# 项目报告

目录

[项目报告 1](#_Toc503097222)

[Project-management-records： 1](#_Toc503097223)

[RIP部分 1](#_Toc503097224)

[Introduction: 1](#_Toc503097225)

[Design： 2](#_Toc503097226)

[Set-up： 3](#_Toc503097227)

[Centralized Routing 5](#_Toc503097228)

[原理 5](#_Toc503097229)

[设计和实现 5](#_Toc503097230)

[运行结果 6](#_Toc503097231)

[Contribution list 9](#_Toc503097232)

# Project-management-records：

见gitlab地址：

https://gitlab.com/liangtj/sysu\_virtual\_routing

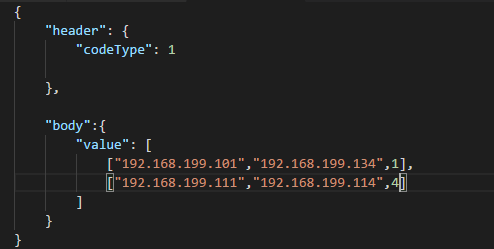
## RIP部分

### Introduction:

* RIP的运作流程见下：
* 1. 初始化：
  1. 在启动路由deamon Process时候，它先判断启动了哪些接口，并在每个接口上发送一个请求报文，要求其他路由发送完整的路由表（由于本次实验实验载体只能是没有在IP层启动路由功能的主机，而且本地主机也没有多个接口，出于简化实验细节考虑，就不判断启动的接口并且全接口发送）。在点对点链路中，发送给终点，在支持广播的网络中，则以广播的形式发送（本次实验因为是逻辑上建立拓扑，并没有因此不可以用广播，只能单播）。默认的UDP端口是520（本次实验用的是5005*）*
* 2. 接收到请求：
  1. 如果该请求是刚才所说的特殊请求，路由器就将本地完整路由表发送给请求者。否则，就处理请求中的每一个表项，更新本地路由表，同时如果本地路由有更新，立即向邻居发送本地路由表
* 3. 接收到响应：
  1. 更新路由表
* 4.定期发送选路更新：
  1. 每过30S就向相邻目标发送完整路由表（为了加快实验现象，目前设为3S）

### Design：

* RIP协议报文结构：



codeType的值是整形，范围是0-3，分别代表不同的情况：

QueryRoutesTable = 0 //初始化查询

AdvertiseRoutesTable = 1 //RIP通告报文

MockSendPacket = 2 // 模拟非RIP报文传递

Unknown = 3 // 未识别报文

body内的value则是路由表（查询报文，这里值为None），每一个表项都代表了一个RIP路由信息，每一个表项都从左到右含有目标主机（为了方便，所以取代目标网络），下一跳主机和度量

* 实现语言：

*python3.5*

* 实现思路：

*就如开头Inroduction所言*

* 关键点：

*关键在与RIP中路由环路，因此本人处理方案是如果收到报文中某一个路由metric是16，而且目标网络一致，下一跳也是发送报文的主机，则无条件将本地路由该项更新为16，表示无法达到主机网络*

* 细节：
* 1. 每一个主机既是服务端，也是客户端，有多少个连接的邻居主机，就有多少个客户端，每一个线程运行并维护一个客户端，一旦定期发送报文给目标主机出错，而且还是ICMP Reset报文（其实就一个异常捕捉。。。），则将本地路由中所有经过该邻居的目标主机的metirc设定为16.
* 2. 超时重传，当发送的报文超时，则重传，若一直超时次数过3次，则终止该进程，并视作邻居不可抵达，路由毒化。
* 3. socket编程时采用UDP，不用非实际的TCP。

### Set-up：

* 文件结构：
* main.py
* pkg/

*Packet/*

*RIPClient/*

*RIPServer/*

*RoutersTable/*

*Status/*

Deploy：

* 由于没有使用虚拟容器比如Docker，K8S等，因此拓扑网络有多少台主机，实际就要有多少台物理主机。并且每一个主机都要在sysu-virtual-routing/RIP目录下运行如下命令：
  1. *python main.py –r remote\_host\_ip\_list -h local\_host\_ip*

Result：

* TIP：
  1. *由于实在没有找到5台有python3，且注册了git账号，所以就只能用3台试验了。*
* 拓扑图：

- PC1:

192.168.199.101

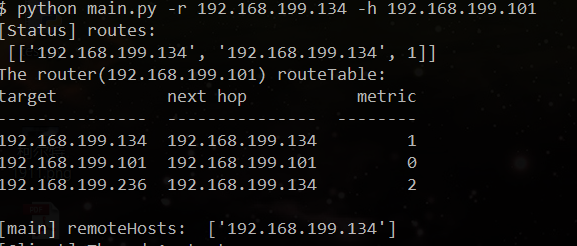
-PC2:

192.168.199.134

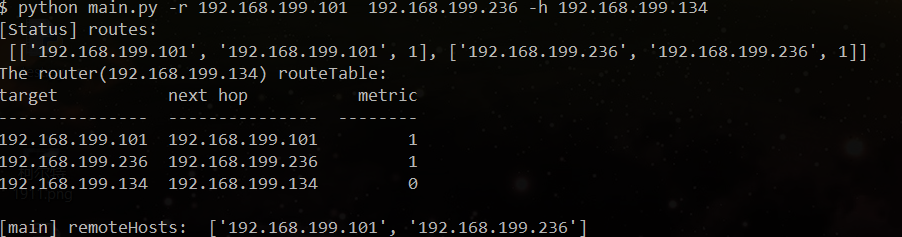
-PC3:

192.168.199.236

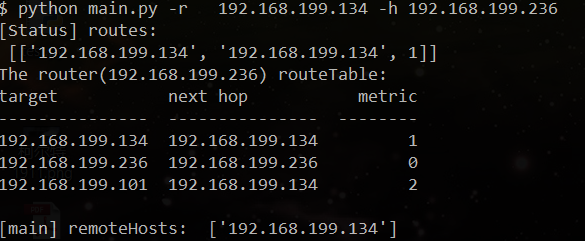
PC1：



PC2：



PC3：



## Centralized Routing

### 原理

中心化的虚拟路由，最核心的原理是Floyd算法。

服务器和节点之间通过socket传递信息，服务器根据各个节点发送给它的信息（直接连通的节点ip、到该节点的花费cost），构建一个图，然后通过算法算出最短路径，给每个节点发送其路由信息。节点根据这个路由信息，就会知道发往某个ip的数据包，下一跳节点是哪个（直连节点）。

### 设计和实现

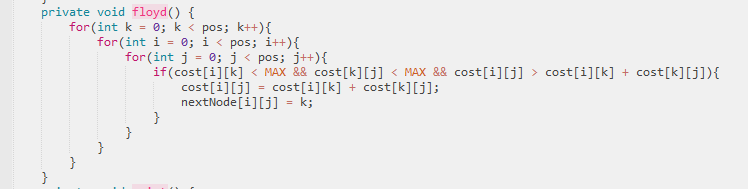
客户端：

初始化一个socket与服务端通信，告知服务端“我”和哪些主机直接连通，以及到这些直接连通的主机的代价，并且用另一个进程监听1234端口，接受服务端发回来的路由信息。

服务端：

用一个进程，监听1234端口，每当收到一个客户端发来的信息，就新建一个线程将这些信息加入本地数据中，并且运行Floyd算法计算下一跳矩阵和最短路径矩阵。算完之后，再将最新的路由信息发送给所有客户端。下一跳矩阵和最短路径矩阵保存了当前拓扑网络所有网络节点之间的转发情况。

Floyd算法部分：

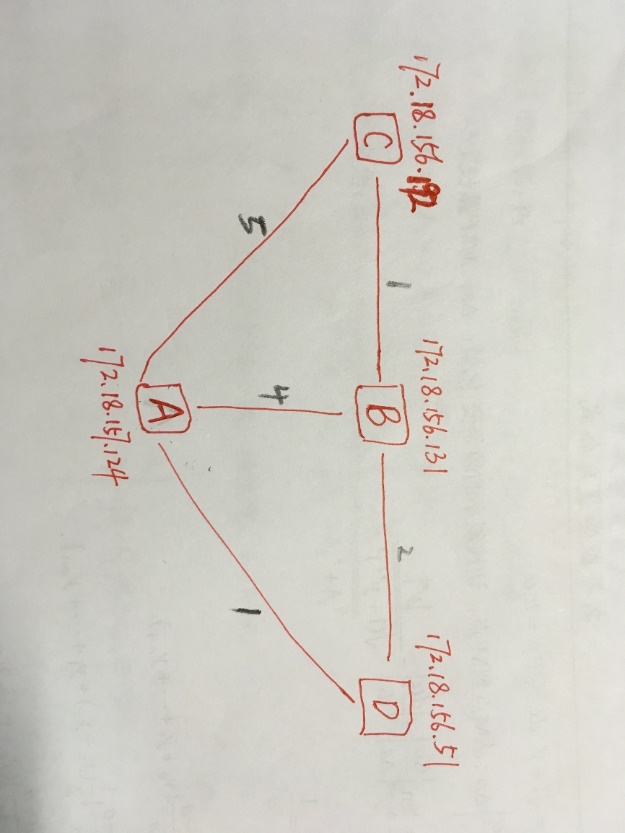


实现过程的主要困难是服务器和节点之间的通信，其中涉及多线程和socket编程。

具体实现请看代码，有很多注释。

### 运行结果

一开始采用的网络拓扑图是：



此时我们很容易看出最短路径的下一跳矩阵如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| A |  | D | D | D |
| B | D |  | C | D |
| C | A/B | A/B |  | B |
| D | A | B | B |  |

各个节点获取各自的路由信息。以节点B为例，节点B在配置完如上拓扑图后，从服务器获得的路由信息是：

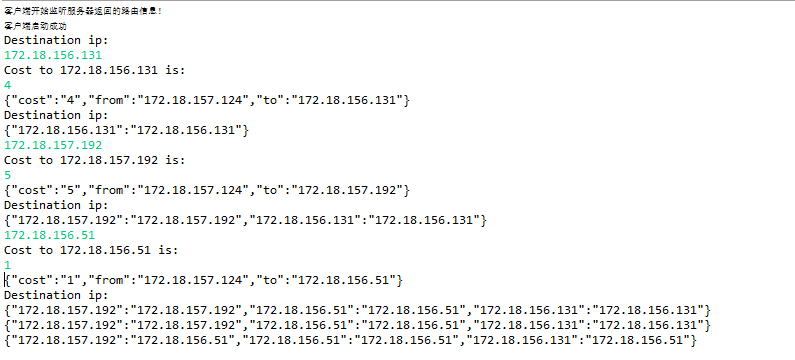


对应成主机编号就是：

|  |  |  |  |
| --- | --- | --- | --- |
| 目的节点 | A | C | D |
| B的下一跳节点 | D | C | D |

显然和上面的矩阵的第三行一致。

再以A节点为例，

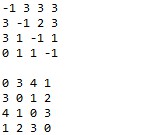


对应成主机编号就是：

|  |  |  |  |
| --- | --- | --- | --- |
| 目的节点 | B | C | D |
| A的下一跳节点 | D | D | D |

这也是跟上面的路由表符合的。

查看服务器的下一跳路由器矩阵，和最短代价矩阵，如下

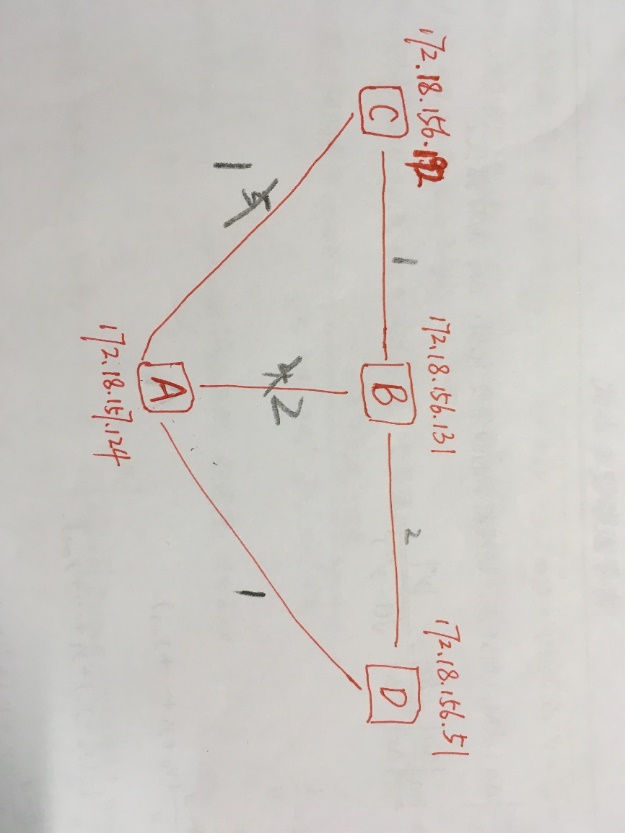


两个矩阵中，Matrix[i][j]]都表示从节点i到节点j，下标的对应关系是：0——A，1——B，2——C，3——D.

第一个矩阵是下一跳路由器矩阵，数字表示的含义：0——A，1——B，2——C，3——D，-1——自己。可以看到这个矩阵是和我们自己算出来的最佳路径中，下一跳路由器符合的。

第二个矩阵是最短路径代价矩阵，也是符合我们直观认识的。

现在我们来修改链路的状态。修改如下：



此时我们可以算出最短路径的下一跳矩阵是这样的：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| A |  | B | C | D |
| B | A |  | C | D |
| C | A | B |  | A |
| D | A | B | A |  |

以节点D为例，此时从服务器得到的路由信息是：

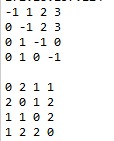


对应到主机编号就是：

|  |  |  |  |
| --- | --- | --- | --- |
| 目的节点 | A | B | C |
| D的下一跳节点 | A | B | A |

显然是和上面的矩阵第四行一样的。

此时服务器的下一跳节点矩阵和最短路径矩阵如下：



也可以看出，是符合我们自己算的结果的。

## Contribution list

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 学号 | 贡献 | 分数 |
| 梁丰 | 15331181 | 中心路由 | 100 |
| 梁天骏 | 15331184 | RIP部分 | 100 |
|  |  |  |  |