

计算机网络课程软件实验报告

王嘉禾 2193211079

同组成员：陈岩宇 2184214562

班级：计算机试验班 001

主要负责工作：实验一 server 部分编写

实验二 学习讨论和调试工作（代码共同完成）

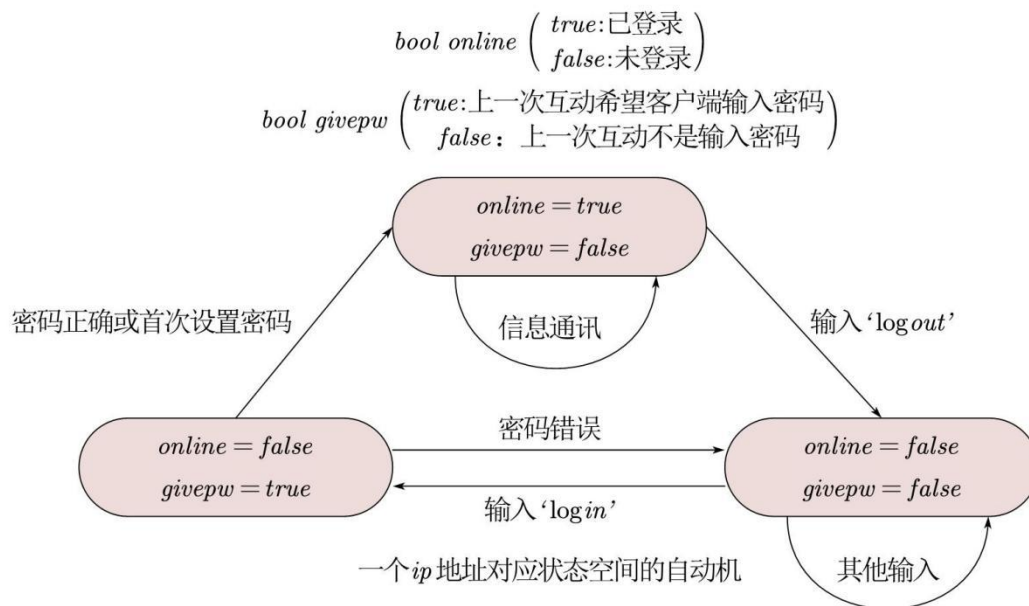
日期：2023/2/6

实验一：socket 网络编程

UDP 部分

实验目标：实现 UDP 连接下的服务器与客户端通信，实现服务器与多个客户端同时通信以及为客户端提供账户功能（包括设置密码，保存密码，登陆过程，登陆后信息通讯等）。

实现过程简述：由于 UDP 协议不存在稳定的信道连接，服务器无条件接受所有客户端发送的讯息，因此，无法采用每次连接后新建线程视为处理单独对象的操作方式，只能在软件层面处理。客户端发送时将信息绑定 ip 地址一起发送，例如 ip 为 127.0.0.1，发送消息“hello”，则发送的报文为“127.0.0.1|#|hello”。服务器中裁剪消息，获得 ip 地址和消息内容，采用一个状态自动机进行互动，每次遇见新的 ip 地址时新建一个信息保存空间，保存其 ip 地址，密码，当前账户状态（是否已登录，上一次互动是否要求客户端输入密码），采取这种方式可以通过查询该 ip 对应的状态知道下一步该如何互动。



代码片段

1. 服务器中对 ip 地址，报文，账户进行封装

```

typedef struct ipstr
{
    int x[4];
}ipstr;

typedef struct message
{
    ipstr ip;
    char content[512];
}message;

typedef struct account
{
    ipstr ip;
    char password[16];
    bool online;
    bool to_give_pw;
}account;

```

2. 服务器对收到的报文进行裁剪处理，获得 ip 地址和内容

```

message get_message(char* recv)
{
    int tmp=0,k=0,i;
    ipstr ip;
    char c,content[512];
    for(i=0;i<(int)strlen(recv);++i)
    {
        if(recv[i]=='|' && recv[i+1]=='#' && recv[i+2]=='|') break;
        c=recv[i];
        if(c=='.')
        {
            ip.x[k]=tmp;
            k++;
            tmp=0;
        }
        else
        {
            tmp=tmp*10+(int)c-48;
        }
    }
    ip.x[k]=tmp;
    if(i+3<strlen(recv)) strcpy(content,recv+i+3); else memset(content,0,sizeof(content));
    message ans;
    ans.ip=ip;
    memset(ans.content,'\0',sizeof(ans.content));
    strcpy(ans.content,content);
    return ans;
}

```

3. 服务器 socket 创建，绑定过程

```

// start
WSADATA wsaData;
flag=WSAStartup(MAKEWORD(2,2),&wsaData);
if (flag!=0)
{
    cout << "WSAStartup failed: %d\n" << flag;
    return 0;
}
// create socket
SOCKET serversocket=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if (serversocket==INVALID_SOCKET)
{
    cout << "socket create failed!\n";
    return 0;
}
// bind socket
SOCKADDR_IN addr;
ZeroMemory(&addr,sizeof(addr));
addr.sin_family=AF_INET;
addr.sin_addr.S_un.S_addr=inet_addr(myIP);
addr.sin_port=htons(myPort);
flag=bind(serversocket,(SOCKADDR*)&addr,sizeof(addr));
if(flag==1)
{
    cout << "bind failed!\n";
    return 0;
}
cout << "server ready.....\n";

```

4. 客户端连接并发送消息过程

```

// create socket
SOCKET clientsocket=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

SOCKADDR_IN addr;
ZeroMemory(&addr,sizeof(addr));
addr.sin_family=AF_INET;
addr.sin_addr.S_un.S_addr=inet_addr(serverIP);
addr.sin_port=htons(serverPort);
int server_len=sizeof(addr);

//communicate
while(1)
{
    getline(cin,tmp_string);
    tmp_string=myIPstr+tmp_string;
    if(tmp_string==myIPstr+"quit") break;

    memset(send_data,'\0',sizeof(send_data));
    tmp_string.copy(send_data,tmp_string.length()+1);

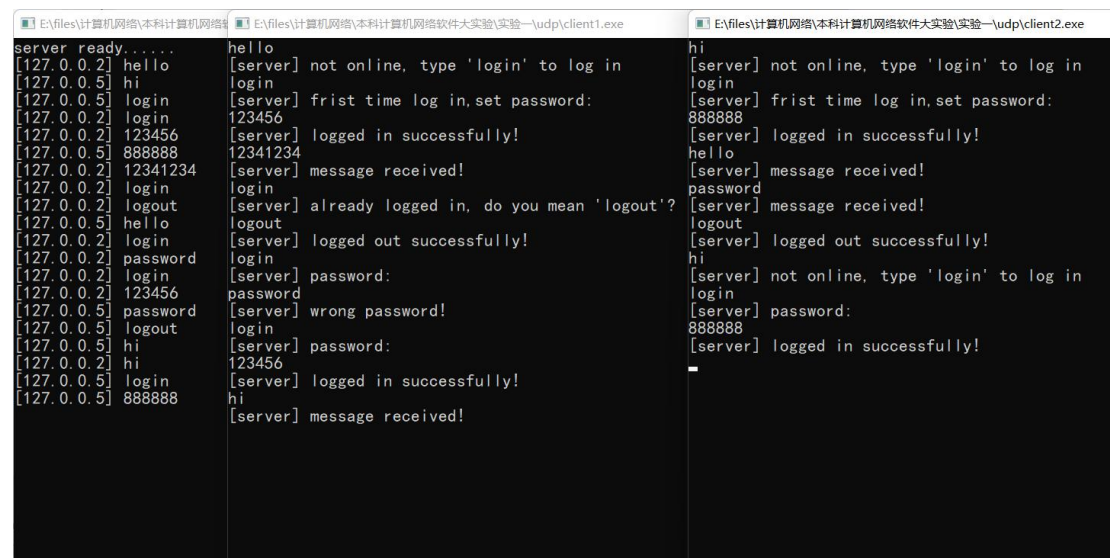
    flag=sendto(clientsocket,send_data,sizeof(send_data),0,(SOCKADDR*)&addr,sizeof(addr));
    if (!flag)
    {
        cout << "fail to send!\n";
    }

    memset(recvd_data,'\0',sizeof(recvd_data));
    flag=recvfrom(clientsocket,recvd_data,512,0,(SOCKADDR*)&addr,&server_len);
    if(flag!=-1) cout << "[server] " << recvd_data << endl;
}

```

完整代码见 [udp/server.cpp](#) 和 [udp/client1.cpp](#)

实验结果测试：



```
server ready,.....
[127.0.0.2] hello
[127.0.0.5] hi
[127.0.0.5] login
[127.0.0.2] login
[127.0.0.2] 123456
[127.0.0.5] 888888
[127.0.0.2] 12341234
[127.0.0.2] login
[127.0.0.2] logout
[127.0.0.5] hello
[127.0.0.2] login
[127.0.0.2] password
[127.0.0.2] login
[127.0.0.2] 123456
[127.0.0.5] password
[127.0.0.5] logout
[127.0.0.5] hi
[127.0.0.2] hi
[127.0.0.5] login
[127.0.0.5] 888888

[server] not online, type 'login' to log in
login
[server] frist time log in,set password:
123456
[server] logged in successfully!
12341234
[server] message received!
login
[server] already logged in, do you mean 'logout'?
logout
[server] logged out successfully!
login
[server] password:
password
[server] wrong password!
login
[server] password:
123456
[server] logged in successfully!
hi
[server] message received!

hi
[server] not online, type 'login' to log in
login
[server] password:
888888
[server] logged in successfully!
```

两个客户端可实现分别互动，测试成功。

TCP 部分

实验目标：实现 TCP 连接下的服务器与客户端通信，实现服务器与多个客户端同时通信以及为客户端提供账户功能（包括设置密码，保存密码，登陆过程，登陆后信息通讯等）。

实现过程简述：TCP 与 UDP 不同，有稳定的信道连接，每次建立连接在服务器进行 listen 过程，客户端进行 connect 过程和服务器 accept 连接之后进行通讯，在服务器端，可建立多个线程，每个线程对特定的客户端 socket 进行服务。并且，可以通过对 socket 的存储记录下连接的客户端个数，对特定的 socket 一直接受消息也可以及时检测客户端是否断开，这些操作是 udp 的机制无法实现的，传输信息的规则以及对账户状态的处理与 UDP 实验中一致。

代码片段：（通信互动过程与 UDP 中类似，不过多展示）

1. 服务器创建 socket 并绑定过程

```

// start
WSADATA wsaData;
flag=WSAStartup(MAKEWORD(2,2),&wsaData);
if (flag!=0)
{
    cout << "WSAStartup failed: %d\n" << flag;
    return 0;
}

// create socket
SOCKET serversocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
//SOCKET serversocket=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

if (serversocket==INVALID_SOCKET)
{
    cout << "socket create failed!\n";
    return 0;
}

// bind socket
SOCKADDR_IN addr;
ZeroMemory(&addr,sizeof(addr));
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=inet_addr(myIP);
//addr.sin_addr.S_un.S_addr=inet_addr(myIP);
addr.sin_port=htons(myPort);
flag=bind(serversocket,(SOCKADDR*)&addr,sizeof(addr));
if(flag==-1)
{
    cout << "bind failed!\n";
    return 0;
}
cout << "server ready.....\n";

```

2. 在端口上监听并同意连接，创建新线程的过程

```

while(1)
{
    if (listen(serversocket,20)!=-1)
    {
        //cout << "waiting for connection.....\n";
        SOCKADDR clientaddr;
        int size=sizeof(clientaddr);
        SOCKET clientsocket=accept(serversocket,&clientaddr,&size);

        // search the account
        vector<SOCKET>::iterator itr = find(client_list.begin(),client_list.end(),clientsocket);
        if (itr==client_list.end()) client_list.push_back(clientsocket);

        DWORD dwPingThreadID;
        //创建新线程处理该客户socket连接
        HANDLE hPingHandle=CreateThread(0,0,server_thread,(LPVOID)clientsocket,0,&dwPingThreadID);
    }
}

```

3. 客户端请求连接的过程


```

// start
WSADATA wsaData;
int flag;
flag=WSAStartup(MAKEWORD(2,2),&wsaData);
if (flag!=0)
{
    cout << "WSAStartup failed: %d\n" << flag;
    return 0;
}

// create socket
SOCKET serversocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
//SOCKET serversocket=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

// bind serversocket
SOCKADDR_IN addr;
ZeroMemory(&addr,sizeof(addr));
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=inet_addr(serverIP);
//addr.sin_addr.s_un.s_addr=inet_addr(serverIP);
addr.sin_port=htons(serverPort);
bind(serversocket,(SOCKADDR*)&addr,sizeof(addr));

//connect
connect(serversocket,(SOCKADDR*)&addr,sizeof(addr));

memset(send_data,'\0',sizeof(send_data));
strcpy(send_data,myIP);
send(serversocket,send_data,(int)strlen(send_data),0);

//communicate
memset(recvd_data,'\0',sizeof(recvd_data));
recv(serversocket,recvd_data,512,0);
cout << "[server] " << recvd_data << endl;

```

完整代码见 [tcp/server.cpp](#) 和 [tcp/client1.cpp](#)

测试结果:

```

E:\files\计算机网络\本科计算机网络软件大实验\实验一\tcp\server(muti thread).exe
server ready.....
receive a connection from [127.0.0.5] recent connection: 1
receive a connection from [127.0.0.2] recent connection: 2
lose connection from [127.0.0.2] recent connection: 1
receive a connection from [127.0.0.2] recent connection: 2
[127.0.0.2] hello
[127.0.0.2] login
[127.0.0.5] hi
[127.0.0.2] 1234
[127.0.0.2] here is 2
[127.0.0.5] login
[127.0.0.5] 123456
[127.0.0.5] quit
lose connection form [127.0.0.5] recent connection: 1
[127.0.0.2] logout

E:\files\计算机网络\本科计算机网络软件大实验\实验一\tcp\client1.exe
[server] server connected
hello
[server] not online, type 'login' to log in
login
[server] frist time log in,set password:
1234
[server] logged in successfully!
here is 2
[server] message received!
logout
[server] logged out successfully!

E:\files\计算机网络\本科计算机网络软件大实验\实验一\tcp\client2.exe
[server] server connected
hi
[server] not online, type 'login' to log in
login
[server] frist time log in,set password:
123456
[server] logged in successfully!
quit

Process exited after 282.7 seconds with return value 0
请按任意键继续. . .

```

同样，两个客户端可并行运作，分别对账户进行互动，且客户端而进程结束时可被服务器检测，测试成功。

实验 2: RIP 路由协议实验

实验目标 实现距离矢量法（Distance Vector Routing）路由协议

实验过程简述

1. 每个路由器维护一个路由表，路由表的每一项记录从该路由器到某一个终端的最短路径的下一条对应的端口。
2. 路由表需要不断迭代更新和维护。
3. 初始时，所有路由器仅记录和自己直接相连的终端的信息，设置到达该终端的最短路径的下一条为他们连接的端口。（大多数情况下这就是正确的最短路径）
4. 每隔一定时间（实验中设置为 5s）所有路由器将自己的路由表传送给相邻的路由器，每隔路由器根据自己周围的其他路由器的路由表，结合自己与该路由器的传输代价，判断是否存在代价更小的路线（例如对 A 路由器，与 B 路由器相连，讨论传到终端 h 的路径，若 A 先将数据包传送到 B，在通过 B 记录的最优路径传输，总代价小于从 A 记录的最优路径，则将 A 到 h 的最优路径更新为下一跳跳往 B，即更新为 A 与 B 连接的对应端口）
5. 为应对网络的拓扑结构变化或路径代价变化造成的最优路径变化（甚至是可达性的变化），在协议中还要采取老化（实验中设置为每条路径 15s 后失效），毒化，毒性反转等算法。

实验代码片段：

拓扑结构创建

```
import sim
def launch (switch_type = sim.config.default_switch_type, host_type = sim.config.default_host_type):
    switch_type.create('A')
    switch_type.create('B')
    switch_type.create('C')
    switch_type.create('D')
    host_type.create('h1')
    host_type.create('h2')
    host_type.create('h3')
    host_type.create('h4')
    h1.linkTo(A, latency=1)
    h2.linkTo(B, latency=1)
    h3.linkTo(C, latency=1)
    h4.linkTo(D, latency=1)
    A.linkTo(B, latency=2)
    A.linkTo(C, latency=7)
    B.linkTo(C, latency=1)
    D.linkTo(B, latency=3)
    D.linkTo(C, latency=1)
```

路由算法实现

函数功能简述：

add_static_route(): 为与路由器直接相连的中断创建路由表项目。。

handle_data_packet(): 传输数据包，调用路由表项目找到对目的地最优路径的端口。

Send_routes(): 将自己的路由表转发给相邻的路由器，这里采用了水平分割或毒性反转算法，能避免环路和更加快速的收敛。具体方法为当路由器将路由表的一项发送至一个邻居路由器时，判断自己发送的这“最优路径”是否经过该路由西（判断下一跳端口是否是与该路由器自己连接的端口即可），若是，则对于水平分割算法，不做处理，对于毒性反转算法，将代价设置为无穷大并发送。

expire_routes(): 路由表中的所有项目都在一定时间后超时，将超时的项目删除。

handle_route_advertisement(): 处理从邻居路由器获得的路由表，对于每一项，判断先传送给该邻居在通过邻居记录的最优路径传送是否优于自己记录的最优路径，若是，则进行表项的更新，具体做法为将下一跳的端口修改为与该邻居连接的端口。


```

def add_static_route(self, host, port):
    assert port in self.ports.get_all_ports(), "Link should be up, but is not."
    self.table[host] = TableEntry(dst = host, port = port, latency = self.ports.get_latency(port), expire_time = FOREVER)

def handle_data_packet(self, packet, in_port):
    dst = packet.dst
    if (dst in self.table and self.table[dst].latency < INFINITY):
        self.send(packet, self.table[dst].port)

def send_routes(self, force=False, single_port=None):
    if (not self.SPLIT_HORIZON and not self.POISON_REVERSE):
        for port in self.ports.link_to_lat.keys():
            for host, entry in self.table.items():
                self.send_route(port, host, entry.latency)
    else:
        for port in self.ports.link_to_lat.keys():
            for host, entry in self.table.items():
                if (entry.port != port):
                    self.send_route(port, host, entry.latency)
                else:
                    if (self.POISON_REVERSE):
                        self.send_route(port, host, INFINITY)

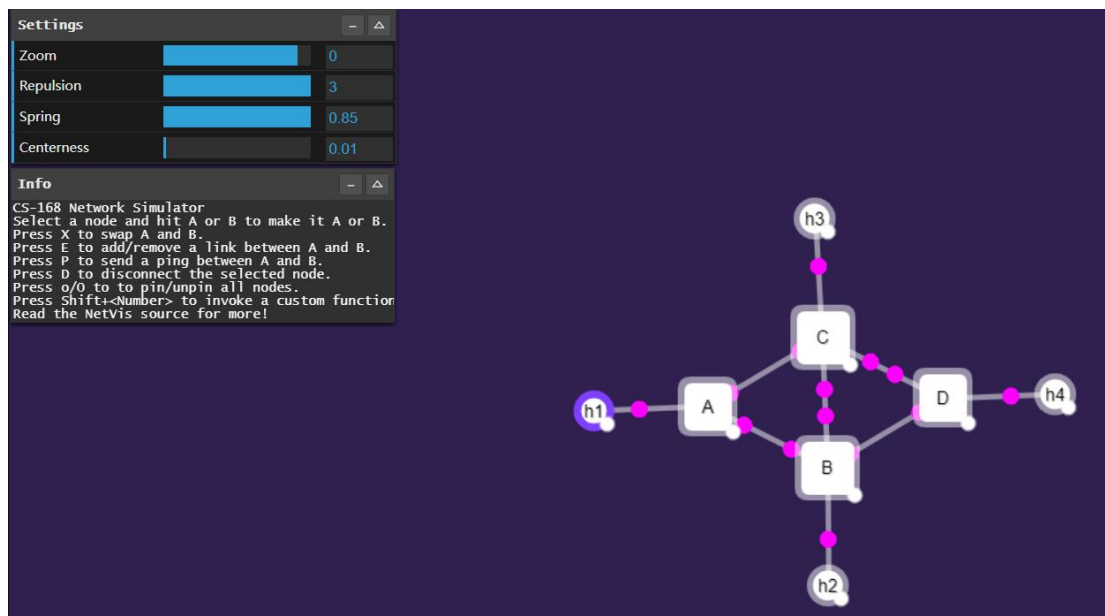
def expire_routes(self):
    expire_list = []
    for host, entry in self.table.items():
        if (api.current_time() >= entry.expire_time):
            expire_list.append(host)
    for host in expire_list:
        del(self.table[host])

def handle_route_advertisement(self, route_dst, route_latency, port):
    if (route_dst not in self.table):
        self.table[route_dst] = TableEntry(dst = route_dst, port = port, latency = route_latency + self.ports.get_latency(port), expire_time = api.current_time() + self.ROUTE_TTL)
    else:
        if (port == self.table[route_dst].port or self.table[route_dst].latency > route_latency + self.ports.get_latency(port)):
            self.table[route_dst] = TableEntry(dst = route_dst, port = port, latency = route_latency + self.ports.get_latency(port), expire_time = api.current_time() + self.ROUTE_TTL)

```

完整代码见 `dv_router.py`

实验结果



在可视化界面可以看到路由器之间传送数据包的过程，每 5 秒进行一次路由表的更新。

```

INFO:simulator:Launching module 'topos.experiment'
INFO:simulator:A up!
INFO:simulator:B up!
INFO:simulator:C up!
INFO:simulator:D up!
INFO:simulator:h1 up!
INFO:simulator:h2 up!
INFO:simulator:h3 up!
INFO:simulator:h4 up!
INFO:simulator:Starting simulation.
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> h1.ping(h4) DEBUG:user:h4:rx: <Ping h1->h4 ttl:15> A,B,C,D,h4
DEBUG:user:h1:rx: <Pong <Ping h1->h4 ttl:15>> D,C,B,A,h1

```

刚启动网络时，h1 ping h4 没有反应，从可视化界面也可以看出数据包传送图中就已经消失，因为路由表没有迭代完成，还没有收集到所有的可达路径，而一段时间后，请求报文发送成功，说明路由表已经找到 h1 到 h4 的路径（在本实验中已经迭代完成）。

```
>>> B.unlinkTo(C)
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> h1.ping(h4)
>>> DEBUG:user:h4:rx: <Ping h1->h4 ttl:16> A,B,D,h4
DEBUG:user:h1:rx: <Pong <Ping h1->h4 ttl:16>> D,B,A,h1
```

将 B 路由器与 C 路由器之间的连接中断，再用 h1 向 h4 发送请求报文，可以看到开始时无法 ping 通，并且通过可视化界面看到 h1 的请求报文到达 B 路由器后就消失，说明 B 的路由表没有更新，依然是指向 C，但是此时连接中断，显然无法发送。一段时间后，B 的该表项超时，重新进行迭代，从路由器 D 出找到了通向 h4 的路径。

```
>>> print(A.table, '\n', B.table, '\n', C.table, '\n', D.table)
=== Table for A ===
name  prt  lat  sec
-----
h1     0   1   inf
=== Table for B ===
name  prt  lat  sec
-----
h2     0   1   inf
=== Table for C ===
name  prt  lat  sec
-----
h3     0   1   inf
=== Table for D ===
name  prt  lat  sec
-----
h4     0   1   inf
```

```
>>> print(A.table, '\n', B.table, '\n', C.table, '\n', D.table)
=== Table for A ===
name  prt  lat  sec
-----
h1     0   1   inf
=== Table for B ===
name  prt  lat  sec
-----
h2     0   1   inf
h3     2   2  14.63
=== Table for C ===
name  prt  lat  sec
-----
h3     0   1   inf
h2     2   2  14.63
h4     3   2  14.63
=== Table for D ===
name  prt  lat  sec
-----
h4     0   1   inf
h3     2   2  14.63
```

```
>>> print(A.table,'\n',B.table,'\n',C.table,'\n',D.table)
=== Table for A ===
name  prt lat  sec
-----
h1     0   1   inf
h2     1   3  14.60
=== Table for B ===
name  prt lat  sec
-----
h2     0   1   inf
h3     2   2  13.59
h1     1   3  14.60
=== Table for C ===
name  prt lat  sec
-----
h3     0   1   inf
h2     2   2  13.59
h4     3   2  13.59
=== Table for D ===
name  prt lat  sec
-----
h4     0   1   inf
h3     2   2  13.59
```

```
>>> print(A.table,'\n',B.table,'\n',C.table,'\n',D.table)
=== Table for A ===
name  prt lat  sec
-----
h1     0   1   inf
h2     1   3  13.55
=== Table for B ===
name  prt lat  sec
-----
h2     0   1   inf
h3     2   2  12.54
h1     1   3  13.55
h4     3   4  14.54
=== Table for C ===
name  prt lat  sec
-----
h3     0   1   inf
h2     2   2  12.54
h4     3   2  12.54
=== Table for D ===
name  prt lat  sec
-----
h4     0   1   inf
h3     2   2  12.53
h2     1   4  14.53
```

以上四张图是网络启动时路由表的迭代过程，与算法设计相符。

拓展分析

关于步骤六（水平分割）和步骤七（毒性逆转）的分析

待解决的问题:当网络中有连接断开时，实际上有相当一部分路由器中会有对应的表项失效。但在前五步的方法中，第一时间只有该连接两端的路由器会出现路由器中表项超时失效的情况。因此，他们会错误地从邻居那里学习到到达连接另一端的路径，但实际上这条路径可能是

通刚才断开的连接到达的，一段时间后，邻居的相应表项也超时，从该结点处获取了新的路径，如此往复迭代，错误的路径会一直存在并且存储的“代价”项越来越高。这个问题称为“无穷计算问题”，如下图所示（AB 断开后的情景）。



水平分割的思路：路由器不会将表项发给通过该路径的邻居。例如在上图的例子中，AB 断开后，B 的路由表中到达 A 的一项超时，它从 C 处获知 C 到达 A 的代价为 2，因此它将自己到 A 的代价更新为 1，并且下一跳为 C。造成这种情况的本质是因为 C 存储的到 A 的路径实际上是通过 B 的，当 AB 断开后，C 的这个表项也应当失效，因此 C 不应把该表项发给 B 用于更新，C 和 D，D 和 E 之间的情况同理。

毒性逆转的思路：毒性逆转实际上是一种更为主动的水平分割。路由器从某个接口上接收到某个网段的路由信息之后，并不是不往回发送信息了，而是将这个网段标志为不可达，再发送出去。收到此种的路由信息后，接收方路由器会立刻抛弃该路由，而不是等待其老化时间到。这样可以加速路由的收敛。

后者的提升本质：对水平分割理解可以表达为：既然你告诉了我到达某一点的信息，那我就不用再告诉你了。而对毒性逆转的理解而可以表达为：既然你告诉了我到达某一点的信息，那么我就告诉你不能从我这走。毒性逆转在避免环路方面要比水平分割更加靠谱。

测试结果

```
C:\WINDOWS\system32\cmd.exe
RouteAd(dst=h2, latency=6)
MISMATCH detected for route advertisements sent to port 2:
MISSING route advertisement:
RouteAd(dst=h2, latency=6)
MISMATCH detected for route advertisements sent to port 10:
MISSING route advertisement:
RouteAd(dst=h2, latency=6)
-----
Ran 31 tests in 0.045s
FAILED (failures=21)
Overall scores:
Stage 1 TestStaticRoutes           : 1 / 1 passed
Stage 2 TestForwarding              : 4 / 4 passed
Stage 3 TestAdvertise               : 1 / 1 passed
Stage 4 TestHandleAdvertisement     : 8 / 8 passed
Stage 5 TestRemoveRoutes            : 4 / 4 passed
Stage 6 TestSplitHorizon            : 1 / 1 passed
Stage 7 TestPoisonReverse            : 1 / 1 passed
Stage 8 TestInfiniteLoops           : 2 / 3 passed (1 FAILED)
Stage 9 TestRoutePoisoning          : 0 / 5 passed (5 FAILED)
Stage 10 TestTriggeredIncrementalUpdates : 10 / 31 passed (21 FAILED)
Total score: 79.89 / 100.00
E:\files\计算机网络\本科计算机网络软件大实验\实验二\cs168_proj_routing_student-mast
```