实验中的问题&解决方案

成镇宇 2017080068 计76

大小端问题

由于实验中RoutingEntry和RipEntry的大小端格式不同，然后又由于本人在做实验的时候没有统一在每一个Entry下的大小端，所以浪费了很多无用的时间。

组播地址和MAC

之前对组播的MAC理解度不够，直接参考了HAL文件夹里面的IGMP的组播地址，结果导致对方不会更新我发的路由表。

收到response之后的处理

这里有很多需要注意的地方：

1. ripentry里面的metric是大端序，需要变成小端序再进行判断。
2. 在这里需要进行对目的地址进行精确匹配。

```
bool query_exact(uint32_t addr, uint32_t *nexthop, uint32_t
*if_index, uint32_t *metric) {
  std::string addr_str = toHex((int)addr);
  int max = -1, max_i = -1;
  for (int i=0; i<routers.size(); i++) {
    std::string tmp = toHex((int)routers.at(i).addr);
    if (addr_str == tmp) {
      max = tmp.length();
      max_i = i;
    }
  }
  if (max_i != -1) {
    *nexthop = routers.at(max_i).nexthop;
    *if_index = routers.at(max_i).if_index;
    *metric = routers.at(max_i).metric;
    return true;
  } else {
    return false;
  }
}
```

3. 假如跳数大于16需要删除路由的时候，需要判断一下收到的路由表是不是从自己的临近router发过来的，同时收到的rip路由的下一跳和自己的路由表地址一致的路由项的下一跳是否一致。假如不是自己临近router并且下一跳一致，则需要进行删除操作。

```
bool tmp = query_exact(rip.entries[i].addr, &query_nexthop,
&query_if_index, &query_metric);
if(rip.entries[i].metric+1 > 16
   && src_addr != 0x0203a8c0 // reverse poisoning detection // &&
src_addr != 0x0204a8c0
   && src_addr != 0x0201a8c0
   && memcmp(&(rip.entries[i].nexthop), &query_nexthop,
sizeof(uint32_t))) {
  rip.entries[i].metric++;
  RoutingTableEntry entry = toRoutingTableEntry(rip.entries[i],
query_if_index, 0);
  update(false, entry); // delete route entry
  deleted.push_back(entry);
}
```

4. 在更新路由表的时候，假如某路由表项的下一跳是0x00000000，则在更新至我们自己的路由表项的时候需要把该表项的下一跳设为这条路由表项的源地址。

```
if (rip.entries[i].nexthop == (uint32_t)0x00000000) {
  rip.entries[i].nexthop = src_addr;
}
```

5. 同时在更新路由表项的时候还需要注意对跳数的更新和判断。

```
if(tmp) {
  if (rip.entries[i].metric+1 <= query_metric) {
    rip.entries[i].metric++;
    update(true, toRoutingTableEntry(rip.entries[i], if_index,
0));
  }
} else {
  rip.entries[i].metric++;
  update(true, toRoutingTableEntry(rip.entries[i], if_index, 0));
// insert if rip is not found in routing table
}
```

转发

假如在自己路由表中找到的路由的下一跳是0，则把目的地址直接赋给它。

```
if (nexthop == 0) {
  nexthop = dst_addr;
}
```

组播

1. 实现水平分割，某一路由往自己学来的端口发路由项的时候，metric设为16实现reverse poisoning。
2. 若表项多过25条，则通过分成若干组25个来向外发送。
3. 同时发出的rip表项的每一个下一跳都设为0。

```
RipPacket ripPacket_o;
ripPacket_o.command = 2;
int j, kk;
for (j=routeIndex, kk = 0; j<(routers.size() < routeIndex+25 ?
routers.size() : routeIndex+25); j++, kk++) {
  if (routers.at(j).nexthop != 0 && routers.at(j).if_index == i) {
    ripPacket_o.entries[kk] = toRipEntry(routers.at(j), 16);
  } else if (routers.at(j).nexthop == 0) {
    ripPacket_o.entries[kk] = toRipEntry(routers.at(j), 1);
  } else {
    ripPacket_o.entries[kk] = toRipEntry(routers.at(j),
routers.at(j).metric);
  }
  ripPacket_o.entries[kk].nexthop = 0;
}
ripPacket_o.numEntries = kk;
```

性能

1. 不要用cout，不要用cout，不要用cout!
2. 原本是实现了在收到response之后再向外重新发出更新表项，但后来为了程序的稳定性，所以注释掉了，但应该会带来性能的提升。

搭建测试环境

以下是一些本人在测试过程中使用到的一些指令，用来搭建自己的虚拟网络。

```
sudo ip netns add net0 # 创建名为 "net0" 的 namespace
sudo ip netns add net1
sudo ip link add veth-net0 type veth peer name veth-net1 # 创建一对相互
连接的 veth pair
sudo ip link set veth-net0 netns net0 # 将 veth 一侧加入到一个 namespace
中
sudo ip link set veth-net1 netns net1 # 配置 veth 另一侧
```

```
sudo ip netns exec net0 ip link set veth-net0 up
sudo ip netns exec net0 ip addr add 10.1.1.1/24 dev veth-net0 # 给
veth 一侧配上 ip 地址
sudo ip netns exec net1 ip link set veth-net1 up
sudo ip netns exec net1 ip addr add 10.1.1.2/24 dev veth-net1

add direct route for PC1 & PC2:
sudo ip ro add default via 192.168.1.1 dev veth-PC1
sudo ip ro add default via 192.168.5.2 dev veth-PC2

ping test:
sudo ip netns exec net0 ping 10.1.1.2

bash:
sudo ip netns exec net0 bash

# bird:
bird -d -c bird_conf_R1.conf -P bird_R1.pid -s bird_R1.socket
bird -d -c bird_conf_R3.conf -P bird_R3.pid -s bird_R3.socket

# bird for linux virtual machine:
bird -d -c bird_conf_R1.conf -P /home/bird_R1.pid -s
/home/bird_R1.socket
bird -d -c bird_conf_R3.conf -P /home/bird_R3.pid -s
/home/bird_R3.socket

# Enable auto forwarding in linux:
echo 1 > /proc/sys/net/ipv4/conf/all/forwarding
sudo ip netns exec R1 sysctl -w net.ipv4.ip_forward=1

Delete auto forwarding in R2:
sudo ip a delete 192.168.3.2/24 dev eth1
sudo ip a delete 192.168.4.1/24 dev eth2

mkdir build
cd build
cmake .. -DBACKEND=Linux  # cmake .. -DHAL_PLATFORM_TESTING=Linux
make router_hal
make shell
cd Example
sudo ./shell

setup 5 links (PC-R-R-R-PC):
# setup namespaces
sudo ip netns add PC1
sudo ip netns add R1
sudo ip netns add R2
sudo ip netns add R3
sudo ip netns add PC2
```

```
# link namespaces by connecting ports
sudo ip link add veth-PC1 type veth peer name veth-R1-1
sudo ip link add veth-R1-2 type veth peer name eth1
sudo ip link add eth2 type veth peer name veth-R3-1
sudo ip link add veth-R3-2 type veth peer name veth-PC2

# connect ports to namespaces
sudo ip link set veth-PC1 netns PC1
sudo ip link set veth-R1-1 netns R1
sudo ip link set veth-R1-2 netns R1
sudo ip link set eth1 netns R2
sudo ip link set eth2 netns R2
sudo ip link set veth-R3-1 netns R3
sudo ip link set veth-R3-2 netns R3
sudo ip link set veth-PC2 netns PC2

# activate ports
sudo ip netns exec PC1 ip link set veth-PC1 up
sudo ip netns exec R1 ip link set veth-R1-1 up
sudo ip netns exec R1 ip link set veth-R1-2 up
sudo ip netns exec R2 ip link set eth1 up
sudo ip netns exec R2 ip link set eth2 up
sudo ip netns exec R3 ip link set veth-R3-1 up
sudo ip netns exec R3 ip link set veth-R3-2 up
sudo ip netns exec PC2 ip link set veth-PC2 up

# configurate IP addresses
sudo ip netns exec PC1 ip addr add 192.168.1.2/24 dev veth-PC1
sudo ip netns exec R1 ip addr add 192.168.1.1/24 dev veth-R1-1
sudo ip netns exec R1 ip addr add 192.168.3.1/24 dev veth-R1-2
sudo ip netns exec R2 ip addr add 192.168.3.2/24 dev eth1
sudo ip netns exec R2 ip addr add 192.168.4.1/24 dev eth2
sudo ip netns exec R3 ip addr add 192.168.4.2/24 dev veth-R3-1
sudo ip netns exec R3 ip addr add 192.168.5.2/24 dev veth-R3-2
sudo ip netns exec PC2 ip addr add 192.168.5.1/24 dev veth-PC2
```

组队测试的话，是通过使用三个树莓派和两台虚拟机，原理和上面的测试代码类似。