对应条目，新建一条
    mospf_nbr_t * new_nbr = (mospf_nbr_t *) malloc(sizeof(mospf_nbr_t));
    new_nbr->alive = 0;
    new_nbr->nbr_id = id;
    new_nbr->nbr_ip = ip;
    new_nbr->nbr_mask = mask;
    list_add_tail(&(new_nbr->list), &iface->nbr_list);
    iface->num_nbr++;
    sending_mospf_lsu();
}

pthread_mutex_unlock(&mospf_lock);
```

5. 函数 void *sending_mospf_lsu_thread(void *param)：

   功能：周期性发送 LSU 报文。

   此函数由一个线程单独运行。开始时获取 mospf 互斥锁，然后调用 handle_mospf_lsu 函数，结束后释放互斥锁，并 sleep 一段时间。

6. 函数 void sending_mospf_lsu()：

   功能：组包并发送 LSU 报文。

   线程函数 sending_mospf_lsu_thread 会周期性调用此函数，结点的邻居发生变动时也会调用此函数。

   该函数向邻居结点发送的链路状态信息为：

   - 该结点路由器 ID，邻居结点路由器 ID，网段和掩码
   - 当该端口没有相邻路由器时也需要发送，将邻居结点路由器 ID 设置为 0
   - 序列号，目的 IP 地址和目的 MAC 地址

```
list_for_each_entry (iface, &instance->iface_list, list) {
    if (iface->num_nbr == 0) { // 没有相邻路由器
        array[index].mask = htonl(iface->mask);
        array[index].network = htonl(iface->ip & iface->mask);
        array[index].rid = 0;
        index++;
    }
```

```
    else { // 有相邻路由器
        mospf_nbr_t * nbr_pos = NULL;

        list_for_each_entry (nbr_pos, &iface->nbr_list, list) {
            array[index].mask = htonl(nbr_pos->nbr_mask);
            array[index].network = htonl(nbr_pos->nbr_ip & nbr_pos->nbr_mask);
            array[index].rid = htonl(nbr_pos->nbr_id);
            index++;
        }
    }
}
```

7. 函数 void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)：

   功能：处理收到的 LSU 报文。

   函数在收到 LSU 报文时，先检查之前是否收到过该结点的链路状态信息。如果未收到过或者新收到的 LSU 报文序列号更大，那么需要更新链路状态数据库， ttl 减少1。如果此时 ttl 仍为正，那需要向除该端口以外的端口转发该报文。

   整个函数的代码如下：

```
void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stdout, "lab11 added: handle mOSPF LSU message.\n");
    struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
    struct mospf_hdr * mospf_head = (struct mospf_hdr *)((char*)ip_hdr + IP_HDR_SIZE(ip_hdr));
    struct mospf_lsu * lsu = (struct mospf_lsu *)((char*)mospf_head + MOSPF_HDR_SIZE);
    struct mospf_lsa * lsa = (struct mospf_lsa *)((char*)lsu + MOSPF_LSU_SIZE);

    pthread_mutex_lock(&mospf_lock);
    u32 rid = ntohl(mospf_head->rid);
    u32 ip = ntohl(ip_hdr->saddr);
    u16 seq = ntohs(lsu->seq);
    u8 ttl = lsu->ttl;
    u32 nadv = ntohl(lsu->nadv);
    int isFound = 0;
    int isUpdated = 0;

    // 查找链路信息数据库
    mospf_db_entry_t * entry_pos = NULL;
    list_for_each_entry (entry_pos, &mospf_db, list) {
        if (entry_pos->rid == rid) { // 查找成功
            if (entry_pos->seq < seq) { // 新收到LSU的序列号更大，需要更新链路状态
                entry_pos->seq = seq;
                entry_pos->nadv = nadv;
                entry_pos->alive = 0;
                for (int i = 0; i < nadv; i++) {
                    entry_pos->array[i].mask = ntohl(lsa[i].mask);
                    entry_pos->array[i].network = ntohl(lsa[i].network);
```

```
                entry_pos->array[i].rid = ntohl(lsa[i].rid);
                fprintf(stdout, "Update db entry "IP_FMT"  "IP_FMT"  "IP_FMT"  "IP_FMT"\n",
                    HOST_IP_FMT_STR(entry_pos->rid),
                    HOST_IP_FMT_STR(entry_pos->array[i].network),
                    HOST_IP_FMT_STR(entry_pos->array[i].mask),
                    HOST_IP_FMT_STR(entry_pos->array[i].rid));
            }
            isUpdated = 1;
        }
        isFound = 1;
        break;
    }
}

// 没有找到，新建
if (!isFound) {
    entry_pos = (mospf_db_entry_t *)malloc(sizeof(mospf_db_entry_t));
    entry_pos->rid = rid;
    entry_pos->seq = seq;
    entry_pos->nadv = nadv;
    entry_pos->alive = 0;

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

pthread_mutex_unlock(&mospf_lock);

if (isUpdated == 0) {
    return;
}


lsu->ttl -- ;
if (lsu->ttl > 0) { // 生存期未耗尽，需要转发
    iface_info_t * iface_pos = NULL;
    list_for_each_entry (iface_pos, &instance->iface_list, list) {
        mospf_nbr_t *nbr = NULL;
        list_for_each_entry(nbr, &iface_pos->nbr_list, list) {
            if (nbr->nbr_id == ntohl(mospf_head->rid)) {
                continue;
            }
            char *forwarding_packet = (char *)malloc(len);
```

```
78                memcpy(forwarding_packet, packet, len);
79
80                struct iphdr * iph = packet_to_ip_hdr(forwarding_packet);
81                iph->saddr = htonl(iface_pos->ip);
82                iph->daddr = htonl(nbr->nbr_ip);
83
84                struct mospf_hdr * mospfh = (struct mospf_hdr *)((char *)iph +
    IP_HDR_SIZE(iph));
85                mospfh->checksum = mospf_checksum(mospfh);
86                iph->checksum = ip_checksum(iph);
87
88                ip_send_packet(forwarding_packet, len);
89            }
90        }
91    }
92
93 }
```

8. 函数 void print_mospf_db()：

   功能：打印链路信息数据库。

9. 函数 void Dijkstra(int prev[], int dist[])：

   功能：执行 Dijkstra 算法。具体实现如下：

```
1  void Dijkstra(int prev[], int dist[]) {
2      int visited[ROUTER_NUM];
3      for(int i = 0; i < ROUTER_NUM; i++) {
4          prev[i] = -1;
5          dist[i] = INT8_MAX;
6          visited[i] = 0;
7      }
8
9      dist[0] = 0;
10
11     for(int i = 0; i < idx; i++) {
12         int u = min_dist(dist, visited, idx);
13         visited[u] = 1;
14         for (int v = 0; v < idx; v++){
15             if (visited[v] == 0 && dist[u] + graph[u][v] < dist[v]) {
16                 dist[v] = dist[u] + graph[u][v];
17                 prev[v] = u;
18             }
19         }
20     }
21
22 }
```

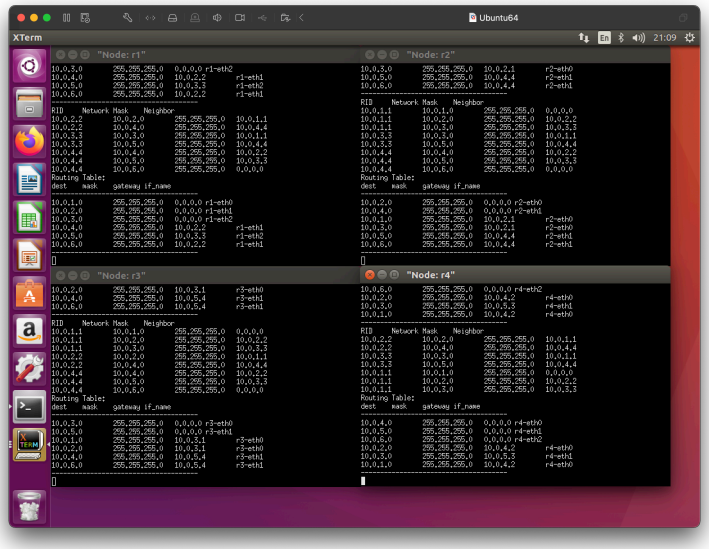
   其中调用了 min_dist 函数。这个函数的功能是在未访问的结点中，找到距离已访问结点最近的一个。

10. 函数 void update_router (int prev[], int dist[])：

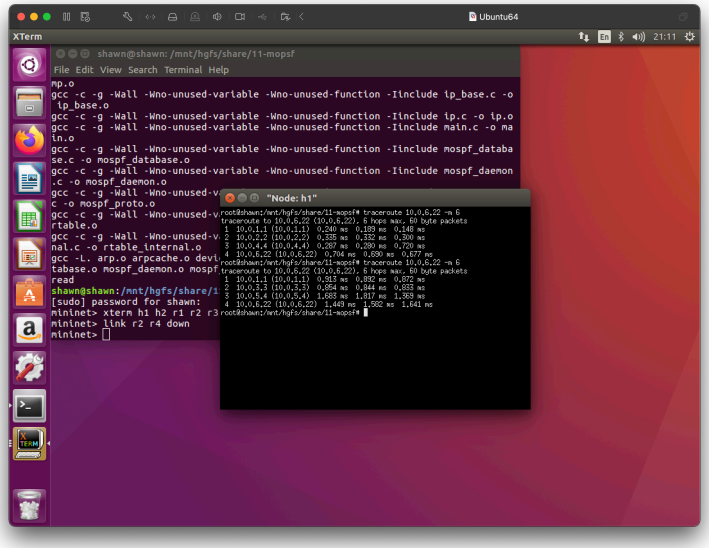
功能：根据函数 Dijkstra 的结果更新路由表。

3. 实验结果分析

实验内容一：测试能否生成正确路由表。



实验内容二：测试路由变动时能否重新生成路由表。



均能完成。

实验总结与思考题

实验总结

1. 使用指针的地址加减时要转换成 char * 类型。
2. 有相同前缀的函数在调用时不能混淆。

思考题

- 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?

  HELLO 报文会定期发送，所以丢失一个 HELLO 报文不会造成很大影响，故结点为了节约网络资源不会回复 HELLO 报文，且使用组播方式发送。

  LSU 报文传递的信息更重要，且 OSPFv2 中结点收到 LSU 报文后需要返回 ACK，故采用单播方式降低出错率，确保结点能收到报文并返回 ACK 。由于 LSU 报文在网络稳定后不会持续发送，所以这么做不会占用太多网络资源。

- 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?

  可以提高 hello_time 和生存时间。

- 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?

  可以使用 hash 等快速查找算法，减少更新路由表时查找所需要的时间。

参考资料

1. 常见路由协议总览以及路由协议分类方式
2. OSPF（一）OSPF协议简介
3. MOSPF
4. IP多播技术介绍(二)
5. Dijkstra算法(一)之 C语言详解