

- 网络路由实验
  - 实验内容
    - 实验内容一
    - 实验内容二
    - 测试与验证
  - 设计思路
    - 生成和处理mOSPF Hello消息
      - `void *sending_mospf_hello_thread(void *param)` 函数
      - `void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)` 函数
      - `void *checking_nbr_thread(void *param)` 函数
    - 生成和处理mOSPF LSU消息
      - `void sending_mospf_lsu()` 函数
      - `void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)` 函数
      - `void *checking_database_thread(void *param)` 函数
    - 实现路由器计算路由表项的相关操作
      - `void update_rtable()` 函数
      - `void init_graph()` 函数
      - `void Dijkstra(int prev[], int dist[])` 函数
      - `void update_router (int prev[], int dist[])` 函数
  - 结果验证
  - 反思与总结

# 网络路由实验

2020年12月2日

蔡润泽

本实验 [Github](#) 地址

## 实验内容

### 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

### 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

## 测试与验证

- 各个节点生成一致的链路状态数据库
- tracerout 能获得正确结果

## 设计思路

### 生成和处理mOSPF Hello消息

**void \*sending\_mospf\_hello\_thread(void \*param) 函数**

该函数被一个线程单独开启运行, 每个节点周期性 (5秒) 节点就向外界宣告自己的存在, 并发送mOSPF Hello消息: 包括router ID, 端口的mask等消息。

该节点发送IP包的目的IP地址为 224.0.0.5, 目的MAC地址为 01:00:5E:00:00:05。

**void handle\_mospf\_hello(iface\_info\_t \*iface, const char \*packet, int len) 函数**

每个端口在收到一个mOSPF HELLO包以后, 调用该函数进行处理。该函数需要从mOSPF HELLO包中获取rid、IP、MASK等信息。

若该发送该包的节点信息以及存在于iface中的nbr\_list中, 则更新其到达时间。否则, 将该节点的相关信息保存在iface中的nbr\_list中。

**void \*checking\_nbr\_thread(void \*param) 函数**

该函数负责处理邻居列表中老化的节点。若一个节点超过 3\*hello\_interval 没有更新, 则代表该节点老化, 需要被清理掉。

### 生成和处理mOSPF LSU消息

**void sending\_mospf\_lsu() 函数**

该函数负责组包并发送mOSPF LSU消息, 会被线程函数 sending\_mospf\_lsu\_thread 以30秒为周期进行调用发送。在该节点的邻居列表发生变动时, 也会调用该函数发送 LSU消息。

该函数向邻居节点发送的链路状态信息为：

- 该节点ID (mOSPF Header)、邻居节点ID、网络和掩码 (mOSPF LSU)
- 当端口没有相邻路由器时，也要表达该网络，邻居节点ID设为0（即端口连接局域网内部节点）
- 序列号(sequence number)，每次生成链路状态信息时加1
- 目的IP地址为邻居节点相应端口的IP地址，目的MAC地址为该端口的MAC地址（通过调用 `ip_send_packet` 函数发包）。

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

该函数在收到收到LSU消息后被调用进行处理数据包。如果之前未收到该节点的链路状态信息，或者该信息的序列号更大，则更新链路状态数据库，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。

### `void *checking_database_thread(void *param)` 函数

该线程函数负责周期性的检查失效节点。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。另外，在该函数会调用 `update_rtable` 进行路由表项的更新。

## 实现路由器计算路由表项的相关操作

### `void update_rtable()` 函数

该函数在 `checking_database_thread` 函数中被周期性调用，来进行路由表项的更新。具体实现如下：

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

其中prev是Dijkstra算法中的前序节点表，dist是不同节点到本节点的距离。

另外设有几个全局变量。idx表示节点个数，graph二维数组表示节点之间的链接数组，router\_list数组记录了每一个idx对应的rid。

### `void init_graph()` 函数

该函数负责初始化节点拓扑结构图的相关参数，具体实现如下：

```

void init_graph() {
    memset(graph, INT8_MAX -1, sizeof(graph));
    mospf_db_entry_t *db_entry = NULL;
    router_list[0] = instance->router_id;
    idx = 1;
    list_for_each_entry(db_entry, &mospf_db, list) {
        router_list[idx] = db_entry->rid;
        idx++;
    }
    db_entry = NULL;
    list_for_each_entry(db_entry, &mospf_db, list) {
        int u = get_router_list_index(db_entry->rid);
        for(int i = 0; i < db_entry->nadv; i++) {
            if(!db_entry->array[i].rid) {
                continue;
            }
            int v = get_router_list_index(db_entry->array[i].rid);
            graph[u][v] = graph[v][u] = 1;
        }
    }
}

```

上述函数中，初始化了router\_list, graph数组。

## void Dijkstra(int prev[], int dist[]) 函数

该函数负责实行Dijkstra算法。具体实现如下：

```

void Dijkstra(int prev[], int dist[]) {
    int visited[ROUTER_NUM];
    for(int i = 0; i < ROUTER_NUM; i++) {
        prev[i] = -1;
        dist[i] = INT8_MAX;
        visited[i] = 0;
    }
    dist[0] = 0;
    for(int i = 0; i < idx; i++) {
        int u = min_dist(dist, visited, idx);
        visited[u] = 1;
        for (int v = 0; v < idx; v++){
            if (visited[v] == 0 && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                prev[v] = u;
            }
        }
    }
}

```

其中调用了 min\_dist(dist, visited, idx) 函数来在未访问的节点中，选取离已访问节点最近的那个。

其具体实现如下：

```

int min_dist(int *dist, int *visited, int num) {
    int index = -1;
    for (int u = 0; u < num; u++) {
        if (visited[u]) {
            continue;
        }
        if (index == -1 || dist[u] < dist[index]) {
            index = u;
        }
    }
    return index;
}

```

## void update\_router (int prev[], int dist[]) 函数

该函数根据Dijkstra算法获得的节点拓扑信息来进行更新路由表。对于每个节点，会根据Dijkstra算法匹配到前序节点。又递归前递可以找到对于本节点而言，每一个其他的节点的下一条节点是多少，并以此更新路由表，从而确定到其他网络的下一跳网关地址、源节点的转发端口。

另外，实验初始化时，会从内核中读入到本地网络的路由条目，更新路由表时需要区分这些条目和计算生成的路由条目。本设计中非默认路由表会在一开始删去。

## 结果验证

本实验实验结果如下：

The screenshot displays a multi-panel view of a network simulation environment. The left panel shows a terminal window where the user 'vincent' is configuring a network topology using 'python topo.py' and running OSPF daemon processes. The right panel shows three separate windows for different nodes: 'Node: r1', 'Node: h1', and 'Node: r4'. Each node window displays its local OSPF configuration, including neighbor lists and routing tables. The routing tables show destinations, masks, gateways, and interface names, indicating the state of the network after configuration.

```

Node: r1
free2
free2
TODO: neighbor list timeout operation.
TODO: link state database timeout operation.
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 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
10.0.4.4 10.0.4.0 255.255.255.0 0.0.0.0
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
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 10.0.3.3 r1-eth2

Node: h1
root@ubuntu:~/code/12-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) 3.844 ms 3.792 ms 3.779 ms
 2 10.0.2.2 (10.0.2.2) 3.770 ms 3.761 ms 3.752 ms
 3 10.0.4.4 (10.0.4.4) 0.994 ms 0.995 ms 0.998 ms
 4 10.0.6.22 (10.0.6.22) 0.994 ms 0.992 ms 0.989 ms
root@ubuntu:~/code/12-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) 0.807 ms 0.764 ms 0.745 ms
 2 10.0.3.3 (10.0.3.3) 0.457 ms 0.468 ms 0.479 ms
 3 10.0.5.4 (10.0.5.4) 1.937 ms 1.929 ms 1.918 ms
 4 10.0.6.22 (10.0.6.22) 1.920 ms 1.932 ms 1.941 ms
root@ubuntu:~/code/12-mopsf#

Node: r4
10.0.1.0 255.255.255.0 10.0.5.3 r4-eth1
10.0.2.0 255.255.255.0 10.0.5.3 r4-eth1
TODO: neighbor list timeout operation.
TODO: link state database timeout operation.
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 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
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
10.0.3.0 255.255.255.0 10.0.5.3 r4-eth1
10.0.1.0 255.255.255.0 10.0.5.3 r4-eth1
10.0.2.0 255.255.255.0 10.0.5.3 r4-eth1

```

可以从上图中看出，不同节点之间可以通过HELLO和LSU信息，生成一致的链路状态信息。

另外tracerout的结果表明可以通过Dijkstra算法生成正确的路由表。并在网络状态改变（link r2 r4 down）时，自动生成新的路由表。

## 反思与总结

本次实验写代码的思路比较清晰，但是由于本次代码涉及到的文件较大，在编程时容易遇到几个问题，因此本次实验中，本人花了很多的时间来debug。有些问题值得总结，需要下次注意。

- IDE代码自动时需要留意。本次实验中，我写的一些变量名非常相近，甚至拥有一样的前缀，在编辑器自动补全时，出现了补全错变量名导致数据完全错误的情况。
- 对于指针的使用需要注意，在地址加减时要记得转换为 `char*` 类型。
- memset赋值存在上界不能超过 `0xff` 这一点需要留意。