基于UDP和IP的RTT简单获取实现

1. **前言**
   1. **UDP**

用户数据报协议(User Datagram Protocol, UDP)，是开放式系统互联(Open System Interconnection, OSI)参考模型中的一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

UDP提供的不可靠、无连接的数据交付服务，主要指的是它没有使用确认机制来确保报文的到达，没有对传入的报文进行排序，也不提供反馈信息来控制机器之间报文传输的速度。因此，UDP报文可能会出现丢失、延迟或乱序到达等现象。而且报文到达的速率可能大于接受进程能够处理的速率。UDP数据报报文格式很简单，分为UDP首部和UDP数据区两个部分。其中UDP首部有4个域组成，其中每个域各占用2个字节，具体包括源端口号、目的端口号、数据报长度和校验值。UDP首部总共只占用8个字节。

单从可靠性的角度看，TCP(Transmission Control Protocol)优于UDP，但是UDP也正正因为不需要考虑可靠性机制，所以UDP的传输处理效率要远优于TCP。所以在对可靠性要求不高或者要求传输效率的时候，UDP是一种适合的方案。

* 1. **IP**

互联网协议(Internet Protocol, IP)是IP层的核心协议，它提供无连接的IP数据报投递服务,无论传输层使用何种协议，都要依靠IP来发送和接收数据。

IP的设计目的是为了提高网络的可扩展性：一是解决互联网问题，实现大规模、异构网络的互联互通；二是分割顶层网络应用和底层网络技术之间的

耦合关系，以利于两者的独立发展。

IP是一种只为主机提供无连接、不可靠、尽力而为的数据报传输服务。

1. 提供一种无连接的投递机制。IP独立地对待要传输的每一个数据报，在传输前不建立连接，从同一源主机到同一目的主机的数据报可能经过不同的传输路径。

2. 不保证数据报的可靠性。数据报在传输过程中可能出现丢失、重复、延迟和乱序，但IP不会将这些现象报告给发送方和接收方，也不会试图去纠正传输中的错误。

3. 提供了尽最大努力的投递机制。IP尽最大努力发送数据报，也就是说，它不会随意放弃数据报，只有当资源耗尽或者底层网络出现故障时，才会出现数据报丢失的情况

* 1. **RTT**

往返时延(Round-Trip Time, RTT)是一个重要的性能指标，它表示从发送方发送数据开始，到发送方收到来自接收方的确认，总共经历的时延。

往返时延RTT由三个部分决定：即链路的传播时间、末端系统的处理时间以及路由器的缓存中的排队和处理时间。其中前两个时间相对固定，路由器的缓存

中的排队和处理时间会随着整个网络拥塞程度的变化而变化，因此往返时延一定程度上反映着网络的状况。

* 1. **路由转发**

路由器仅根据网络地址进行转发。当IP数据报经由路由器转发时，如果目标网络与本地路由器直接相连，则直接将数据包交付给目标主机；否则，路由器通过路由表产找路由信息，并将数据报转交给指明的下一跳路由器；如果没有路由，但路由表中哟一个默认路由，则把数据报传送给指明的默认路由；如果两者都没有，则丢弃数据包并报告错误。

1. **仿真实现**
   1. **目的**

本文目的是得到通信两端的往返时延RTT。

* 1. **原理**

本文原理是在现有的网路层和传输层的基础上传输定制的报文信息。将时间信息存储在报文中，将获取RTT等任务分配到每一个具体的报文传输中去，有效地减少客户端的存储压力。（首先本文假设考虑每一台虚拟机上的时间是一致的，都是通过网络一致的时间，也就是统一时钟）。

本文基于所构建的网络环境比较简单，没有复杂的网络拓扑，传输的数据量也比较少，一般不会造成网络拥塞现象。因此考虑两端的往返时间RTT = 2 \* T来表示，其中T为单向的时延。

* 1. **实现**
     1. **环境部署**

将网络环境配置成如下拓扑图，如图1

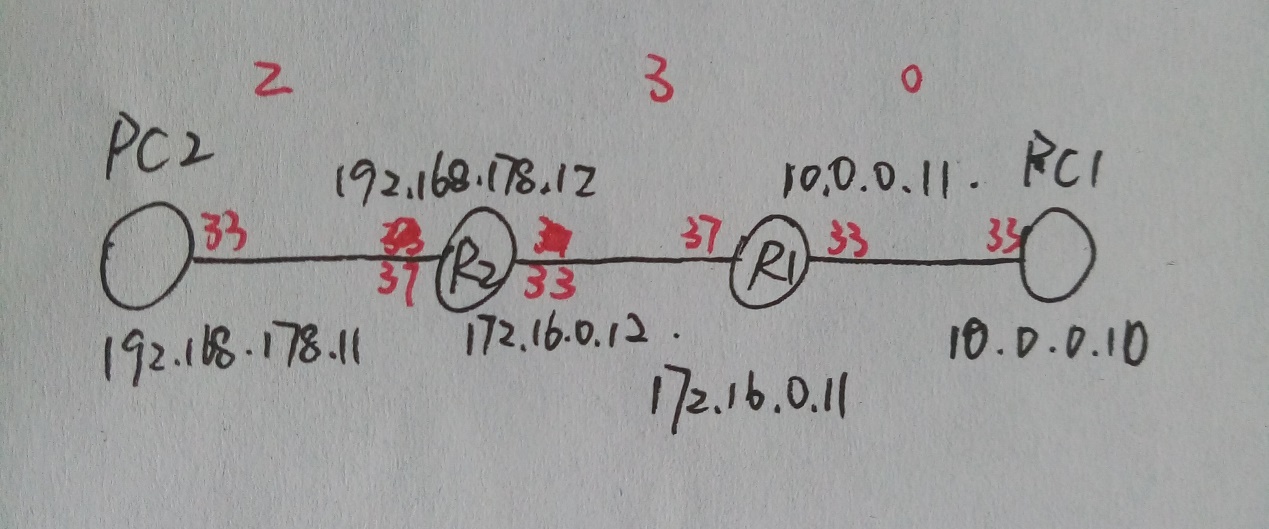


图 1

其中每一个节点都是Centos7虚拟机，将四台虚拟机首尾相连，构建成三个虚拟内网。接下来通过配置中间两台虚拟机，使其变为具有转发功能的路由器（以下称为路由器）。然后配置对应的IP地址和对应的默认网关，最终实现主机和路由器互联。

1. **首尾相连构建不同的虚拟内网，如图2**



图 2

1. **中间两台虚拟机配置成具有转发功能的路由器**

|  |
| --- |
| # 对于R1,R2一样配置 # 先关闭防火墙 iptables -F  # 将默认关闭的路由转发功能打开 echo "1" > /proc/sys/net/ipv4/ip\_forword |

1. **添加虚拟网卡，配置相应的ip地址**

其中R1和R2为两个网卡，并且配置如拓扑图的ip地址信息，构成多个内网模拟现实状态。（其中使用的VMware中ens33或者ens37接口）

1. **配置两台主机的默认网关，配置两台路由器的路由信息，如图3，4，5，6**

|  |
| --- |
| # R1添加路由信息 # 去往192.168.178.0/24段的路由 route add -net 192.168.178.0/24 gw 172.16.0.12 dev ens33 |

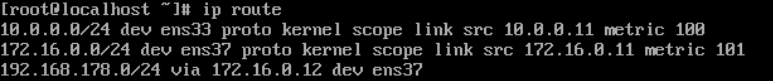


图 3

|  |
| --- |
| # R2添加路由信息 # 去往10.0.0.0/24段的路由 route add -net 10.0.0.0/24 gw 172.16.0.11 dev ens33 |

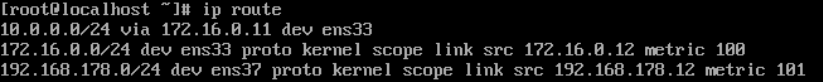


图 4

|  |
| --- |
| # PC1添加默认网关 route add -net default gw 10.0.0.11 |

route-PC1

图 5

|  |
| --- |
| # PC2添加默认网关 route add -net default gw 192.168.178.12 |

route-PC2

图 6

1. **测试两台主机的连通性，如图7**

|  |
| --- |
| # PC1 ping PC2 ping 192.168.178.11 |

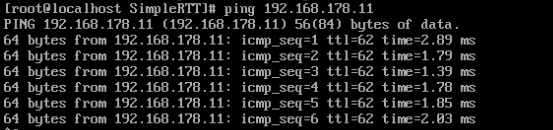


图 7

* + 1. **代码实现**
* **定义报文格式**

因为本报文是定义在传输层之上的，因此报文格式不需要对报文内容的每一项进行长度大小的限定。报文应该由五项组成，分别是源IP地址、目的IP地址、报文协议命令、单向传输开始时间戳和单向传输结束时间戳，其中使用12321端口号作为默认端口。源IP地址是客户端的IP地址；目的IP地址是服务器的IP地址；报文协议命令由SYN和ACK两个命令构成，其中SYN是由客户端向服务器发送请求时使用的协议命令，而ACK时服务器对请求应答时携带的命令；单向开始传输时间戳是客户端发送请求报文时填充的时间节点；而单向传输结束时间戳是服务器方接收到SYN请求报文立即进行应答时填充的时间节点。通过客户端发送SYN请求，到最后接收到ACK应答，最终拆开ACK报文，将其中的单向传输结束时间戳减掉单向传输结束时间戳就可以得到单向传输时延T，最后得到RTT。

报文格式如下。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 源IP地址 | 目的IP地址 | 报文协议命令 | 单向传输开始时间戳 | 单向传输结束时间戳 |

报文实现如下

|  |
| --- |
| 1. **class** Mess(object): 2. """The message transfered between client and server""" 3. # some order of Message 4. # SYN, ACK 5. SYN = "SYN" 6. ACK = "ACK" 8. **def** \_\_init\_\_(self, src\_ip, dst\_ip, order=SYN, startT=time.time(), endT=None): 9. self.src\_ip = src\_ip    # source ip 10. self.dst\_ip = dst\_ip    # destination ip 11. self.order = order      # mess order 12. self.startT = startT    # start time 13. self.endT = endT        # end time |

* **服务器实现**

服务器需要做的就是对到来的请求报文进行解析并且提取报文的内容再进行填充单向传输结束时间戳，最后向客户端进行应答。实现代码如下。

|  |
| --- |
| 1. **class** Server(object): 2. """A simple server to answer the request from the client""" 3. **def** \_\_init\_\_(self, port=PORT): 4. self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) 5. self.sock.bind((SERVER, port)) 6. self.start\_serve() 8. **def** start\_serve(self): 9. **while** True: 10. data, address = self.sock.recvfrom(1024) 11. mess = Mess.from\_str(data.decode('utf-8')) 12. mess.src\_ip, mess.dst\_ip = mess.dst\_ip, mess.src\_ip 13. mess.order = Mess.ACK 14. mess.endT = time.time() 15. self.sock.sendto(mess.\_\_str\_\_().encode('utf-8'), address) |

* **客户端实现**

客户端需要向服务器发送请求并且填入相应的数据和单向传输开始时间戳。代码实现如下。

|  |
| --- |
| 1. **class** Client(object): 2. """A simple client to send the mess to server""" 3. **def** \_\_init\_\_(self, port=PORT): 4. self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) 5. self.sock.connect((SERVER, PORT)) 6. **while** True: 7. mess = Mess(CLIENT, SERVER) 8. self.sock.sendall(mess.\_\_str\_\_().encode('utf-8')) 9. data = self.sock.recv(1024) 10. ack\_mess = Mess.from\_str(data.decode('utf-8')) 11. **print**(ack\_mess) 12. **print**("start time: ", ack\_mess.startT) 13. **print**("end time: ", ack\_mess.endT) 14. **print**("RTT: ", ack\_mess.endT - ack\_mess.endT) 15. **break** |

* + 1. **结果**

通过本文中构建的简单网络拓扑测试本算法实现的RTT获取，得到上述网络拓扑两端主机的往返时延结果为

RTT = 2 \* （end time – start time）= 0.8548140毫秒

如下图8.



图 8

本文通过配置中间两台虚拟机作为路由器用作转发工具，然后成功实现基于网络层和传输层之上的定制化的报文传输。

但是存在一个问题是中间的两台路由器是不会向我们发送方传输我们定制化的报文的，如果要实现这个功能，我们需要在本实验的基础上实现一个在传输层之上的一个新的传输层协议来控制中间两台路由器的定制化的报文的传输。 因为考虑到基于网络层的转发功能是不会拆解传输层的报文的，而我们在当前情况只能在网络层传输层之上进行编程，所以如果要实现新增的传输层协议，我们不能仅仅依靠网络层的传输功能，而且需要新增的传输层进行拆包查看里面的内容再进行控制转发。这样每一次的转发都要先经过传输层，具体测试出来的RTT会变大。

1. **结论与分析**

本文基于假设每一个主机是同一时钟的基础上通过利用客户端和服务器在UDP和IP的基础上实现了RTT的简单获取。本文具体通过报文存储时间节点，将具体的获取RTT认为分派到每一个具体的报文任务上，有效减缓了发送方的存储压力，并且能通过报文的传输内容计算出两个端点之间的往返时延。

**附件**

源码以及相关文件https://github.com/beaterjy/SimpleRTT.git