

华中科技大学

课程实验报告

课程名称: 计算机通信与网络

专业班级: IOT1601
学号: U201614897
姓名: 潘越
指导教师: 鲁宏伟
报告日期: 2018年11月21日

计算机科学与技术学院

目 录

实验一 使用网络协议分析仪 Wireshark.....	1
1.1 实验环境.....	1
1.2 实验目的.....	1
1.3 实验内容及步骤.....	1
1.3.1 软件安装.....	1
1.3.2 使用 Wireshark 分析协议.....	1
1.4 实验结果.....	2
1.4.1 协议分析.....	2
1.4.2 traceroute 结果分析.....	3
1.5 实验中的问题及心得.....	4
1.5.1 实验中的问题.....	4
1.5.2 实验总结.....	5
实验二 使用网络模拟器 PacketTracer.....	6
2.1 实验环境.....	6
2.2 实验目的.....	6
2.3 实验内容及步骤.....	6
2.3.1 软件安装.....	6
2.3.2 简单网络拓扑绘制.....	7
2.3.3 观察与 IP 网络接口的各种网络硬件.....	8
2.3.4 网络拓扑（含路由器）配置.....	9
2.4 实验结果.....	11
2.4.1 ping 命令观察.....	11
2.4.2 tracert 命令观察.....	14
2.4.3 复杂拓扑.....	16
2.5 实验中的问题及心得.....	17
2.5.1 实验中的问题.....	17
2.5.2 实验总结.....	17
实验三&四 分析 Ethernet II 帧、分析集线器和交换机工作机理、分析 IP 和 ARP 协议.....	18
3.1 实验环境.....	18
3.2 实验目的.....	18
3.3 实验内容及步骤.....	18
3.3.1 分析踪迹文件中的帧结构.....	18
3.3.2 分析以太帧结构.....	19
3.3.3 分析 IP 报文结构.....	20
3.3.4 分析 ARP 协议的工作原理.....	20
3.3.5 分析集线器和交换机工作原理.....	21
3.4 实验结果.....	22

3.4.1 以太网帧结构分析.....	22
3.4.2 IP 报文结构分析.....	23
3.4.3 ARP 工作原理分析.....	24
3.4.4 集线器和交换机工作原理分析.....	25
3.5 实验中的问题及心得.....	30
3.5.1 实验中的问题.....	30
3.5.2 实验总结.....	30
 实验五 配置路由器的路由选择协议.....	31
4.1 实验环境.....	31
4.2 实验目的.....	31
4.3 实验内容及步骤.....	31
4.3.1 网络拓扑配置.....	31
5.3.2 规划 IP 地址并配置.....	31
5.3.3 配置路由器选路协议.....	32
4.4 实验结果.....	33
4.4.1 检查路由器选路协议的作用.....	33
4.4.2 三个路由器的路由表.....	35
4.5 实验中的问题及心得.....	36
4.5.1 实验中的问题.....	36
4.5.2 实验总结.....	36
 实验六 运输层实验.....	36
5.1 实验环境.....	36
5.2 实验目的.....	37
5.3 实验内容及步骤.....	37
5.3.1 网络拓扑配置.....	37
5.3.2 运输层端口观察实验--事件捕获.....	37
5.3.3 UDP 和 TCP 协议的对比分析.....	38
5.3.4 TCP 连接管理分析.....	39
5.4 实验结果.....	39
5.4.1 通过捕获的 DNS 事件查看并分析 UDP 的端口号.....	39
5.4.2 通过捕获的 HTTP 事件查看并分析 TCP 的端口号.....	41
5.4.3 分析运输层端口号.....	41
5.4.4 观察 UDP 与 TCP 协议的工作模式.....	41
5.4.5 TCP 连接建立阶段的三次握手分析.....	43
5.4.6 TCP 连接释放阶段的四次握手分析.....	44
5.5 实验中的问题及心得.....	45
5.5.1 实验中的 5.问题.....	45
5.5.2 实验总结.....	47
 实验七 DNS 解析实验.....	47
6.1 实验环境.....	47

6.2 实验目的.....	47
6.3 实验内容及步骤.....	47
6.3.1 网络拓扑配置.....	47
6.3.2 观察本地域名解析过程.....	48
6.3.3 观察外网域名解析过程.....	48
6.3.4 观察 DNS 缓存的作用.....	48
6.4 实验结果.....	49
6.4.1 观察本地域名解析过程.....	49
6.4.2 观察外网域名解析过程.....	50
6.4.3 观察 DNS 缓存的作用.....	54
6.5 实验中的问题及心得.....	54
6.5.1 实验中的问题.....	54
6.5.2 实验总结.....	55
实验八 利用 C++开发网络应用程序.....	55
7.1 实验环境.....	55
7.2 实验目的.....	55
7.3 实验内容及步骤.....	55
7.3.1 编写 Ping 网络程序.....	55
7.3.2 编写 Web 代理服务器程序.....	66
7.4 实验结果.....	80
7.4.1 ping 程序编译运行.....	80
7.4.2 Web 代理服务器程序编译运行.....	80
7.5 实验中的问题及心得.....	82
7.5.1 实验中的问题.....	82
7.5.2 实验总结.....	83
参考文献.....	84

实验一 使用网络协议分析仪 Wireshark

1.1 实验环境

1. 实验环境: 运行 Arch Linux x86_64 操作系统的 PC 机一台
2. 网络平台: 校园网 (HUST_WIRELESS)
3. IP 地址: 10.14.118.185
4. Wireshark 版本: 2.6.3

1.2 实验目的

1. 能够正确安装备置配置网络协议分析软件 Wireshark。
2. 熟悉使用 Wireshark 分析网络协议的基本方法, 加深对协议格式、协议层次和协议交互过程的理解。

1.3 实验内容及步骤

1.3.1 软件安装

在 Arch Linux 下执行 pacman -Ss wireshark, 可以看到有如下的安装包:

```
panyue@Saltedfish ~ $ pacman -Ss wireshark
community/wireshark-cli 2.6.3-1 [已安装]
    a free network protocol analyzer for Unix/Linux and Windows - CLI version
community/wireshark-common 2.6.3-1 [已安装]
    Common files used by wireshark-gtk and wireshark-qt
community/wireshark-gtk 2.6.3-1
    a free network protocol analyzer for Unix/Linux and Windows - GTK frontend
community/wireshark-qt 2.6.3-1 [已安装]
    a free network protocol analyzer for Unix/Linux and Windows - Qt frontend
panyue@Saltedfish ~ $
```

图 1.1 搜索 wireshark 的包

我们执行如下命令安装第一、二、四个即可。

```
sudo pacman -S wireshark-cli wireshark-common wireshark-qt
```

这里第三个和第四个分别是 wireshark 的 gtk 和 qt 编写的图形界面, 选择一个即可。

1.3.2 使用 Wireshark 分析协议

打开 Wireshark, 我们可以看到如下的界面。

界面中间显示的是现有的网卡, 这里选择 wlp3s0 (无线网卡), 进行操作。

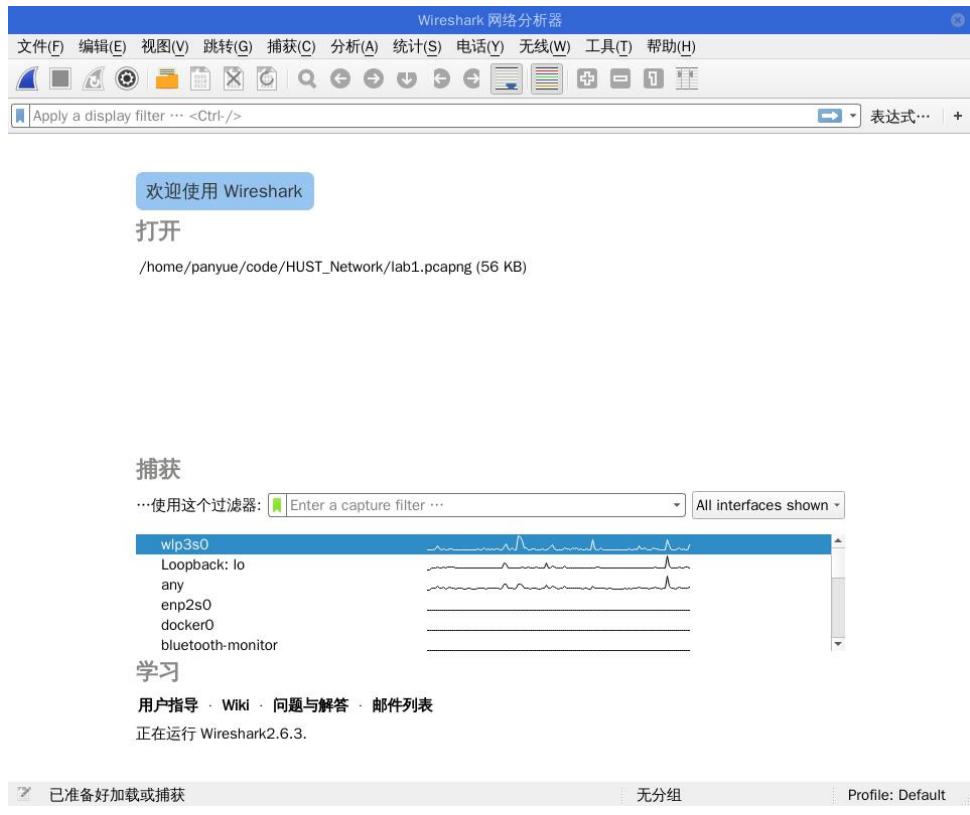


图 1.2 wireshark 主界面

实验的步骤如下：

1. 点击左上角“鲨鱼”形状按钮，开始俘获分组
2. 在命令行执行命令 traceroute www.baidu.com，等待程序执行完毕
3. traceroute 执行完毕后，结束俘获，将信息保存位 lab1.pcapng
4. 观察俘获到的任意分组，查看其各字段
5. 在筛选框中输入 icmp，筛选 icmp 包进行分析，并与 traceout 的输出进行比较。

1.4 实验结果

1.4.1 协议分析

随便点击一个分组，可以看到系统能够对俘获或者打开的踪迹文件中的分组信息进行分析，有编号、事件、源地址、目的地址、协议、长度和信息等列。最下面窗口中是对应所选分组以十六进制数和 ASCII 形式的内容。

这里选择 TCP 协议的源端口号，可以看到下面窗口中对应的十六进制位和 ASCII 形式的内容。并且从右侧内容中可以看出这是一个 HTTP POST 请求。

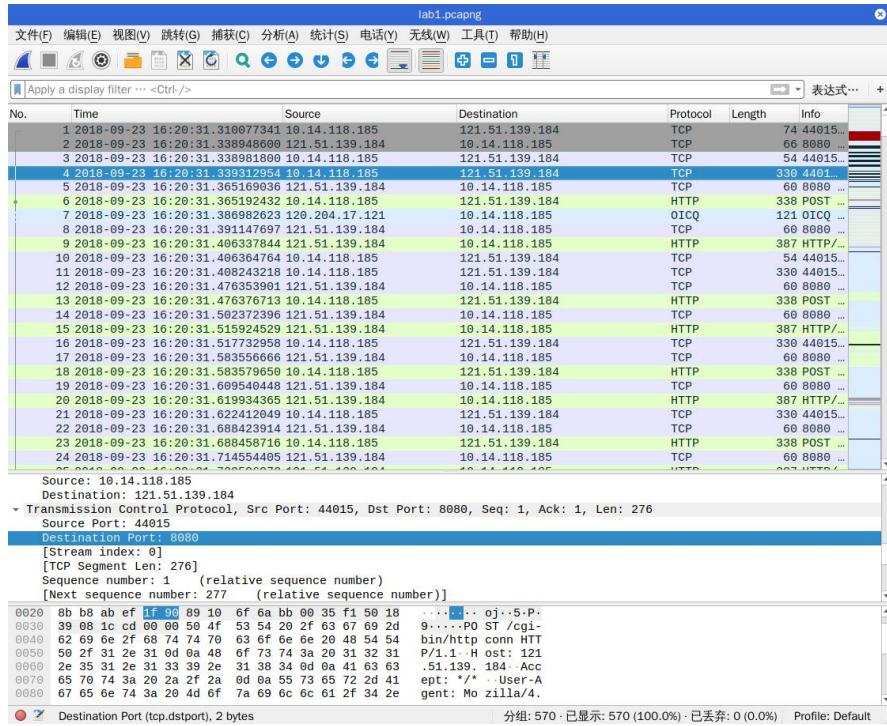


图 1.3 wireshark 协议分析

1.4.2 traceroute 结果分析

traceroute 的输出结果如下，显示为*的表示没有数据返回：

```
panyue@Saltedfish ~ traceroute www.baidu.com
traceroute to www.baidu.com (111.13.100.92), 30 hops max, 60 byte packets
1 * *
2 192.168.243.33 (192.168.243.33) 4.958 ms 4.966 ms 4.961 ms
3 192.168.243.129 (192.168.243.129) 4.955 ms 5.557 ms 5.558 ms
4 111.47.18.1 (111.47.18.1) 25.458 ms 25.465 ms 25.459 ms
5 * *
6 221.183.58.101 (221.183.58.101) 11.621 ms 6.706 ms 6.653 ms
7 221.183.37.225 (221.183.37.225) 23.821 ms 23.816 ms 23.289 ms
8 * *
9 111.13.98.93 (111.13.98.93) 23.886 ms 111.13.0.174 (111.13.0.174) 27.807 ms 111.13.98.93 (111.13.98.93) 24.618 ms
10 111.13.98.101 (111.13.98.101) 24.625 ms 111.13.98.93 (111.13.98.93) 24.459 ms 111.13.108.22 (111.13.108.22) 26.740 ms
11 111.13.112.61 (111.13.112.61) 28.846 ms 111.13.112.57 (111.13.112.57) 36.061 ms *
12 * *
13 * *
14 * *
15 * *
16 * *
17 * *
18 * *
19 * *
20 * *
21 * *
22 * *
23 * *
24 * *
25 * *
26 * *
27 * *
28 * *
29 * *
30 * *
```

图 1.4 traceroute 输出结果

通过网络上的 IP 位置查询工具，给出每一步的 IP 地理位置

1. * * * 三次均拒绝，推测为第一级路由器
2. 192.168.243.33 本地局域网
3. 192.168.243.129 本地局域网

4. 111.47.18.1 武汉

5. * * *

6. 221.183.58.101 中国移动

7. 221.183.37.225 中国移动

8. * * *

9. 111.13.98.93 111.13.0.174 111.13.98.93 北京市 中国移动

10. 111.13.98.101 111.13.98.93 111.13.108.22 北京市 中国移动

11. 111.13.112.61 111.13.112.57 * 北京市 中国移动

12. * * *

Wireshark 筛选出的 icmp 包如下图：

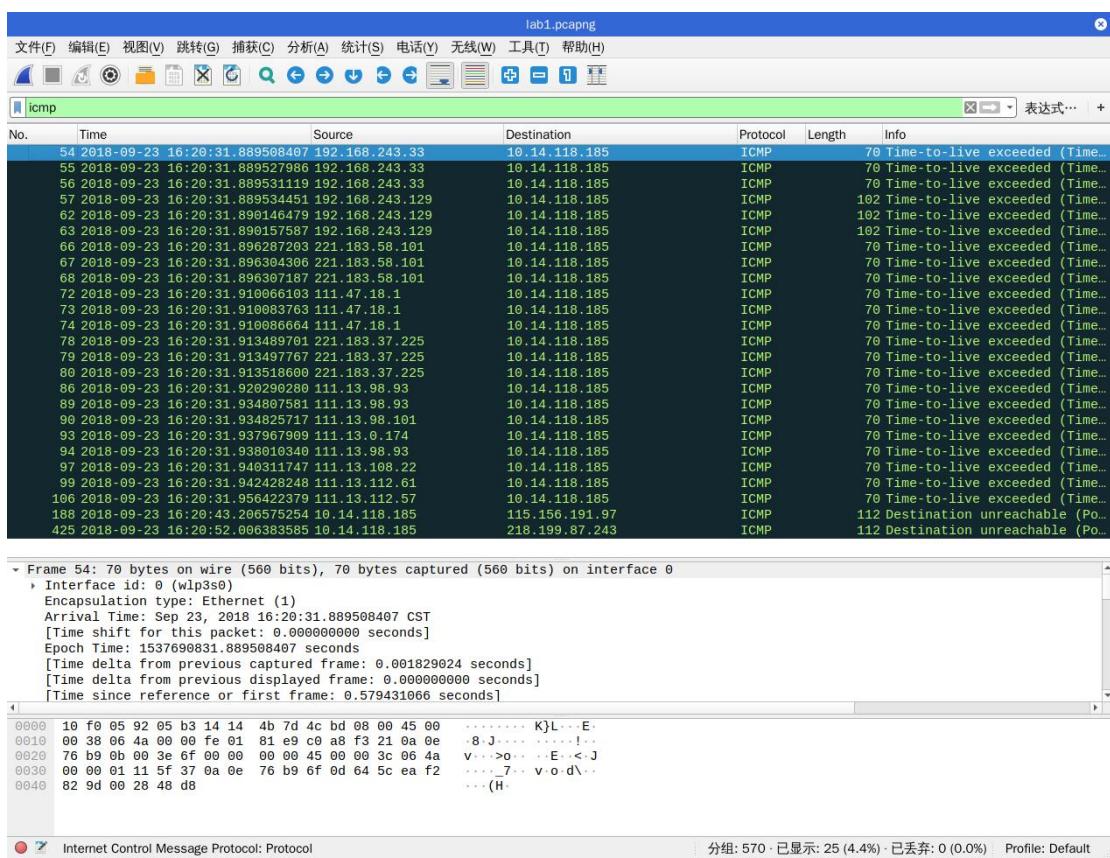


图 1.5 icmp 包俘获结果

仔细观察结果，可以看到俘获的包结果和 traceroute 的输出是一一对应的，traceroute 默认是发送三个分组，这里每一跳一般都有三个 icmp 包返回，出现*的第 11 级就只有两个返回，并且 IP 地址都是对应的，验证了实验的正确性。

1.5 实验中的问题及心得

1.5.1 实验中的问题

1. 为什么有些级中出现了相同的 IP 地址？

`traceroute` 会发送三个分组，这些分组可能走了不同的路径，111.13.98.93 可能在第一个分组走的路径上是第九跳，在另一个分组走的路径上是第十跳，因此不同的级中出现了相同的 IP 地址。

2. Wireshark 中出现了很多不同的颜色，代表什么？

点击视图->着色规则，可以看到如下图所示的窗口：



图 1.6 Wireshark 着色规则

从图中我们就可以看出，红色的往往是一些丢弃的包，黑色是错误的包，浅色的一般是正常的包等等，熟练记住了颜色分类有助于更高效地使用 Wireshark。

1.5.2 实验总结

本次实验练习 Wireshark 的使用，自己以前也玩过这方面的一些操作，并不难。总之加深了对网络协议的理解，熟悉了一次网络请求的整个过程，为之后的实验做好了准备。

实验二 使用网络模拟器 PacketTracer

2.1 实验环境

1. 实验环境：运行 Windows 10 64 位操作系统的 PC 机一台
2. PacketTracer 版本：7.0.0.0202

2.2 实验目的

1. 正确安装和配置网络模拟器软件 PacketTracer。
2. 掌握使用 PacketTracer 模拟网络场景的基本方法，加深对网络环境、网络设备和网络协议交互过程等方面的理解。
3. 观察与 IP 网络接口的各种网络硬件及其适用场合。

2.3 实验内容及步骤

2.3.1 软件安装

在思科官方网站中注册帐号并下载安装 PacketTracer，安装完成后，界面如下图：

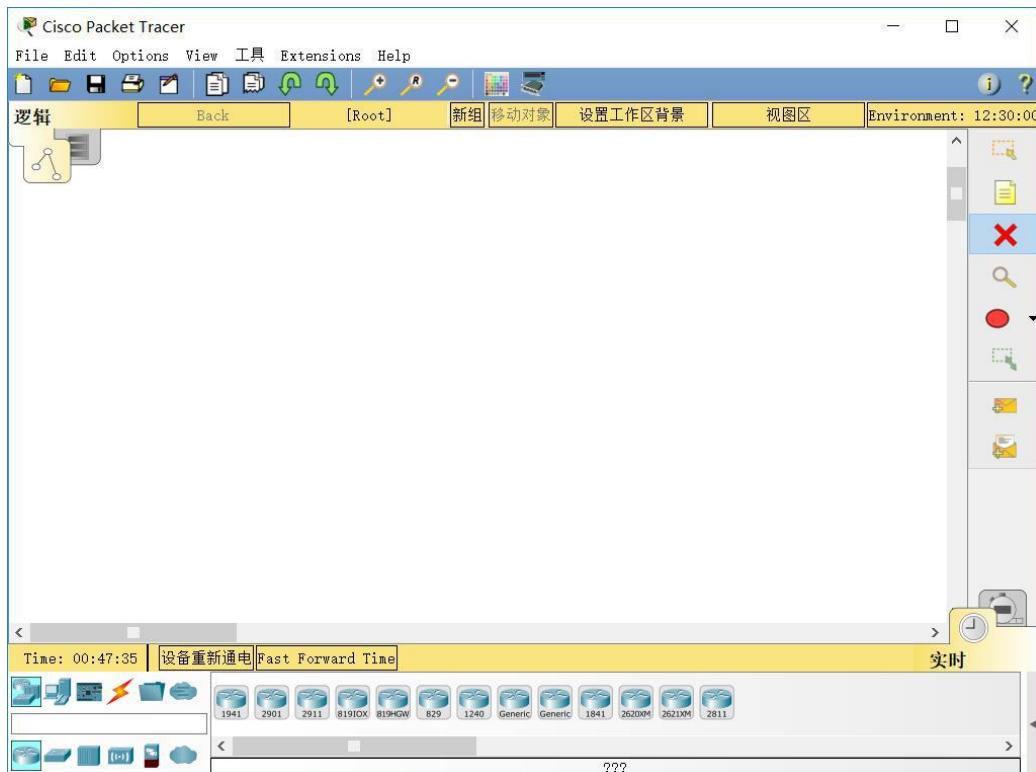


图 2.1 PacketTracer 界面

下方提供了各种常见的网络设备，如 PC、交换机、路由器等等，我们可以直接拖动来搭建网络拓扑。

2.3.2 简单网络拓扑绘制

在工作区添加两台 PC 和一台 2960 交换机，用直通线连接如下：



图 2.2 搭建简单的网络拓扑

在上图中，绿色灯代表网络线路已联通正常，橙色代表网络线路无法联通，需等待几秒钟。

待交换机两侧的灯都变成绿色之后，进行一下简单的 ping 测试。

双击主机 0，选择 Config/FastEthernet0，配置 IP 和子网掩码如下：IP 选择静态配置，地址为 192.168.1.1，子网掩码为 255.255.255.0

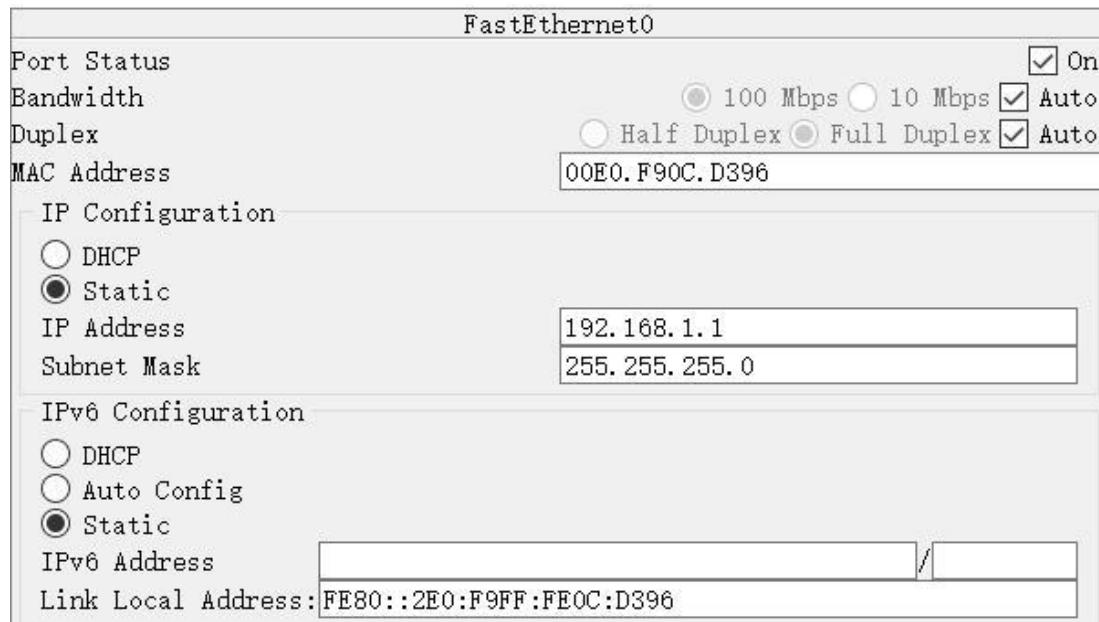


图 2.3 配置主机 0 的 IP

类似的，将主机 1 的 IP 地址配置为 192.168.1.2。然后双击主机 0，选择 Desktop/Command Prompt，执行 ping 192.168.1.2，测试是否可以 ping 通主机 1。

```
C:\>ping 192.168.1.2

Pinging 192.168.1.2 with 32 bytes of data:

Reply from 192.168.1.2: bytes=32 time<lms TTL=128

Ping statistics for 192.168.1.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>
```

图 2.4 ping 测试

观察到主机 1 给出了相应，验证网络拓扑配置正确。

2.3.3 观察与 IP 网络接口的各种网络硬件

在工作区添加一个路由器 2620XM，双击并选择 Physical/Modules，添加模块 NM-1FE-2X 如下图：

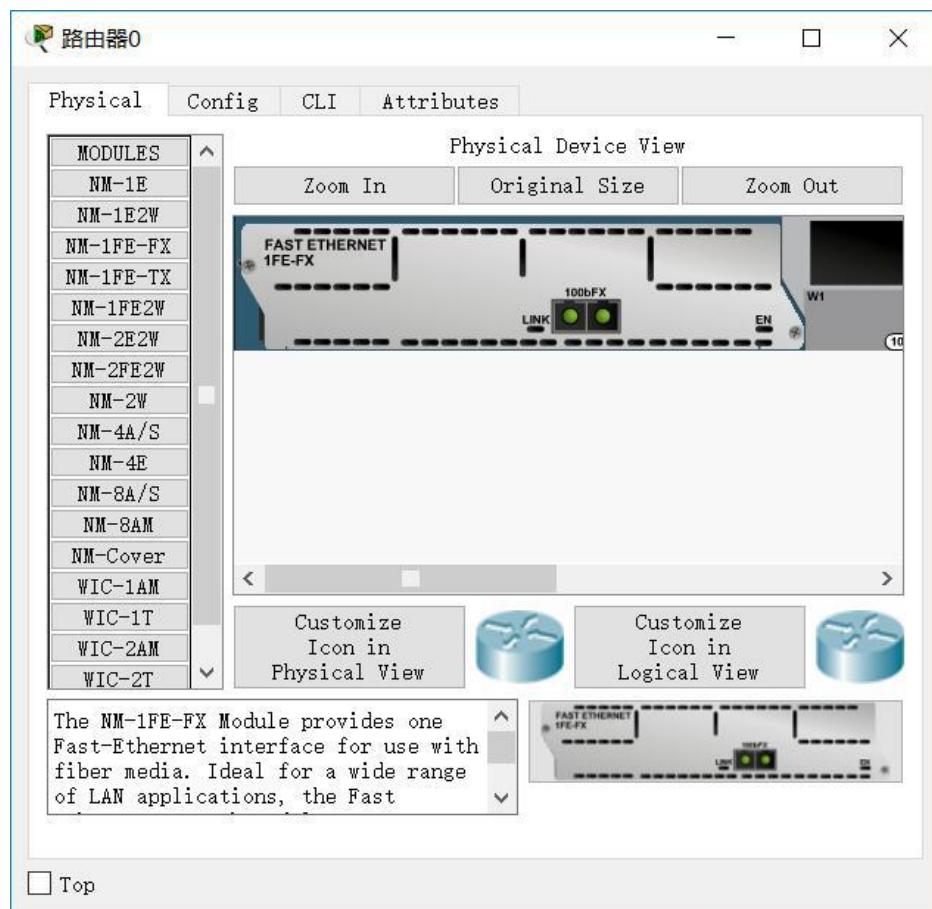


图 2.5 路由器模块观察

通过拖动左边的模块，可以查看各个模块的接口情况。

通过网上查阅资料，了解到各个模块的作用如下：

表 2.1 路由器 2620XM 各模块作用

模块	作用
NM-1FE-FX	1 口快速以太网模块 FX 光纤接口
NM-1FE-TX	1 口快速以太网模块 TX 双绞线接口
NM-2FE2W	2 口 10/100 以太网 2 个广域网卡插槽
NM-8AM	8 口模拟 Modem 网络模块
NM cover plate	NM 盖板可以保护内部的电子元件.有助于保持足够的冷却气流

2.3.4 网络拓扑（含路由器）配置

增添路由器等设备，重新构建网络拓扑，在工作区添加两台 PC 机和一台 1841 路由器，选择自动添加链路，结果如下图，链路两端亮红灯表示链路不通：

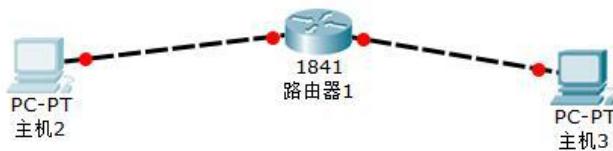


图 2.6 新构建的网络拓扑

接下来配置网络设备，双击主机 2，选择 Desktop/IP Configuration，IP 地址设置为 192.168.1.2，子网掩码设置为 255.255.255.0，默认网关设置为 192.168.1.1，如下图：

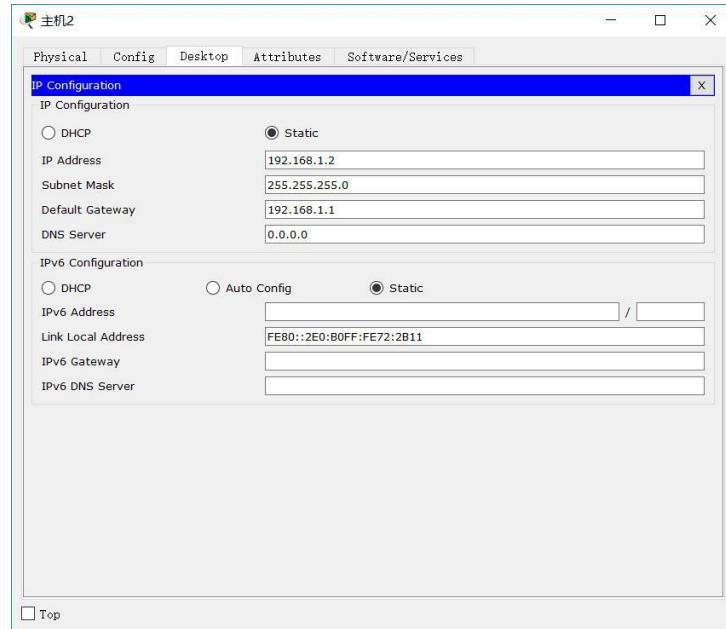


图 2.7 主机 IP 配置

同样我们双击主机 3，设置 IP 地址为 192.168.2.2，子网掩码为 255.255.255.0， 默认网关为 192.168.2.2。

同时，路由器和交换机不同，交换机不需要配置但路由器需要手动配置，双击路由器，选择 Config/FastEthernet0/0，选择 IP Configuration，设置 IP 地址为 192.168.1.1，子网掩码为 255.255.255.0，选择 Config/FastEthernet0/1，设置 IP 地址为 192.168.2.1，子网掩码为 255.255.255.0，如下图：

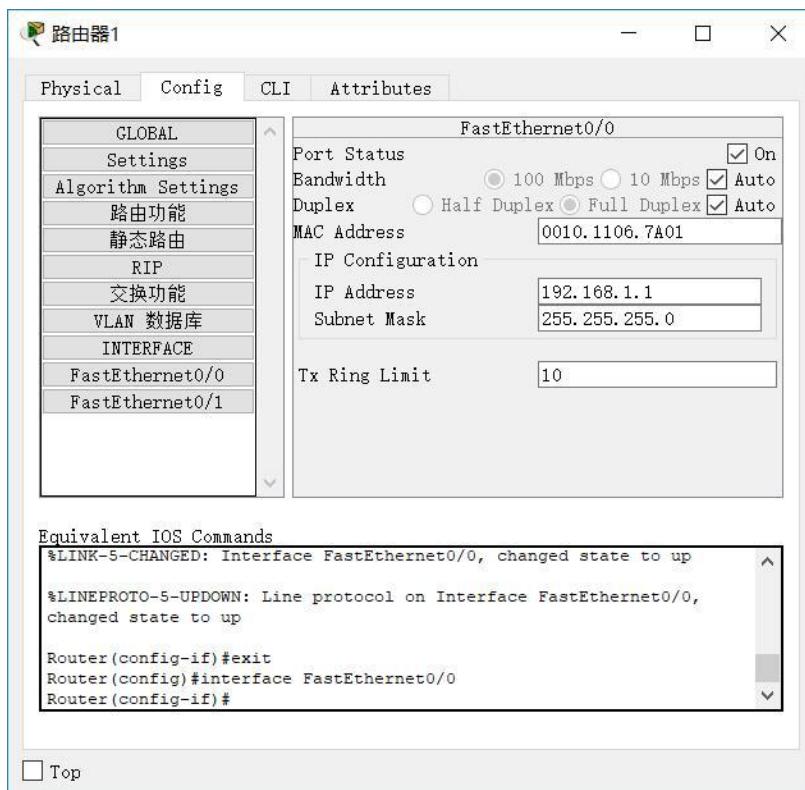


图 2.8 路由器 IP 配置

当我们配置好 IP 之后，将端口的状态 Port Status 设置为 ON，就可以发现链路两端已经变成了绿色，如下图：



图 2.9 IP 配置完成后的网络拓扑

2.4 实验结果

2.4.1 ping 命令观察

接着就可以进行观察实验了，点击 PacketTracer 右下角进入模拟模式，双击主机 2，执行 ping 192.168.2.2，同时点击模拟面板的自动捕获/播放，系统会开始演示 ping 命令的数据包传递过程，具体的过程如下图：

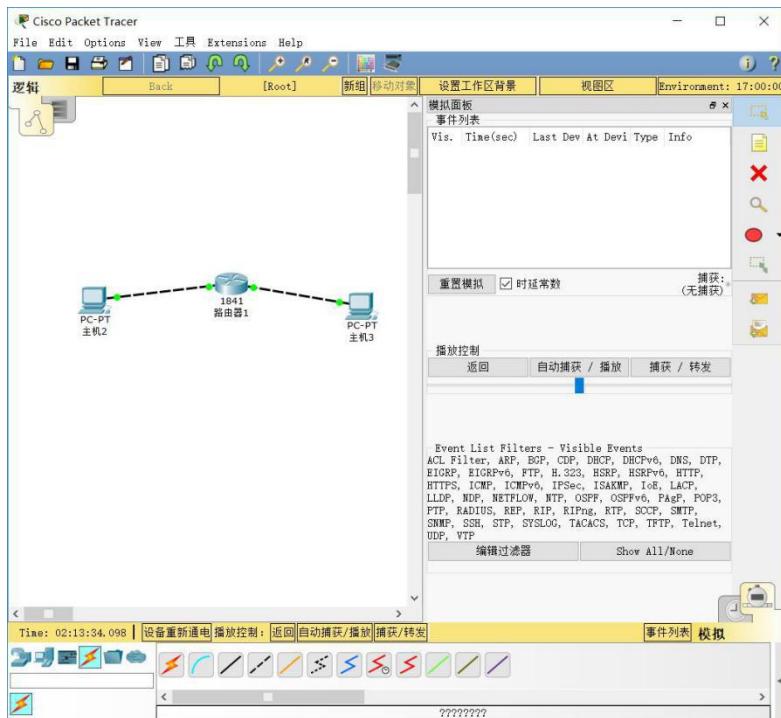


图 2.10 PacketTracer 模拟模式

我们可以看到数据包的传递过程：



图 2.11 PacketTracer 模拟数据包传递

在主机 2 上，执行 ping 192.168.2.2，尝试 ping 主机 3：

```
C:\>ping 192.168.2.2

Pinging 192.168.2.2 with 32 bytes of data:

Request timed out.
Reply from 192.168.2.2: bytes=32 time=4ms TTL=127
Reply from 192.168.2.2: bytes=32 time=4ms TTL=127
Reply from 192.168.2.2: bytes=32 time=4ms TTL=127

Ping statistics for 192.168.2.2:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 4ms, Maximum = 4ms, Average = 4ms

C:\>
```

图 2.12 主机 2 尝试 ping 主机 3

当一打开端口，模拟就可以开始捕获结果。

一段时间以后，观察模拟面板中的事件列表，可以看到从打开端口开始到整个 ping 命令执行结束的过程中，捕获到的数据包如下：

事件列表					
Vis.	Time(sec)	Last Devic	At Devic	Type	Info
99.396	--		路由器1	ARP	
99.396	--		主机3	ARP	
99.396	--		路由器1	ARP	
99.396	--		主机2	ARP	
99.397	路由器1		主机3	ARP	
99.397	主机3		路由器1	ARP	
99.397	路由器1		主机2	ARP	
00.207	主机2		路由器1	ARP	

图 2.13 ping 命令模拟结果（一）

事件列表						
Vis.	Time(sec)	Last Devic	At Devic	Type	Info	
	99.397	主机2	路由器1	ARP		
	159.403	--	主机2	ICMP		
	159.403	--	主机2	ARP		
	159.404	主机2	路由器1	ARP		
	159.405	路由器1	主机2	ARP		
	159.405	--	主机2	ICMP		
	159.406	主机2	路由器1	ICMP		
	159.406	路由器1	主机2	ARP		

图 2.14 ping 命令模拟结果（二）

分析前两张图，从 99.396 时刻到 99.397 时刻是路由器刚刚打开端口的时刻，各个设备（路由器、两台主机）分别使用 ARP 包获取自身的 MAC 地址并存储在 ARP 表中，同时路由器的两个端口，主机 2 和主机 3 都广播发送 ARP 包（99.397 时刻的 ARP 包均为请求包）。

事件列表						
Vis.	Time(sec)	Last Devic	At Devic	Type	Info	
	159.406	--	路由器1	ARP		
	159.407	路由器1	主机3	ARP		
	159.408	主机3	路由器1	ARP		
	165.408	--	主机2	ICMP		
	165.409	主机2	路由器1	ICMP		
	165.410	路由器1	主机3	ICMP		
	165.411	主机3	路由器1	ICMP		

图 2.15 ping 命令模拟结果（三）

从图二和图三可以看出，从 159.403 时刻开始，ping 命令执行，简单分析一下过程：

主机 2 首先需要向路由器发送 ICMP 包，但主机 2 并没有路由器的 MAC 地址，于是首先 159.404 时刻，主机 2 向路由器（192.168.1.1）发送 ARP 包，路由器在 159.405 时刻，向路由器发送 ARP 应答包，在 159.406 时刻，路由器成功将 ICMP 包发向路由器。

159.407 和 159.408 两个时刻的数据包均为 ARP 应答包，发送成功之后，主机 3 的 ARP 表中就有了路由器 1 的 MAC 地址，此时，主机 2、路由器、主机 3 已经可以互相通信。

但此时 ping 命令已经超时，在命令行（图 12）中也可以看到第一个包是丢失了的，没有再继续发送。

事件列表						
Vis.	Time(sec)	Last Devic	At Devic	Type	Info	
	165.412	路由器1	主机2	ICMP		
	166.414	--	主机2	ICMP		
	166.415	主机2	路由器1	ICMP		
	166.416	路由器1	主机3	ICMP		
	166.417	主机3	路由器1	ICMP		
	166.418	路由器1	主机2	ICMP		
	167.419	--	主机2	ICMP		
	167.420	主机2	路由器1	ICMP		
	167.421	路由器1	主机3	ICMP		
	167.422	主机3	路由器1	ICMP		
	167.423	路由器1	主机2	ICMP		
	167.424	主机2	路由器1	ICMP		

重置模拟 时延常数 捕获到: * 2268.898 秒 *

图 2.16 ping 命令模拟结果 (四)

模拟面板						
事件列表						
Vis.	Time(sec)	Last Devic	At Devic	Type	Info	
	166.417	主机3	路由器1	ICMP		
	166.418	路由器1	主机2	ICMP		
	167.419	--	主机2	ICMP		
	167.420	主机2	路由器1	ICMP		
	167.421	路由器1	主机3	ICMP		
	167.422	主机3	路由器1	ICMP		
	167.423	路由器1	主机2	ICMP		
	167.424	主机2	路由器1	ICMP		

重置模拟 时延常数 捕获到: * 2268.898 秒 *

图 2.17 ping 命令模拟结果 (五)

此后，从 165.408 时刻开始，ICMP 包正常传输，按照主机 2->路由器，路由器->主机 3，主机 3->路由器，路由器->主机 2 的顺序发送数据包。

2.4.2 tracert 命令观察

在主机 2 上，执行 `tracert 192.168.2.2`，点击自动捕获/播放，终端输出如下图：

```
C:\>tracert 192.168.2.2

Tracing route to 192.168.2.2 over a maximum of 30 hops:
 1  2 ms        2 ms        2 ms      192.168.1.1
 2  4 ms        4 ms        4 ms      192.168.2.2

Trace complete.
```

图 2.18 tracert 命令输出

一段时间以后，观察模拟面板中的事件列表，可以看到从打开端口开始到整个 ping 命令执行结束的过程中，捕获到的数据包如下：

模拟面板					
事件列表					
Vis.	Time(sec)	Last Devic	At Devic	Type	Info
	150.084	--	主机2	ICMP	█
	150.085	主机2	路由器1	ICMP	█
	150.085	--	路由器1	ICMP	█
	150.086	路由器1	主机2	ICMP	█
	150.188	--	主机2	ICMP	█
	150.189	主机2	路由器1	ICMP	█
	150.189	--	路由器1	ICMP	█
	150.190	路由器1	主机2	TCP	█

重置模拟 时延常数 捕获到: * 300.617 秒

图 2.19 tracert 命令演示结果 (一)

模拟面板					
事件列表					
Vis.	Time(sec)	Last Devic	At Devic	Type	Info
	150.190	路由器1	主机2	ICMP	█
	150.294	--	主机2	ICMP	█
	150.295	主机2	路由器1	ICMP	█
	150.295	--	路由器1	ICMP	█
	150.296	路由器1	主机2	ICMP	█
	150.396	--	主机2	ICMP	█
	150.397	主机2	路由器1	ICMP	█
	150.398	路由器1	主机2	TCP	█

重置模拟 时延常数 捕获到: * 300.617 秒

图 2.20 tracert 命令演示结果 (二)

观察结果可以发现，从 150.084 时刻开始到 150.296 时刻，是 tracert 的第一级，主机 2 向路由器发送了 3 个包，路由器做出应答，同时也可以看见发送的 ICMP 包到路由器之后就被自己丢掉了，符合 tracert 设置的 TTL。

模拟面板					
事件列表					
Vis.	Time(sec)	Last Devic	At Devic	Type	Info
	150.398	路由器1	主机3	ICMP	█
	150.399	主机3	路由器1	ICMP	█
	150.400	路由器1	主机2	ICMP	█
	150.503	--	主机2	ICMP	█
	150.504	主机2	路由器1	ICMP	█
	150.505	路由器1	主机3	ICMP	█
	150.506	主机3	路由器1	ICMP	█
	150.507	路由器1	主机2	TCP	█

重置模拟 时延常数 捕获到: * 300.617 秒

图 2.21 tracert 命令演示结果 (三)

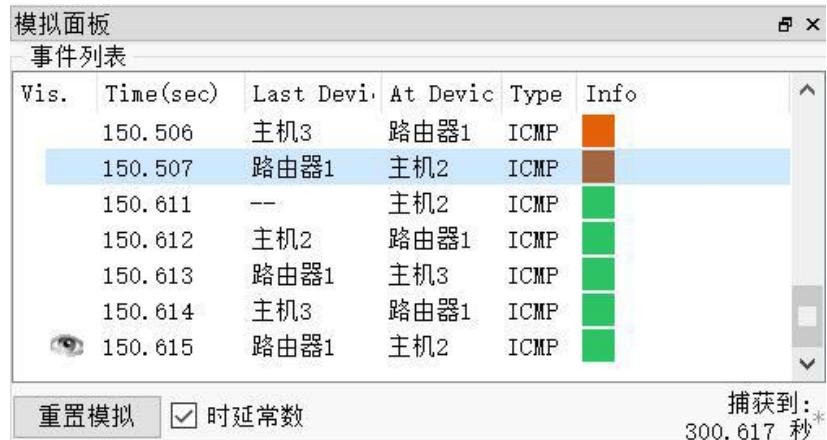


图 2.22 tracert 命令演示结果 (四)

从 150.397 时刻开始，即是 tracert 第二级的 3 个 ICMP 包发送的流程，主机 2->路由器，路由器->主机 3，主机 3->路由器，路由器->主机 2。

2.4.3 复杂拓扑

简单的利用主机，交换机，路由器构建如下的网络拓扑：

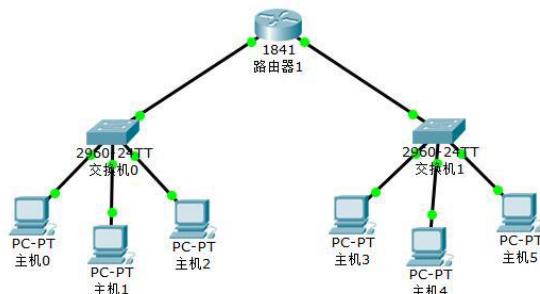


图 2.23 网络拓扑构建

配置左边三个主机的 IP 为 192.168.1.2~192.168.1.4，右边三个主机为 192.168.2.2~192.168.2.4，配置路由器同上。

简单在主机 0 上执行以下 ping 192.168.2.4 命令，可以清晰地看到数据包整个的发送过程，包括 ARP 和交换机 STP 的广播，ICMP 包的传递等等。

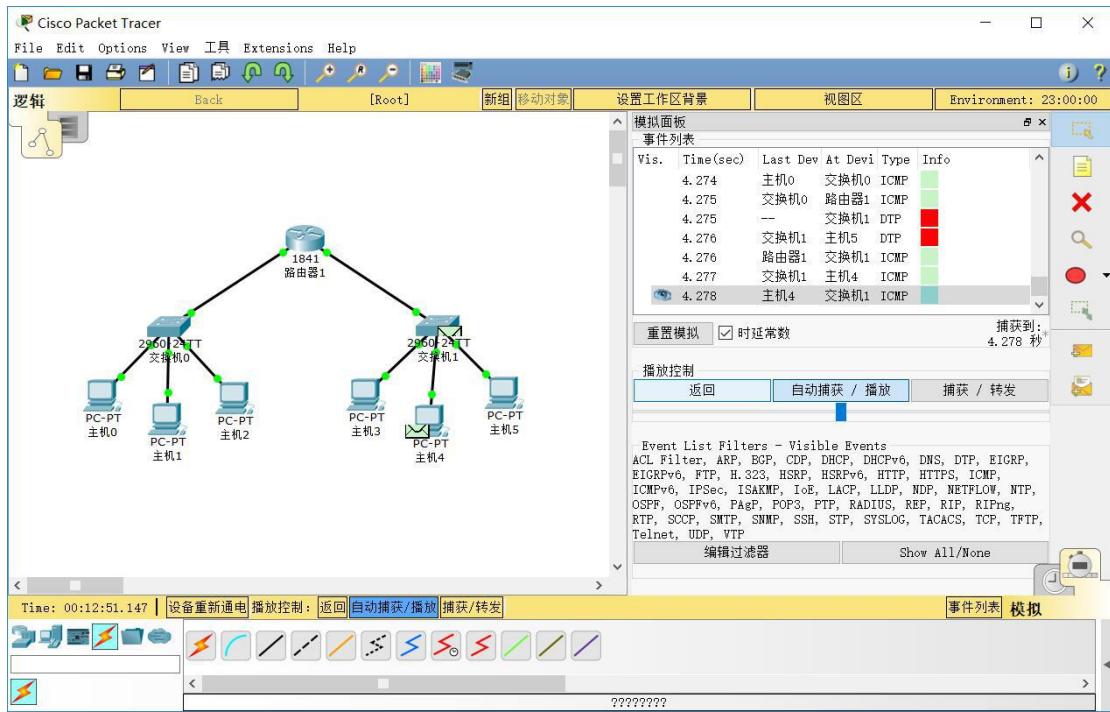


图 2.24 模拟运行截图

2.5 实验中的问题及心得

2.5.1 实验中的问题

1. ARP 包的原理与功能？

ARP（地址解析协议），其基本功能为透过目标设备的 IP 地址，查询目标设备的 MAC 地址。在以太网协议中规定，同一局域网中的一台主机要和另一台主机进行直接通信，必须要知道目标主机的 MAC 地址。但是在 TCP/IP 协议中，网络层和传输层只关心目标主机的 IP 地址。于是就需要一种协议，将网络层地址转换为链路层地址，这就是 ARP。

当一台主机接收到 ARP 请求包之后，会发送对应的 ARP 应答包，其中包含自己的 MAC 地址；当一台主机接收到 ARP 应答包之后，会将对应的 MAC 地址和 IP 地址记录到本机的 ARP 表中，这样就可以进行 MAC 地址和 IP 地址之间的转换了。

2.5.2 实验总结

本次实验练习了使用 PacketTracer 软件，了解了如何使用它来模拟真实的网络，同时也亲手模拟了一下两个常见命令的数据包发送方式，同时也查阅了解了一些上课中还没有讲到的问题，收获很多。

实验三&四 分析 Ethernet II 帧、分析集线器和交换机工作机 理、分析 IP 和 ARP 协议

3.1 实验环境

1. 实验环境：运行 Arch Linux x86_64 位操作系统的 PC 机一台
2. 网络环境：校园网（教育网）
3. IP 地址：10.11.56.37
4. ARP 实验网络环境：电信网，两台主机处于同一子网内，
A 本机 IP：192.168.1.108，B 主机 IP：192.168.1.103
5. Wireshark 版本：2.6.4
6. PacketTracer 版本：7.0.0.0202

3.2 实验目的

1. 深入理解 Ethernet II 帧结构。
2. 深入理解 IP 报文结构和工作原理。
3. 深入理解地址解析协议 ARP 的工作原理。
4. 基本掌握使用 Wireshark 分析俘获的踪迹文件的基本技能。
5. 理解 IP 和以太网协议的关系，掌握 IP 地址和 MAC 地址的映射机制，搞清楚 IP 报文是如何利用底层的以太网帧进行传输的。
6. 观察交换机处理广播和单播报文的过程
7. 比较交换机和集线器的工作过程。
8. 掌握使用 PacketTracer 模拟网络场景的基本方法，加深对网络环境，网络设备和网络协议交互过程等方面的理解。

3.3 实验内容及步骤

3.3.1 分析踪迹文件中的帧结构

打开 Wireshark 并俘获一组网络分组。如下图所示，选择第 12 号帧（TCP）进行分析。

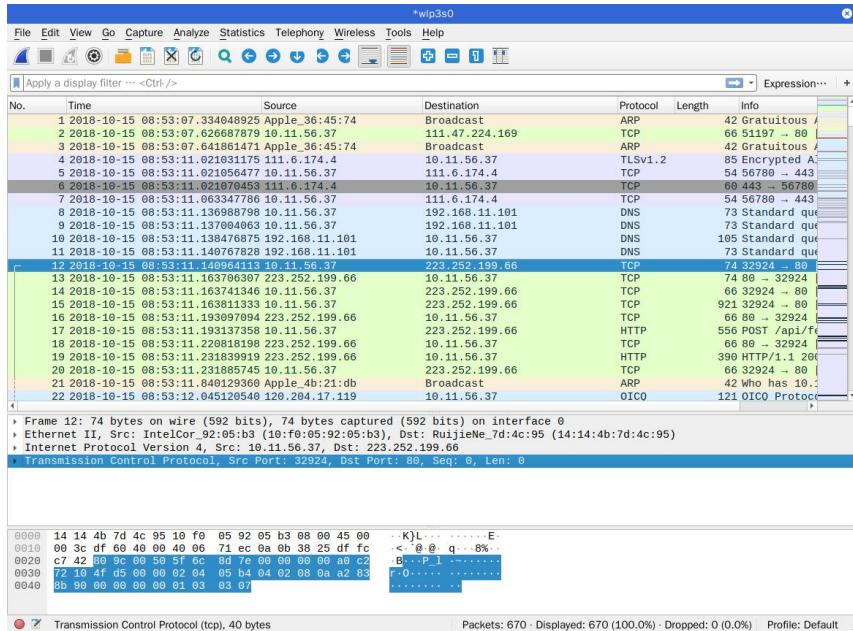


图 3.1 Wireshark 俘获帧

该帧显示为绿色，表明是一个正常的 TCP 数据包，同时我们可以在下面看到该帧有 Ethernet:IP:TCP 的封装结构。

进一步分析，点击首部细节信息栏中的“Ethernet II”行，可以看到如下的信息：

```

    ▶ Frame 12: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
    ▶ Ethernet II, Src: IntelCor_92:05:b3 (10:f0:05:92:05:b3), Dst: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
      ▶ Destination: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
        Address: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
          .... ..0. .... .... .... = LG bit: Globally unique address (factory default)
          .... ..0. .... .... .... = IG bit: Individual address (unicast)
      ▶ Source: IntelCor_92:05:b3 (10:f0:05:92:05:b3)
        Address: IntelCor_92:05:b3 (10:f0:05:92:05:b3)
          .... ..0. .... .... .... = LG bit: Globally unique address (factory default)
          .... ..0. .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
    ▶ Internet Protocol Version 4, Src: 10.11.56.37, Dst: 223.252.199.66
    ▶ Transmission Control Protocol, Src Port: 32924, Dst Port: 80, Seq: 0, Len: 0
  
```

图 3.2 Ethernet II 详细信息

从上图中我们可以看到如下的信息：

源 MAC 地址：10:f0:05:92:05:b3

目的 MAC 地址：14:14:4b:7d:4c:95

以太网类型字段 (Type) : 0x0800，表示上层封装的是 IP 数据报

3.3.2 分析以太帧结构

在终端中执行 ping www.baidu.com，并使用 Wireshark 俘获分组，筛选出 ICMP 报文如下：

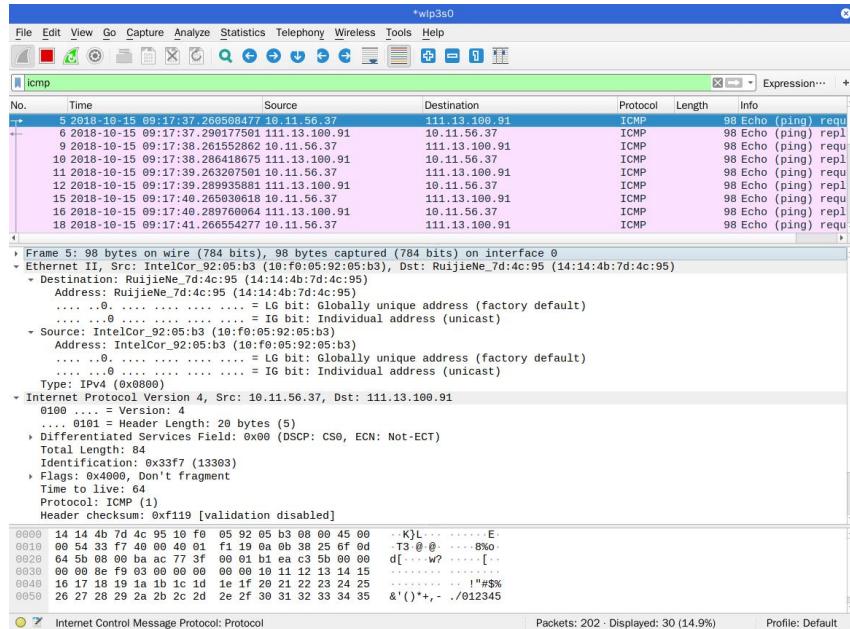


图 3.3 ping 命令分组俘获

3.3.3 分析 IP 报文结构

接着使用 5 号帧，并展开其详细信息进行分析，具体见实验结果。

3.3.4 分析 ARP 协议的工作原理

在终端中执行 `ifconfig -a`，可以看到设备所有网卡的信息，输出如下：

```
panyue@Saltedfish: ~ ifconfig -a
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
        inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
                ether 02:42:58:a5:56:d0 txqueuelen 0 (Ethernet)
                RX packets 0 bytes 0 (0.0 B)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 0 bytes 0 (0.0 B)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp2s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
        ether b0:25:aa:23:1d:69 txqueuelen 1000 (Ethernet)
                RX packets 0 bytes 0 (0.0 B)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 0 bytes 0 (0.0 B)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
                inet6 ::1 prefixlen 128 scopeid 0x10<host>
                    loop txqueuelen 1000 (Local Loopback)
                    RX packets 93604 bytes 12291336 (11.7 MiB)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 93604 bytes 12291336 (11.7 MiB)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 10.11.56.37 netmask 255.255.240.0 broadcast 10.11.63.255
                inet6 fe80::acb3:1d8c:a792:40c4 prefixlen 64 scopeid 0x20<link>
                inet6 2001:250:1:4000:40df:cbla:bf15:cc7:62e5 prefixlen 64 scopeid 0x0<global>
                ether 10:f0:05:92:05:b3 txqueuelen 1000 (Ethernet)
                RX packets 418558 bytes 547348279 (521.9 MiB)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 167825 bytes 17030633 (16.2 MiB)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

图 3.4 ifconfig -a 命令输出

由输出可知，设备所使用的无线网卡 wlp3s0 的 MAC 地址为 10:f0:05:92:05:b3，IP 地址为 10.11.56.37，子网掩码为 255.255.240.0。

执行 `arp -a` 命令，可以看到设备的 ARP 表如下：

```
panyue@Saltedfish ~ $ arp -a
? (10.11.56.55) at 14:14:4b:7d:4c:95 [ether] on wlp3s0
_gateway (10.11.63.254) at 14:14:4b:7d:4c:95 [ether] on wlp3s0
? (10.11.56.58) at 14:14:4b:7d:4c:95 [ether] on wlp3s0
? (10.11.56.45) at e4:47:90:15:a2:73 [ether] on wlp3s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on enp2s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on docker0
```

图 3.5 arp -a 命令输出

在主机 A 与 B 上分别执行 ARP 表项清除命令，结果如下图：

```
panyue@Saltedfish ~ $ arp -a
_gateway (192.168.1.1) at f4:83:cd:56:c3:d2 [ether] on wlp3s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on wlp3s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on docker0
apollo.archlinux.org (138.201.81.199) at <incomplete> on enp2s0
panyue@Saltedfish ~ $
```

图 3.6 清空主机 A 上的 ARP 表

接口:	Internet 地址	物理地址	类型
192.168.217.1 --- 0x6	224.0.0.22	01-00-5e-00-00-16	静态
192.168.19.1 --- 0x12	224.0.0.22	01-00-5e-00-00-16	静态
192.168.1.103 --- 0x14	192.168.1.1	f4-83-cd-56-c3-d2	动态
	224.0.0.22	01-00-5e-00-00-16	静态

图 3.7 清空主机 B 上的 ARP 表

接着在主机 A 上运行 Wireshark 程序，执行包俘获操作，稍后停止发 Ping 包。分别检查两台主机的 ARP 表。

3.3.5 分析集线器和交换机工作原理

打开 PacketTracker，构建拓扑如下图：

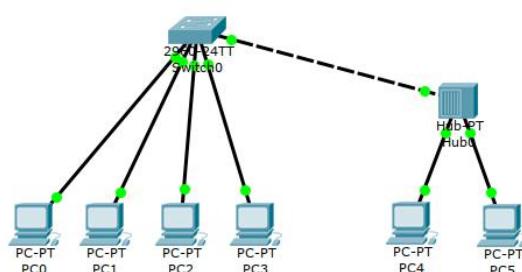


图 3.8 拓扑构建

接着，为每台 PC 配置自己的 IP 地址，从左到右分别配置为 192.168.1.2~192.168.1.7，子网掩码均设置为 255.255.255.0，如下图：

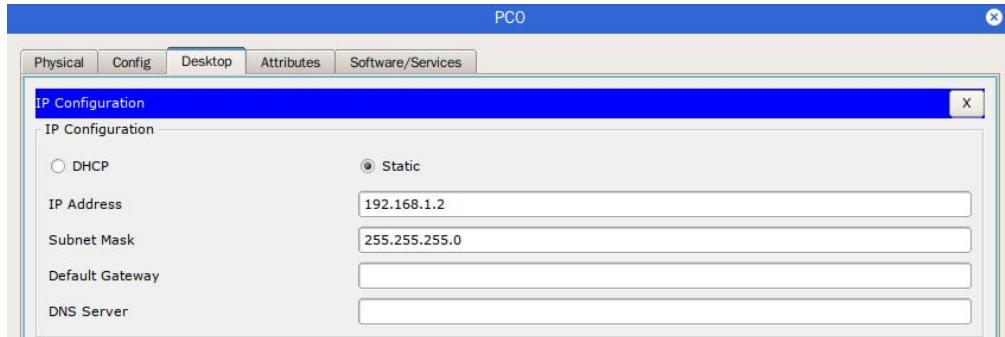


图 3.9 主机 IP 配置

在实时与模拟模式之间切换 4 次，完成生成树协议，所有链路指示灯变为绿色，最后停留在模拟模式中，就可以开始观察实验了。

3.4 实验结果

3.4.1 以太网帧结构分析

任选一个上面俘获到的 ICMP 帧进行分析，这里选择第一个，如下图：

```

Frame 5: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: IntelCor_92:05:b3 (10:f0:05:92:05:b3), Dst: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
  Destination: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
    Address: RuijieNe_7d:4c:95 (14:14:4b:7d:4c:95)
    .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    .... ..0. .... .... .... .... = IG bit: Individual address (unicast)
  Source: IntelCor_92:05:b3 (10:f0:05:92:05:b3)
    Address: Intelcor_92:05:b3 (10:f0:05:92:05:b3)
    .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    .... ..0. .... .... .... .... = IG bit: Individual address (unicast)
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 10.11.56.37, Dst: 111.13.100.91
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 84
  Identification: 0x33f7 (13303)
  Flags: 0x4000, Don't fragment
  Time to live: 64
  Protocol: ICMP (1)
  Header checksum: 0xf119 [validation disabled]

```

图 3.10 ICMP 数据帧分析

由上图可以分析，并回答实验提出的如下问题：

(1) 本机的 48 比特 MAC 地址是什么？

本机的 MAC 地址即上述报文中的源 MAC 地址 10:f0:05:92:05:b3

(2) 以太帧中目的 MAC 地址是什么？它是你选定的远地 Web 服务器的 MAC 地址吗？

目的 MAC 地址为：14:14:4b:7d:4c:95，显然这并不是 Web 服务器的 MAC 地址而是路由器的 MAC 地址。实际上不同的 ICMP 数据帧的目的 MAC 地址并不同，他们是该数据帧应该走向的下一跳设备的 MAC 地址。

(3) 给出 2 字节以太类型字段的十六进制的值。它表示该以太帧包含了什么样的协议？上网查找如果其中封装的 IPv6 协议，其值应为多少？

以太帧中的 Type 字段为 0x0800，表示封装的是 IPv4 协议，查找资料知道 IPv6 的 Type 字段值为 0x86dd。

3.4.2 IP 报文结构分析

选择和上一个实验同样的数据帧，展开其 IP 层详细信息如下：

```

- Internet Protocol Version 4, Src: 10.11.56.37, Dst: 111.13.100.91
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0x33f7 (13303)
  ▶ Flags: 0x4000, Don't fragment
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0xf119 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.11.56.37
    Destination: 111.13.100.91

```

图 3.11 IP 数据帧分析

由上图可以分析，并回答实验提出的如下问题：

(1) 你使用的计算机的 IP 地址是什么？

从源地址中得到我的 IP 地址为 10.11.56.37。

(2) 在 IP 数据报首部，较高层协议字段中的值是什么？

高层协议字段 Protocol 为 1，表明是 ICMP 协议。

(3) IP 首部有多少字节？载荷字段有多少字节？

从 Header Length 字段中知道 IP 首部有 20 字节，从 Total Length 字段中知道总长度位 84 字节，所以载荷字段有 64 字节。

(4) 该 IP 数据报分段了没有？如何判断该 IP 数据报有没有分段？

没有，从 Flags 字段为 0x4000 可知，标志位为 010，即 DF 字段为 1，表示 Don't fragment 即不分段。当 DF 字段为 0 时表示分段。

(5) 关于高层协议有哪些有用信息？

展开高层协议的信息如下图：

```

- Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0xbaac [correct]
  [Checksum Status: Good]
  Identifier (BE): 30527 (0x773f)
  Identifier (LE): 16247 (0x3f77)
  Sequence number (BE): 1 (0x0001)
  Sequence number (LE): 256 (0x0100)
  [Response frame: 6]
  Timestamp from icmp data: Oct 15, 2018 09:17:37.000000000 CST
  [Timestamp from icmp data (relative): 0.260508477 seconds]
  - Data (48 bytes)
    Data: 8ef9030000000000101112131415161718191a1b1c1d1e1f...
    [Length: 48]

```

图 3.12 IP 数据帧分析

从图中可以看出，类型字段 Type 为 8，代码字段 Code 为 0，表示这是一个

ping 回显请求包，校验和为 0xbaac，校验正常，其余为一些表示符和序列号。

3.4.3 ARP 工作原理分析

停止俘获后，首先，在两台主机上分别查看 ARP 表，发现均多了对方主机的信息。如下图：

```
* panyue@Saltedfish ~ arp -a
? (192.168.1.103) at 00:c2:c6:b9:4d:80 [ether] on wlp3s0
_gateway (192.168.1.1) at f4:83:cd:56:c3:d2 [ether] on wlp3s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on wlp3s0
apollo.archlinux.org (138.201.81.199) at <incomplete> on docker0
apollo.archlinux.org (138.201.81.199) at <incomplete> on enp2s0
panyue@Saltedfish ~
```

图 3.13 主机 A 的 ARP 表

C:\windows\system32>arp -a		
接口:	Internet 地址	物理地址
接口: 192.168.217.1 --- 0x6		
Internet 地址	物理地址	类型
224.0.0.22	01-00-5e-00-00-16	静态
224.0.0.251	01-00-5e-00-00-fb	静态
接口: 192.168.19.1 --- 0x12		
Internet 地址	物理地址	类型
224.0.0.22	01-00-5e-00-00-16	静态
224.0.0.251	01-00-5e-00-00-fb	静态
接口: 192.168.1.103 --- 0x14		
Internet 地址	物理地址	类型
192.168.1.1	f4-83-cd-56-c3-d2	动态
192.168.1.108	10-f0-05-92-05-b3	动态
224.0.0.22	01-00-5e-00-00-16	静态
224.0.0.251	01-00-5e-00-00-fb	静态

图 3.14 主机 B 的 ARP 表

在 Wireshark 中筛选出 ARP 和 ICMP 报文，得到：

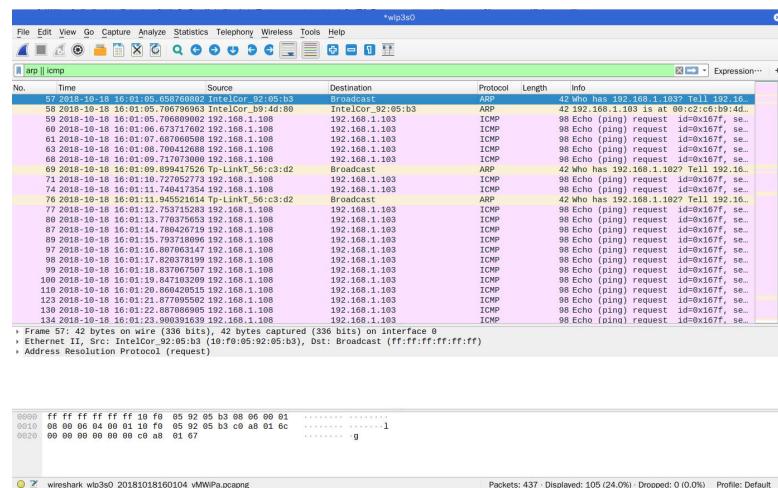


图 3.15 Wireshark 筛选 ARP 和 ICMP 报文

根据报文，对 ARP 协议执行的全过程如下分析：

1. 主机 A 发送 ARP 广播包，询问谁有 192.168.1.103

2. 主机 B 收到主机 A 的 ARP 请求，在自己的 ARP 表中记录主机 A 的 MAC 地址

3. 主机 B 发送 ARP 应答包，回复自己的 MAC 地址

4. 主机 A 收到主机 B 的 ARP 应答，在自己的 ARP 表中记录主机 B 的 MAC 地址

5. 现在，主机 A 和 B 的 ARP 表中都有了对方的 MAC 地址，于是可以进行正常的 ping 通信了。

3.4.4 集线器和交换机工作原理分析

首先，使用 Inspect 工具打开 PC 0 和 PC 1 的 ARP 表以及交换机的 MAC 表，结果观察不到任何 ARP 信息和 MAC 信息，将选择箭头移到交换机上，查看交换机端口及其接口 MAC 地址的摘要如下图：

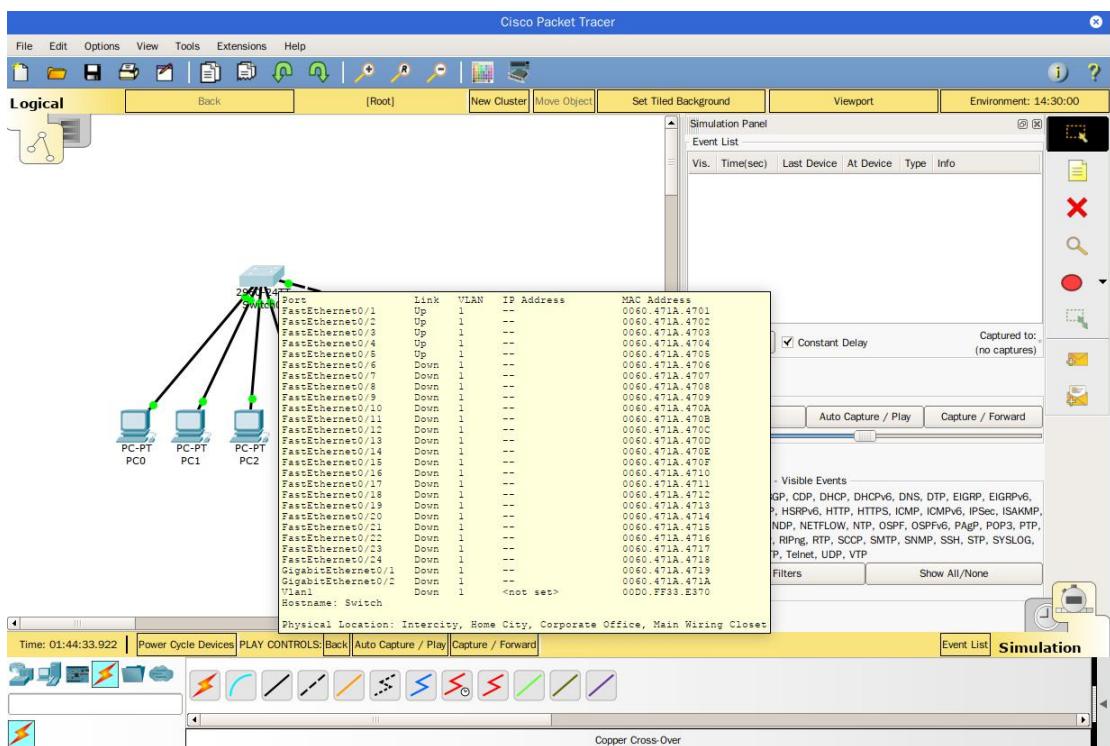


图 3.16 交换机接口 MAC 地址摘要

点击 Add Simple PDU，从 PC 0 发送一个 ping 到 PC 1 如下图，Event List 中可以看到一个 ICMP 回应和一个 ARP 请求。

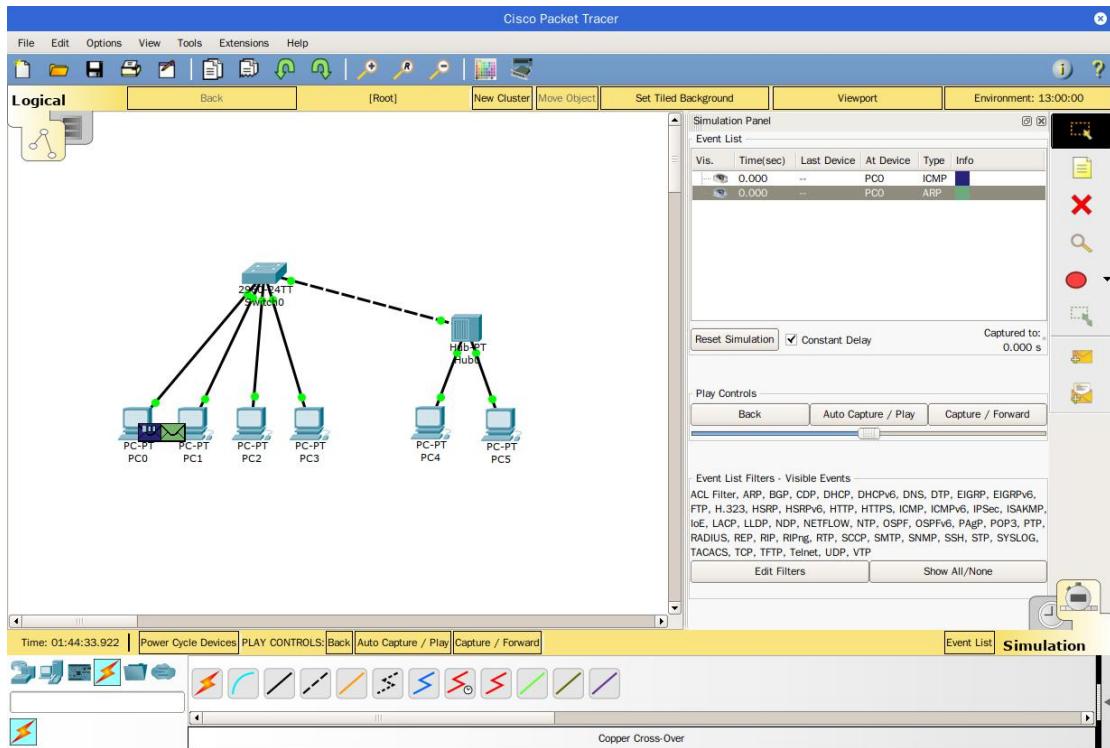


图 3.17 添加 PDU

开始逐步运行模拟，点击 Capture/Forward 按钮跟踪数据包的最终顺序。

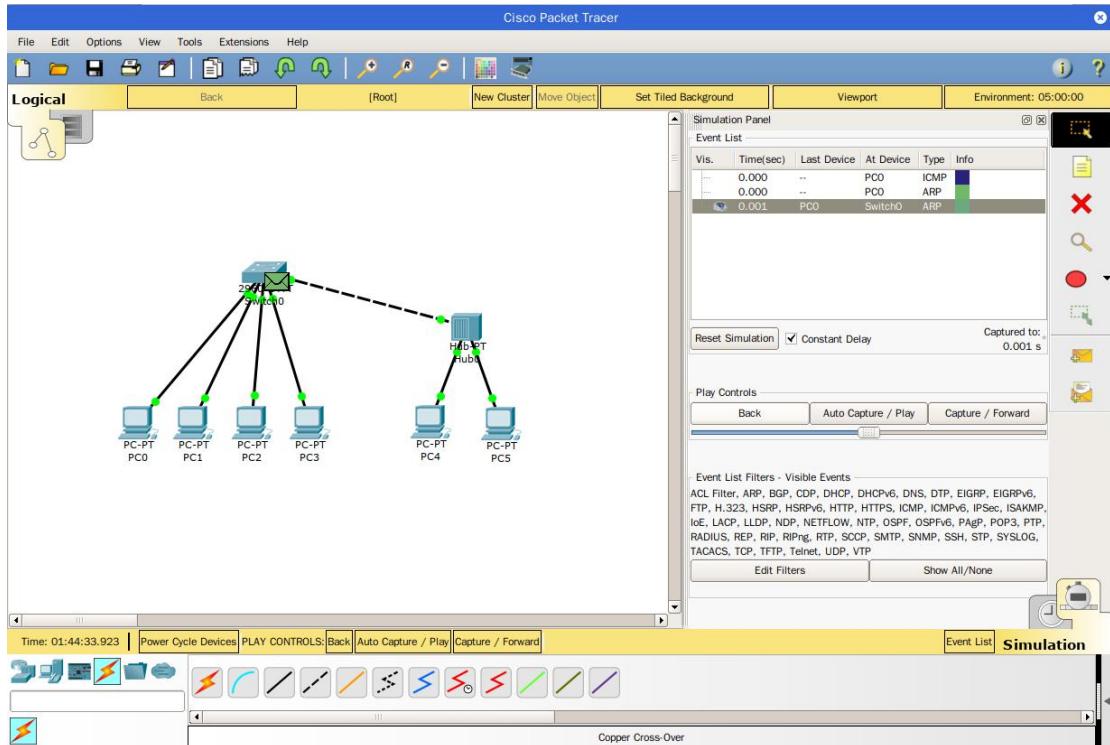


图 3.18 ARP 发送（一）

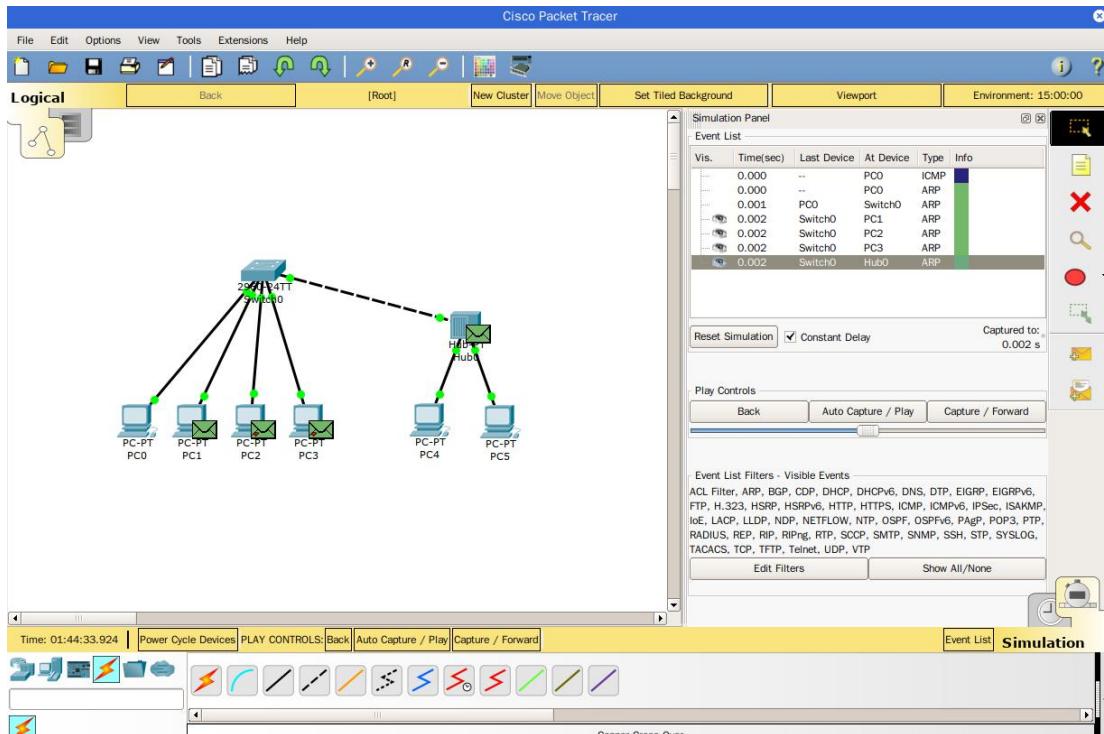


图 3.19 ARP 发送（二）

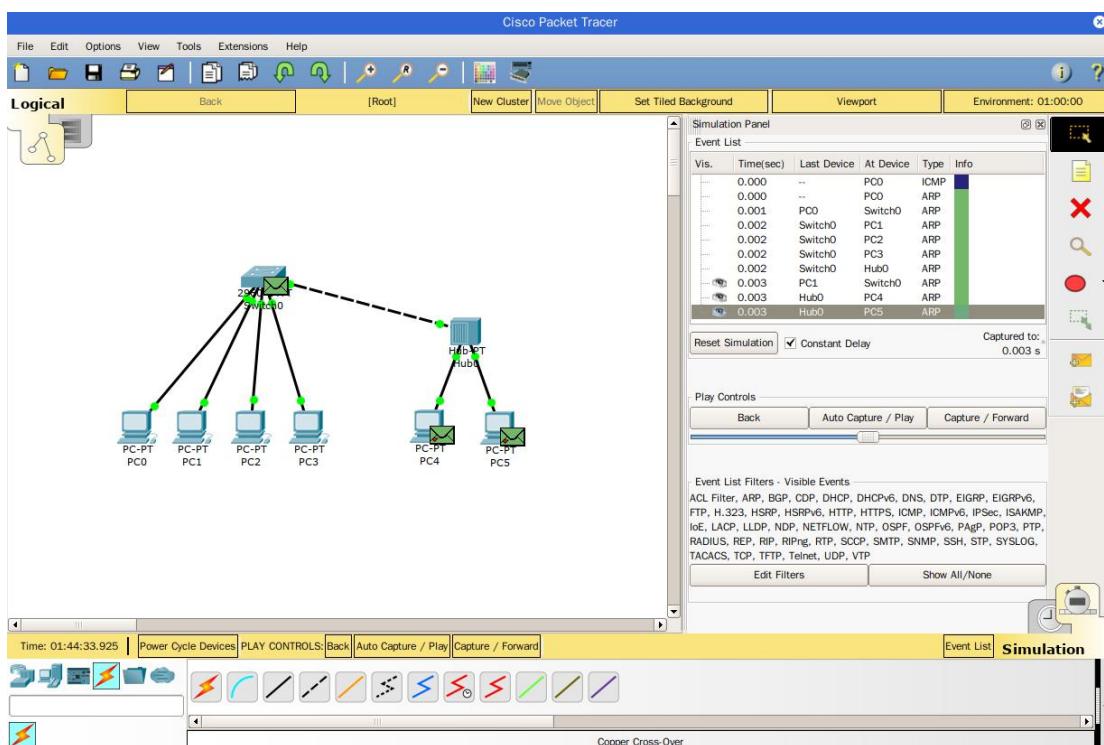


图 3.20 ARP 发送（三）

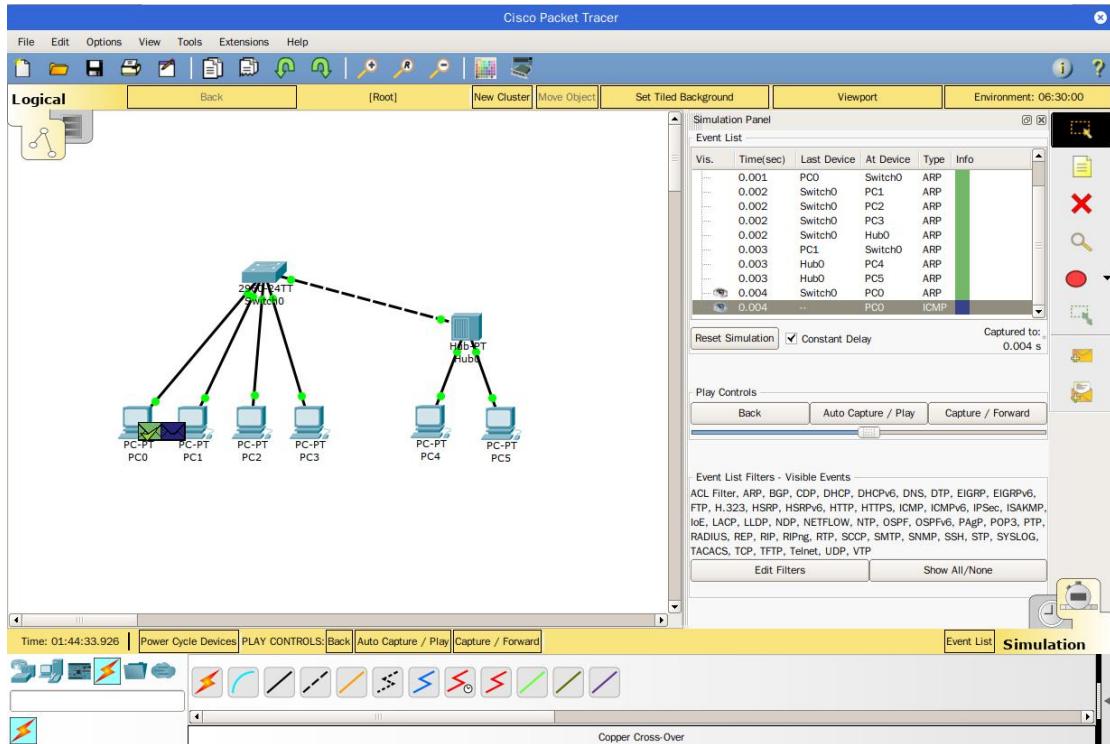


图 3.21 ARP 发送 (四)

经过这四步之后，查看 PC 0 和 PC 1 的 ARP 表，即可看到 ARP 表中已经有了相应表项。

ARP Table for PC0		
IP Address	Hardware Address	Interface
192.168.1.3	0050.0F9C.6848	FastEthernet0

图 3.22 PC 0 的 ARP 表

ARP Table for PC1		
IP Address	Hardware Address	Interface
192.168.1.2	0090.2B80.8C53	FastEthernet0

图 3.23 PC 1 的 ARP 表

从这个过程中可以观察到，ARP 请求始终是广播，交换机会将 ARP 请求从所有端口泛洪出去。

接下来观察交换机如何处理未知单播。

点击交换机，点击 CLI 选项卡，按几次 Enter 键，将会显示 Switch>提示。键入 enable 并按 Enter 键，提示会变为 Switch#，此时键入命令 clear mac-address-table dynamic，并按 Enter 键，可以消除交换机的 MAC 表，此时重新发送数据包，可以看到当 ICMP 请求到达交换机时，交换机仍会将其当做广播包广播出去，如下图所示：

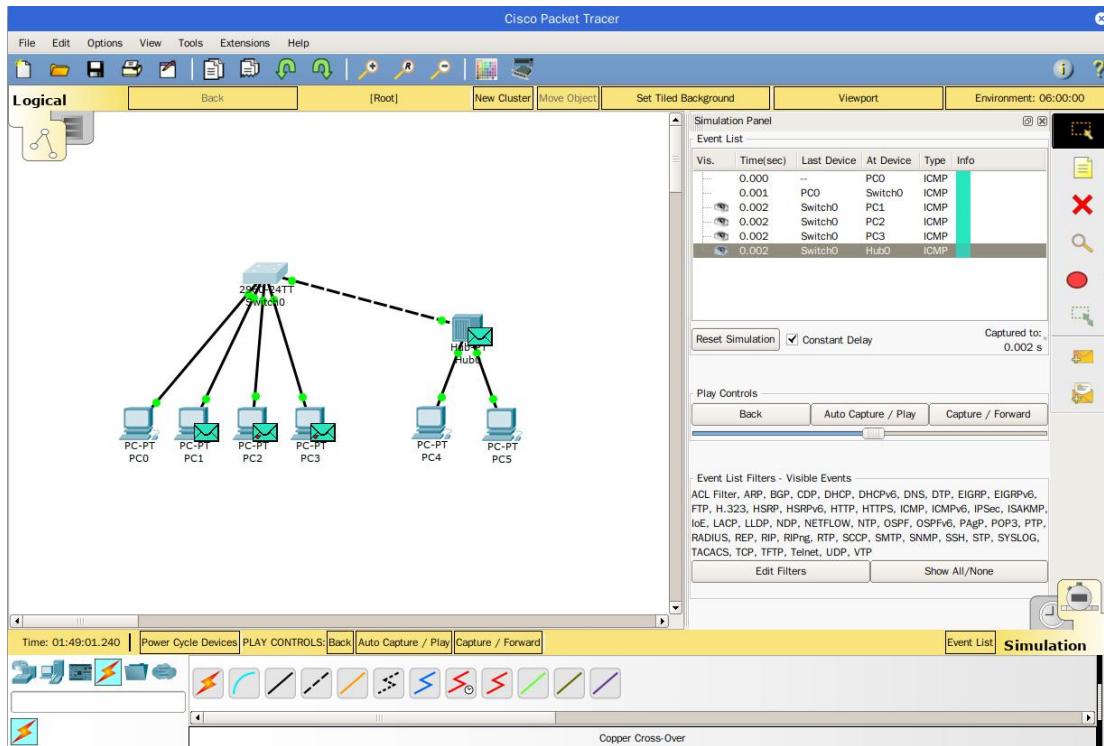


图 3.24 交换机广播请求

接下来观察集线器如何处理数据包。

点击 Add Simple PDU, 从 PC 5 发送一个 ping 到 PC 3, 集线器将该数据包从来源以外的地方广播了出去。

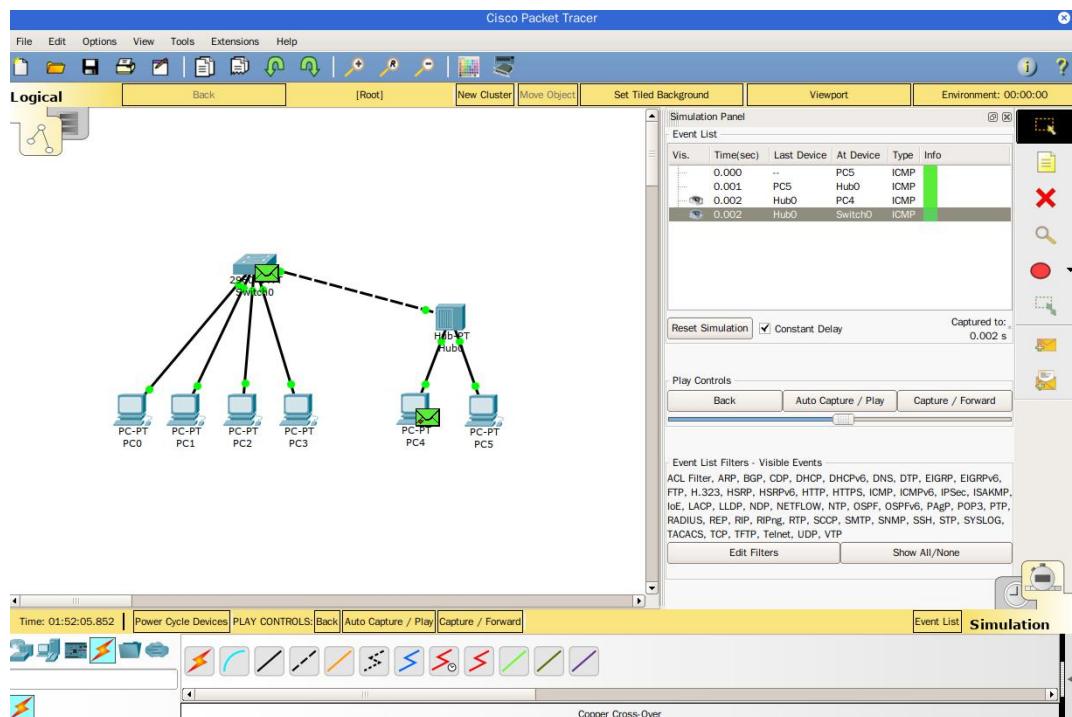


图 3.25 集线器广播请求

3.5 实验中的问题及心得

3.5.1 实验中的问题

为什么在寝室网络环境下做 ARP 实验时，ARP 请求没有经过路由器转发直接在两台主机之间通信了呢？

这是因为路由器的桥接”过程影响了 ARP 请求的传输，但是寝室路由器没有开启这项功能，所以没有途径路由器直接在两台主机间进行了数据的传输，完成了操作。

3.5.2 实验总结

本次实验需要动手的地方不是特别多，主要是对协议通信过程的分析和思考，通过本次实验，我更加深刻掌握了 ARP 的工作原理，同时也更加熟练地使用 PacketTracker，玩熟了交换机和集线器的配置，为接下来的实验打好了基础同时也强化了上课学习的知识。

实验五 配置路由器的路由选择协议

4.1 实验环境

1. 实验环境：运行 Arch Linux x86_64 操作系统的 PC 机一台
2. PacketTracer 版本：7.0.0.0202

4.2 实验目的

1. 深入理解路由器中路由选择协议的工作原理。
2. 能够配置路由器的路由选择协议 RIP。

4.3 实验内容及步骤

4.3.1 网络拓扑配置

打开 PacketTracer，配置网络拓扑如下图，其中 CISCO1841 路由器 3 台和 PC 两台。其中，需要注意的是，标配的 1841 路由器仅带有两个 10/100Mbps 的以太端口。这时，可以在路由器的物理设备视图中增加 WIC-1ENET 模块，从而增加一个 10Mbps 以太接口，以连接 PC。

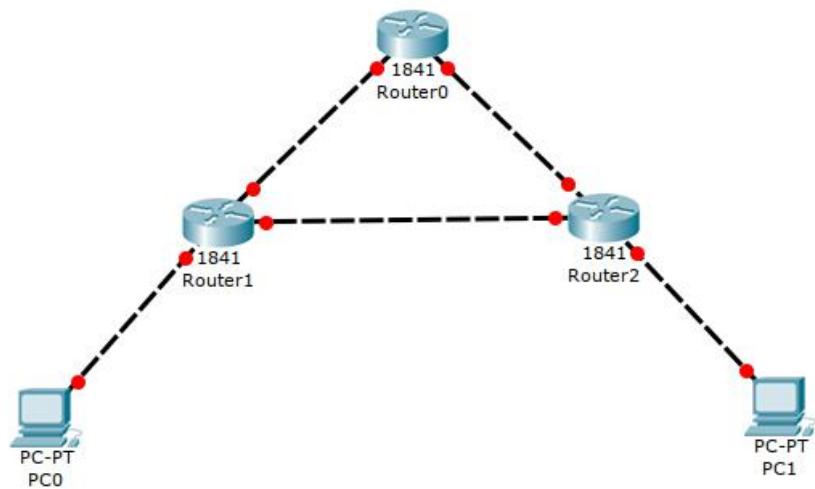


图 5.1 网络拓扑配置

刚配置好拓扑时，可以看到所有的灯均为红色，网络尚未接通。

5.3.2 规划 IP 地址并配置

规划好 IP 地址，对路由器和 PC 的各个端口配置 IP，如下图所示。完成配置之后，打开各个路由器的端口，此时网络接通。

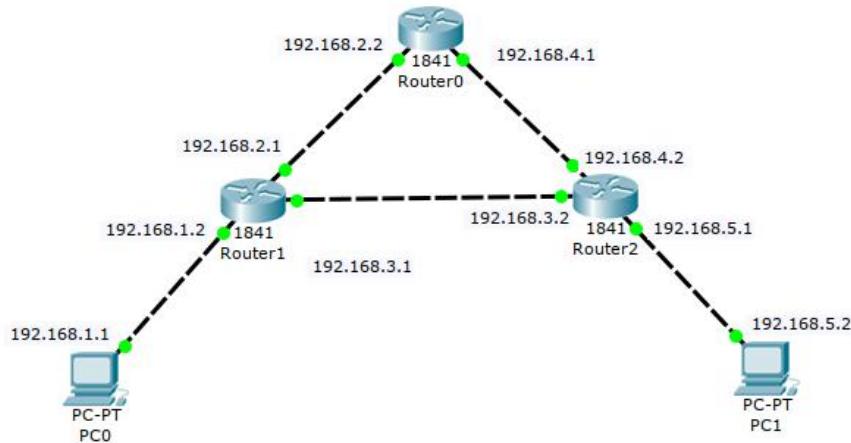


图 5.2 搭建简单的网络拓扑

这时检查一下网络是否可以正常工作，在 PC0 的命令提示符中键入 ping 192.168.5.2，发现无法 ping 通，输出如下：

```
C:\>ping 192.168.5.2

Pinging 192.168.5.2 with 32 bytes of data:

Reply from 192.168.1.2: Destination host unreachable.
Request timed out.
Reply from 192.168.1.2: Destination host unreachable.
Reply from 192.168.1.2: Destination host unreachable.

Ping statistics for 192.168.5.2:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

```

图 5.3 ping 命令输出

因此，我们还需要进一步的配置路由器的相关功能。

5.3.3 配置路由器选路协议

接下来为路由器配置 RIP 协议，点击路由器 1，然后点击 Config/ROUTING，如下图所示，添加表项。

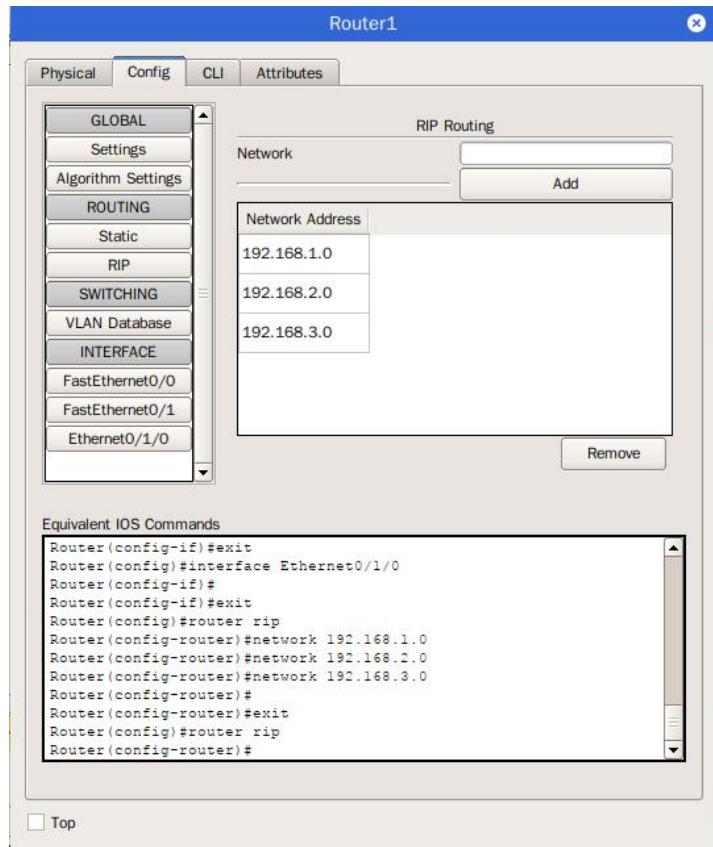


图 4.4 RIP 协议配置

接着对其他两个路由器也进行类似的配置。

配置完成后，再次在 PC0 的命令提示符中键入 ping 192.168.5.2，结果如下图：

```

C:\>ping 192.168.5.2

Pinging 192.168.5.2 with 32 bytes of data:

Reply from 192.168.5.2: bytes=32 time<1ms TTL=126
Reply from 192.168.5.2: bytes=32 time=1ms TTL=126
Reply from 192.168.5.2: bytes=32 time<1ms TTL=126
Reply from 192.168.5.2: bytes=32 time<1ms TTL=126

Ping statistics for 192.168.5.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms

C:\>

```

图 4.5 ping 命令输出（成功）

4.4 实验结果

4.4.1 检查路由器选路协议的作用

点击 Simulation，切换进模拟模式，点击 Add Simple PDU，从 PC0 到 PC1 发送一个 ICMP 包，然后点击 Captur/Forward，一步步观察这个 ICMP 包是怎样

发送过去的，结果如图（其中一步）：

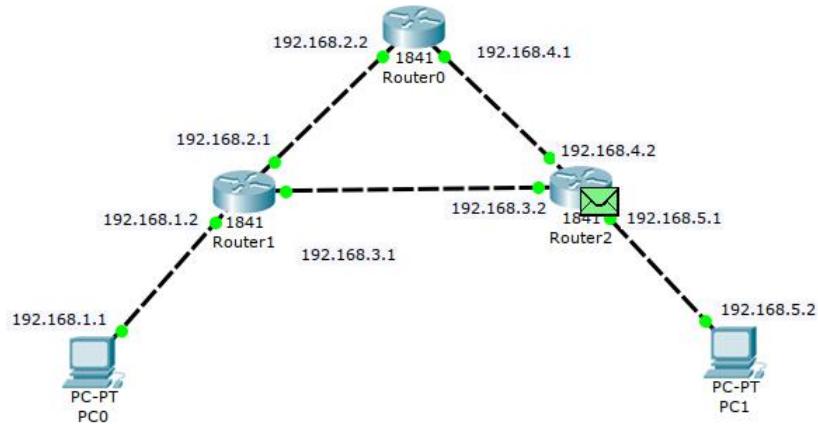


图 4.6 ICMP 包传递过程（一）

很明显，分组传输的路径是 PC0->Router1->Router2->PC1。

接下来，切断 Router1->Router2 的链路，重新观察发送，结果如下图：

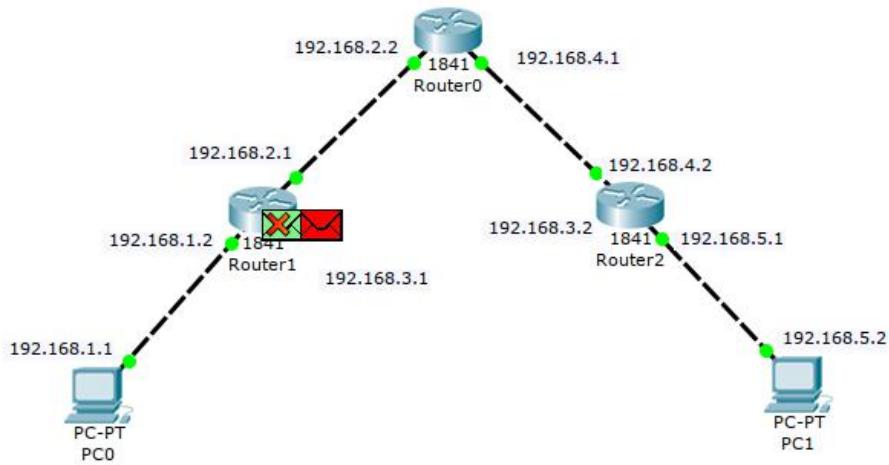


图 4.7 ICMP 包传递过程（二）

ICMP 包在第一次发送到 Router1 时，被路由器丢掉并发送一个 ICMP 回应。说明路由器无法转发这个分组，随后，路由器之间很快进行了 RIPv1 协议分组的交换，并更新了路由器的路由表，此后就可以发送 ICMP 包了，分组按照 PC0->Router1->Router0->Router2->PC1 的顺序传输。

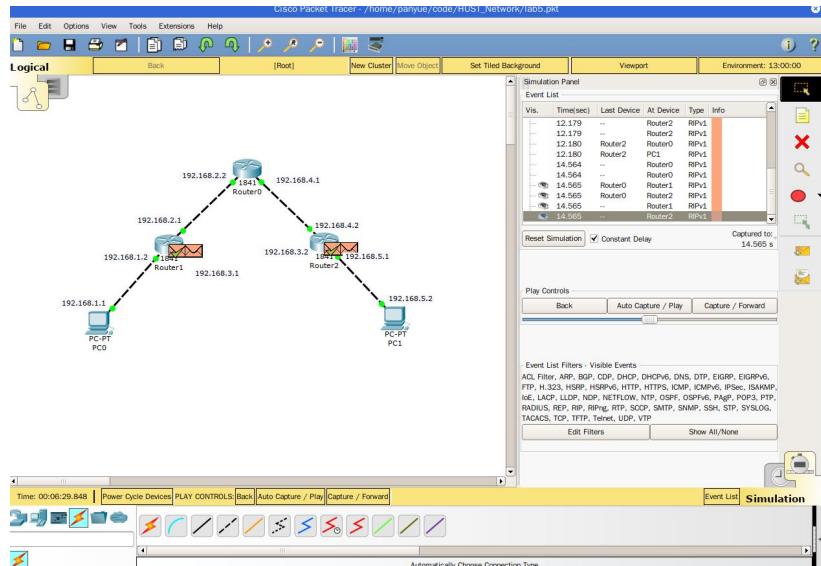


图 4.8 RIPv1 分组传递

4.4.2 三个路由器的路由表

最后，三个路由器的路由表如下：

Routing Table for Router1				
Type	Network	Port	Next Hop IP	Metric
C	192.168.1.0/24	FastEthernet0/0	---	0/0
C	192.168.2.0/24	FastEthernet0/1	---	0/0
R	192.168.4.0/24	FastEthernet0/1	192.168.2.2	120/1
R	192.168.5.0/24	FastEthernet0/1	192.168.2.2	120/2

图 4.9 路由器 1 的路由表

由表项可知，目的子网为 192.168.4.0/24 和 192.168.5.0/24 的分组的下一跳是 192.168.2.2

Routing Table for Router2				
Type	Network	Port	Next Hop IP	Metric
R	192.168.1.0/24	FastEthernet0/1	192.168.4.1	120/2
R	192.168.2.0/24	FastEthernet0/1	192.168.4.1	120/1
C	192.168.4.0/24	FastEthernet0/1	---	0/0
C	192.168.5.0/24	FastEthernet0/0	---	0/0

图 4.10 路由器 2 的路由表

由表项可知，目的子网为 192.168.1.0/24 和 192.168.2.0/24 的分组的下一跳是 192.168.4.1

Routing Table for Router0					
Type	Network	Port	Next Hop IP	Metric	
R	192.168.1.0/24	FastEthernet0/0	192.168.2.1	120/1	
C	192.168.2.0/24	FastEthernet0/0	---	0/0	
R	192.168.3.0/24	FastEthernet0/1	192.168.4.2	120/16	
C	192.168.4.0/24	FastEthernet0/1	---	0/0	
R	192.168.5.0/24	FastEthernet0/1	192.168.4.2	120/1	

图 4.11 路由器 3 的路由表

由表项可知，目的子网为 192.168.1.0/24 的分组的下一跳是 192.168.2.1

目的子网为 192.168.5.0/24 的分组的下一跳是 192.168.4.2

4.5 实验中的问题及心得

4.5.1 实验中的问题

1. 路由器接口不够的情况下，如何增加？

可以给路由器添加不同的模块，利用模块上带有的接口来增加可用接口。

2. 网络配置完成后，如果链接显示为红色，如何解决？

打开路由器的端口，将每个端口的 Port Status 后面的 ON 打钩，链接即可显示为绿色。

3. IP 层连通的条件是什么？

1. 每个设备被配置了正确的 IP 和子网掩码
2. 每台主机设置了默认网关
3. 每台路由器有正确的路由表项

4.5.2 实验总结

本次实验学习路由选择协议 RIP，整体非常简单。就是在实验的过程中，RIP 报文只在一开始行发送，当结果收敛时便不再发送，所以观察上有点麻烦，只有反复修改网络结构，让其 RIP 表改变才能观察到 RIP 报文。通过本次实验，我很好的理解了 RIP 的工作原理和方式，更加熟练了通过模拟模式下步传递演示来观察网络传输的细节的能力。

实验六 运输层实验

5.1 实验环境

1. 实验环境：运行 Arch Linux x86_64 操作系统的 PC 机一台

2. PacketTracer 版本: 7.0.0.0202

5.2 实验目的

1. 理解运输层的端口与应用层的进程之间的关系。
2. 了解端口号的划分和分配。
3. 熟悉 UDP 与 TCP 协议的主要特点及支持的应用协议。
4. 理解 UDP 的无连接通信与 TCP 的面向连接通信。
5. 熟悉 TCP 报文段和 UDP 报文的数据封装格式。
6. 熟悉 TCP 通信的三个阶段。
7. 理解 TCP 连接建立过程和 TCP 连接释放过程。

5.3 实验内容及步骤

5.3.1 网络拓扑配置

打开 PacketTracer，配置网络拓扑如下图。

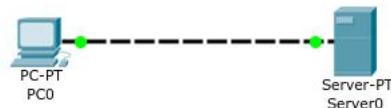


图 5.1 网络拓扑配置

其中，各台主机配置如下：

PC0: IP:192.168.1.2; Gateway:192.168.1.254; DNS Server:192.168.1.1

Server0: IP:192.168.1.2; Gateway:192.168.1.254

同时，检查 Server 端的服务配置，HTTP 和 DNS 的服务都需要打开。

5.3.2 运输层端口观察实验--事件捕获

点击 PC0/Desktop，点击浏览器 Web Browser，输入域名如下：



图 5.2 浏览器输入域名

不断点击 Capture/Forward，可以观察到，PC0 和 Server 之间经历了 ARP 协议获取 MAC 地址，DNS 协议解析域名，TCP 协议建立连接，HTTP 协议传输文本的整个过程，最后 Event List 中的事件如下：

Simulation Panel					
Event List					
Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.000	--	PC0	DNS	
	0.001	PC0	Server0	DNS	
	0.002	Server0	PC0	DNS	
	0.002	--	PC0	TCP	
	0.003	PC0	Server0	TCP	
	0.004	Server0	PC0	TCP	
	0.004	--	PC0	HTTP	
	0.005	PC0	Server0	TCP	
	0.005	--	PC0	HTTP	
0.006	PC0	Server0	HTTP		

Reset Simulation Constant Delay Captured to: 0.080 s

图 5.3 事件列表

可以观察到，在经历很多次 HTTP 报文传输之后，浏览器上最终出现了我们请求的结果。

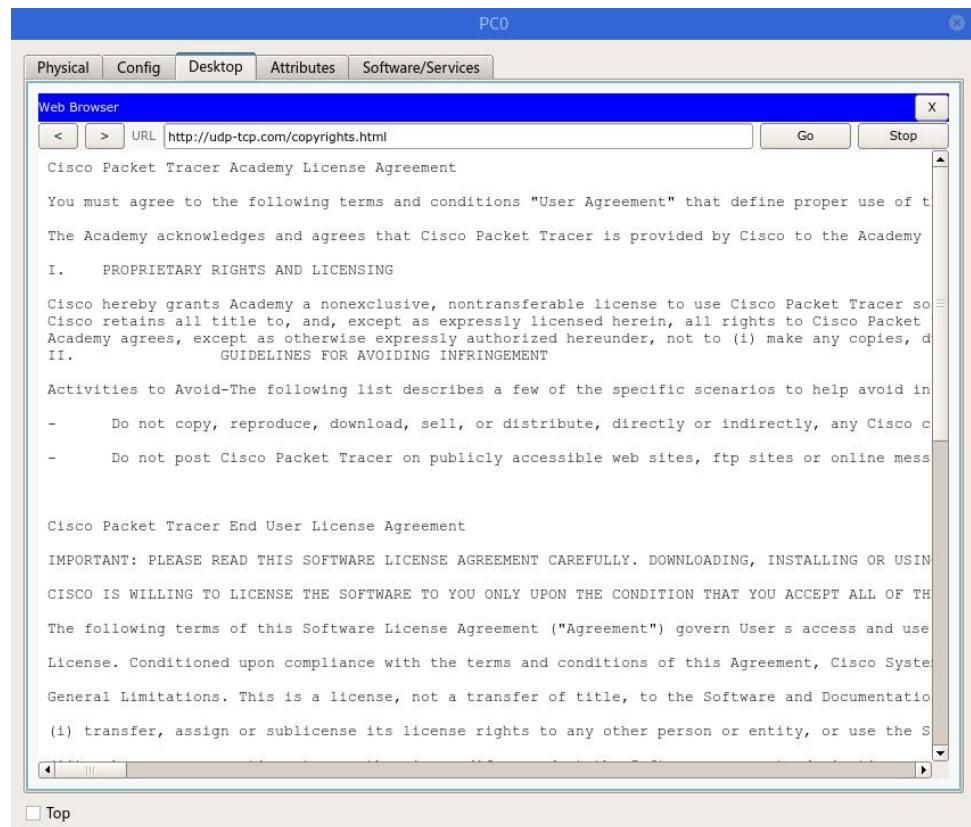


图 5.4 浏览器请求结果

有了事件列表信息，我们就可以分析 UDP/TCP 中的端口号了。

5.3.3 UDP 和 TCP 协议的对比分析

单击 Reset Simulation，重新进行一次浏览器请求，并捕获所有的报文。

5.3.4 TCP 连接管理分析

点击 Server，在 HTTP 服务中新建一个 test.html，里面内容只写上 Hello world！作为实验文件。在 PC0 的浏览器中请求这个文件，在 Event List 中查看捕获到的事件，如下图：

Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.000	--	PC0	DNS	
	0.001	PC0	Server0	DNS	
	0.002	Server0	PC0	DNS	
	0.002	--	PC0	TCP	
	0.003	PC0	Server0	TCP	
	0.004	Server0	PC0	TCP	
	0.004	--	PC0	HTTP	
	0.005	PC0	Server0	TCP	
	0.005	--	PC0	HTTP	
	0.006	PC0	Server0	HTTP	

Reset Simulation Constant Delay Captured to:
140.896 s

图 5.5 Event List 结果（一）

Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.004	--	PC0	HTTP	
	0.005	PC0	Server0	TCP	
	0.005	--	PC0	HTTP	
	0.006	PC0	Server0	HTTP	
	0.007	Server0	PC0	HTTP	
	0.007	--	PC0	TCP	
	0.008	PC0	Server0	TCP	
	0.009	Server0	PC0	TCP	
	0.010	PC0	Server0	TCP	

Reset Simulation Constant Delay Captured to:
140.896 s

图 5.6 Event List 结果（二）

从途中我们可以清晰的看到 TCP 连接时的三次握手过程，接下来在实验结果中对捕获到的数据包进行具体分析。

5.4 实验结果

5.4.1 通过捕获的 DNS 事件查看并分析 UDP 的端口号

在 Event List 中点击数据包，可以查看其详细信息，其中 DNS 请求和应答的详细信息如下：

UDP		31 Bits	
0	16		
SRC PORT: 1040	DEST PORT: 53		
LENGTH: 0x23	CHECKSUM: 0x0		
DATA (VARIABLE)			

DNS Header		31 Bits				
0	1	5	8 9	12	15	Bits
ID						
OPCODE	A A	T C	R D	R A	Z	RCODE
QDCOUNT: 1						
ANCOUNT: 0						
NSCOUNT: 0						
ARCOUNT: 0						

DNS Query		31 Bits	
0	16		
NAME: udp-tcp.com			
TYPE: 0x0001	CLASS: 0x0001		

图 5.7 DNS 请求报文

UDP		31 Bits	
0	16		
SRC PORT: 53	DEST PORT: 1040		
LENGTH: 0x3c	CHECKSUM: 0x0		
DATA (VARIABLE)			

DNS Header		31 Bits				
0	1	5	8 9	12	15	Bits
ID						
OPCODE	A A	T C	R D	R A	Z	RCODE
QDCOUNT: 1						
ANCOUNT: 1						
NSCOUNT: 0						
ARCOUNT: 0						

DNS Query		31 Bits	
0	16		
NAME: udp-tcp.com			
TYPE: 0x0001	CLASS: 0x0001		

图 5.8 DNS 应答报文

观察发现 PC0 上的端口是 1040, Server 上的端口是 53, 在一次 DNS 请求中是成对出现, 没有变化的。此时 PC0 相当于客户端, Server 相当于服务器, 因为 Server 使用的是 DNS 的熟知端口 53。

重新回到 PC 机的浏览器窗口单击 Go 按钮再次请求相同的网页, 再次捕获得到的结果如下:

UDP		31 Bits	
0	16		
SRC PORT: 1041	DEST PORT: 53		
LENGTH: 0x23	CHECKSUM: 0x0		
DATA (VARIABLE)			

DNS Header		31 Bits				
0	1	5	8 9	12	15	Bits
ID						
OPCODE	A A	T C	R D	R A	Z	RCODE
QDCOUNT: 1						
ANCOUNT: 0						
NSCOUNT: 0						
ARCOUNT: 0						

DNS Query		31 Bits	
0	16		
NAME: udp-tcp.com			
TYPE: 0x0001	CLASS: 0x0001		

图 5.9 DNS 请求报文

UDP		31 Bits	
0	16		
SRC PORT: 53	DEST PORT: 1041		
LENGTH: 0x3c	CHECKSUM: 0x0		
DATA (VARIABLE)			

DNS Header		31 Bits				
0	1	5	8 9	12	15	Bits
ID						
OPCODE	A A	T C	R D	R A	Z	RCODE
QDCOUNT: 1						
ANCOUNT: 1						
NSCOUNT: 0						
ARCOUNT: 0						

DNS Query		31 Bits	
0	16		
NAME: udp-tcp.com			
TYPE: 0x0001	CLASS: 0x0001		

图 5.10 DNS 应答报文

观察发现, 此时 PC0 上使用的端口变成了 1041, 于是猜测 DNS 客户端使用端口的规律是依次递增 1。

于是进行多次浏览器请求/捕获的操作，最后观察端口从 1042,1043 开始逐渐递增，验证了这个猜测。

而服务端上使用的端口始终是周知端口 53。

5.4.2 通过捕获的 HTTP 事件查看并分析 TCP 的端口号

在 Event List 中点击数据包，可以查看其详细信息，其中选取一对 HTTP 请求和应答报文，如下图：

TCP		16	31 Bits
SRC PORT: 1033	DEST PORT: 80		
SEQUENCE NUM: 1			
ACK NUM: 1			
OFF.	RES.	PSH + ACK	WINDOW
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

HTTP

```
Get /copyrights.html HTTP/1.1
Accept-Language: en-us
Accept: /*
Connection: close
Host: udp-tcp.com
```

图 5.11 HTTP 请求报文

TCP		16	31 Bits
SRC PORT: 80	DEST PORT: 1033		
SEQUENCE NUM: 1			
ACK NUM: 116			
OFF.	RES.	ACK	WINDOW
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

图 5.12 HTTP 应答报文

观察发现 PC0 上的端口是 1033，Server 上的端口是 80。此时 PC0 相当于客户端，Server 相当于服务器，因为 Server 使用的是 TCP 的周知端口 80。

5.4.3 分析运输层端口号

4.1 中 DNS 服务器端的端口号和 4.2 中服务器端的端口号并不相同，这是因为端口号在网络中唯一标识进程，DNS 服务器和 HTTP 服务器是两个不同的进程，因此端口号不会相同。

重新捕获 HTTP 事件，发现此时 PC0 上的端口由 1033 变成了 1034，反复操作发现端口号依次递增。于是归纳得到，运输层动态端口号的分配是从第一个端口号开始依次递增 1 的。

5.4.4 观察 UDP 与 TCP 协议的工作模式

首先观察 UDP 的无连接的工作模式。

运输层的 UDP 发送 DNS 请求之前，没有建立连接，一开始就发送了 DNS 报文，如下图：

Event List					
Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.000	--	PC0	DNS	
	0.001	PC0	Server0	DNS	
	0.002	Server0	PC0	DNS	
	0.002	--	PC0	TCP	
	0.003	PC0	Server0	TCP	
	0.004	Server0	PC0	TCP	
	0.004	--	PC0	HTTP	
	0.005	PC0	Server0	TCP	
	0.005	--	PC0	HTTP	
	0.006	PC0	Server0	HTTP	

图 5.13 Event List (一)

UDP 首部中 LENGTH 字段的值为 0x23，即 35，而 UDP 首部有 8 个字节，因此其数据部分的长度为 27 个字节。

对于 TCP，最后一个 HTTP 事件之后的 Event List 如下图：

Event List					
Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.083	--	Server0	HTTP	
	0.084	--	Server0	HTTP	
	0.084	Server0	PC0	HTTP	
	0.084	--	Server0	HTTP	
	0.085	Server0	PC0	HTTP	
	0.085	--	PC0	TCP	
	0.086	PC0	Server0	TCP	
	0.087	Server0	PC0	TCP	
	0.088	PC0	Server0	TCP	

图 5.14 Event List (二)

从上面两张图可以看到，在第一个 HTTP 事件之前和最后一个 HTTP 事件之后，都存在 TCP 事件，记录最后两个 HTTP 报文如下图：

IP		31 Bits	
0	4 8 16 19	TL:	576
4	IHL	DSCP: 0x0	
ID: 0xb2	0x2	0x0	
TTL: 128	PRO: 0x6	CHKSUM	
SRC IP: 192.168.1.1			
DST IP: 192.168.1.2			
OPT: 0x0	0x0		
DATA (VARIABLE LENGTH)			

IP		31 Bits	
0	4 8 16 19	TL:	237
4	IHL	DSCP: 0x0	
ID: 0xb3	0x2	0x0	
TTL: 128	PRO: 0x6	CHKSUM	
SRC IP: 192.168.1.1			
DST IP: 192.168.1.2			
OPT: 0x0	0x0		
DATA (VARIABLE LENGTH)			

TCP		31 Bits	
0	16	31 Bits	
SRC PORT: 80	DEST PORT: 1041		
SEQUENCE NUM: 13901			
ACK NUM: 116			
OFF. RES. ACK	WINDOW		
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

TCP		31 Bits	
0	16	31 Bits	
SRC PORT: 80	DEST PORT: 1041		
SEQUENCE NUM: 14457			
ACK NUM: 116			
OFF. RES. PSH + ACK	WINDOW		
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

图 5.15 第一个 HTTP 报文

图 5.16 最后一个 HTTP 报文

两个报文之间的 sequence number 之差为 556，刚好是发过去的报文的 data

length 的大小（利用 IP 长度 576 减去首部 20 字节得到）

5.4.5 TCP 连接建立阶段的三次握手分析

点击 Event List 中的数据包，得到三次握手中的 TCP 数据包的详细信息如下图：

IP										31 Bits	
0	4	8	16	19							
4	IHL	DSCP: 0x0		TL: 44							
	ID: 0xf2	0x2		0x0							
TTL: 128	PRO: 0x6			CHKSUM							
	SRC IP: 192.168.1.2										
	DST IP: 192.168.1.1										
	OPT: 0x0			0x0							
	DATA (VARIABLE LENGTH)										

TCP										31 Bits	
0	16										
	SRC PORT: 1044		DEST PORT: 80								
	SEQUENCE NUM: 0										
	ACK NUM: 0										
OFF.	RES.	SYN		WINDOW							
	CHECKSUM: 0x0		URGENT POINTER								
	OPTION		PADDING								
	DATA (VARIABLE)										

图 5.17 TCP 连接三次握手（一）

IP										31 Bits	
0	4	8	16	19							
4	IHL	DSCP: 0x0		TL: 44							
	ID: 0x1c0	0x2		0x0							
TTL: 128	PRO: 0x6			CHKSUM							
	SRC IP: 192.168.1.1										
	DST IP: 192.168.1.2										
	OPT: 0x0			0x0							
	DATA (VARIABLE LENGTH)										

TCP										31 Bits	
0	16										
	SRC PORT: 80		DEST PORT: 1044								
	SEQUENCE NUM: 0										
	ACK NUM: 1										
OFF.	RES.	SYN + ACK		WINDOW							
	CHECKSUM: 0x0		URGENT POINTER								
	OPTION		PADDING								
	DATA (VARIABLE)										

图 5.18 TCP 连接三次握手（二）

IP										31 Bits	
0	4	8	16	19							
4	IHL	DSCP: 0x0		TL: 40							
	ID: 0xf3	0x2		0x0							
TTL: 128	PRO: 0x6			CHKSUM							
	SRC IP: 192.168.1.2										
	DST IP: 192.168.1.1										
	OPT: 0x0			0x0							
	DATA (VARIABLE LENGTH)										

TCP										31 Bits	
0	16										
	SRC PORT: 1044		DEST PORT: 80								
	SEQUENCE NUM: 1										
	ACK NUM: 1										
OFF.	RES.	ACK		WINDOW							
	CHECKSUM: 0x0		URGENT POINTER								
	OPTION		PADDING								
	DATA (VARIABLE)										

图 5.19 TCP 连接三次握手（三）

分析 TCP 连接的建立如下：

1. PC0 发给 Server 一个 TCP SYN 报文，其中 SEQ 和 ACK 均为 0。
 2. Server 令 ACK 等于收到的上一个报文的 SEQ 值+1，然后令自己的 SEQ 为 0，发送一个 TCP SYN+ACK 报文给 PC0。
 3. PC0 收到报文之后，令 SEQ 为自己上一次发送的报文的 SEQ+1，ACK 等于收到的上一个报文的 SEQ+1，发送一个 TCP ACK 报文给 Server。
- 完成上述步骤之后，一个 TCP 连接就建立了。

分析 TCP 连接的状态变迁如下：

1. 开始时，PC0 和 Server 都处于 CLOSED 状态。
2. Server 被动打开，处于 LISTEN 状态。
3. PC0 发送一个 SYN 报文之后，进入 SYS_SENT 状态。
4. Server 收到 SYN 报文之后，进入 SYN_RECV 状态，并向 PC0 发送 SYN+ACK 报文。
5. PC0 收到 SYN+ACK 报文之后，进入 ESTABLISHED 状态，并向 Server 发送 ACK 报文。
6. Server 收到 ACK 报文之后，进入 ESTABLISHED 状态，连接建立。

5.4.6 TCP 连接释放阶段的四次握手分析

点击 Event List 中的数据包，得到三次握手中的 TCP 数据包的详细信息如下图：

IP		31 Bits	
0	4	8	16 19
4	IHL	DSCP: 0x0	TL: 40
ID: 0x101	0x2	0x0	
TTL: 128	PRO: 0x6	CHKSUM	
SRC IP: 192.168.1.2			
DST IP: 192.168.1.1			
OPT: 0x0	0x0		
DATA (VARIABLE LENGTH)			

IP		31 Bits	
0	4	8	16 19
4	IHL	DSCP: 0x0	TL: 40
ID: 0x1cc	0x2	0x0	
TTL: 128	PRO: 0x6	CHKSUM	
SRC IP: 192.168.1.1			
DST IP: 192.168.1.2			
OPT: 0x0	0x0		
DATA (VARIABLE LENGTH)			

TCP		31 Bits	
0	16	31 Bits	
SRC PORT: 1045	DEST PORT: 80		
SEQUENCE NUM: 110			
ACK NUM: 138			
OFF. RES. FIN + ACK	WINDOW		
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

TCP		31 Bits	
0	16	31 Bits	
SRC PORT: 80	DEST PORT: 1045		
SEQUENCE NUM: 138			
ACK NUM: 111			
OFF. RES. FIN + ACK	WINDOW		
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

图 5.20 TCP 连接释放四次握手（一）

图 5.21 TCP 连接释放四次握手（二）

IP		31 Bits	
0	4	8	16 19
4	IHL	DSCP: 0x0	TL: 40
ID: 0x102	0x2	0x0	
TTL: 128	PRO: 0x6	CHKSUM	
SRC IP: 192.168.1.2			
DST IP: 192.168.1.1			
OPT: 0x0	0x0		
DATA (VARIABLE LENGTH)			

TCP		31 Bits	
0	16	31 Bits	
SRC PORT: 1045	DEST PORT: 80		
SEQUENCE NUM: 111			
ACK NUM: 138			
OFF. RES. ACK	WINDOW		
CHECKSUM: 0x0	URGENT POINTER		
OPTION	PADDING		
DATA (VARIABLE)			

图 5.22 TCP 连接释放四次握手（三）

分析 TCP 连接的建立如下：

1. PC0 发给 Server 一个 TCP FIN 报文，其中 SEQ 和 ACK 分别为 110 和 138。
 2. Server 令 ACK 等于收到的上一个报文的 SEQ 值+1，然后令自己的 SEQ 为 138，发送一个 TCP ACK 报文给 PC0。
 3. Server 发给 PC0 一个 TCP FIN+ACK 报文，SEQ 和 ACK 和上一个发送的 ACK 报文一致。
 4. PC0 收到报文之后，令 SEQ 为自己上一次发送的报文的 SEQ+1，ACK 等于收到的上一个报文的 SEQ+1，发送一个 TCP ACK 报文给 Server。
- 完成上述步骤之后，一个 TCP 连接就释放了。
在这里，步骤 2 和步骤 3 被合并为一个数据包一起发送了。

分析 TCP 连接的状态变迁如下：

1. 开始时，PC0 和 Server 都处于 ESTABLISHED 状态。
2. PC0 向 Server 发送 FIN 报文之后，进入 FIN_WAIT_1 状态。
3. Server 收到 PC0 发送的 FIN 报文之后，进入 CLOSE_WAIT 状态。
4. Server 向 PC0 发送 FIN+ACK 报文之后，进入 LAST_ACK 状态。
5. PC0 收到 Server 发送的 FIN+ACK 报文之后，进入 CLOSING 状态，并向 Server 发送一个 ACK 报文，进入 TIME_WAIT 状态。
6. Server 收到 PC0 发来的 ACK 报文之后，进入 CLOSED 状态。
7. PC0 过一段时间（一般是两倍的报文寿命）之后，进入 CLOSED 状态，连接完全释放。

5.5 实验中的问题及心得

5.5.1 实验中的 5.问题

1. 运输层如何区分应用层的不同进程？

通过端口来唯一标识进程，以此来区分不同的进程。

2. 若使用 Reset Simulation 按钮后再重新进程捕获，端口号如何变化？新的值与与充值前有关吗？

有关，新的值仍然是之前的端口值加一，该值由系统维护，并不会因为重置模拟而重置。

3. TCP 报文首部中的序号和确认号有什么作用？

可以为字节流提供可靠的传输服务。

4. 无连接的 UDP 和面向连接的 TCP 各有什么优缺点？

无连接的 UDP 可以以任何速率向网络层注入数据，在能够容忍少量的数据

丢失，并不需要可靠的运输服务的实时应用中可以避开 TCP 拥塞控制机制和通信开销的不利影响。而 TCP 则为网络传输提供了可靠的传输服务，能够为进程数据传输提供无差错，按适当顺序交付的服务。TCP 还提供了拥塞控制服务，有利于提高因特网的整体性能，在电子邮件，文档下载等应用中使用效果很好。

5. 连接建立阶段的第一次握手是否需要消耗一个序号？其 SYN 报文段是否携带数据？为什么？第二次握手呢？

从图 17 中可以看到，TCP 连接建立阶段的第一次握手需要消耗一个序号，但 SYN 报文段不需要携带数据，第二个报文段也同样要消耗一个序号，都不能携带数据。

6. 本实验中连接释放过程的第二、三次握手是同时进行的还是分开进行的？这两次握手何时需要分开进行？

本实验中第二次和第三次握手是同时进行的。当 PC0 释放连接之后，Server 仍要发送数据时，第二次和第三次握手就要分开进行。

7. 本实验中连接释放阶段的第四次握手，PC 向 Server 发送最后一个 TCP 确认报文段后，为什么不是直接进入 CLOSED 状态，而是进入 CLOSING 连接状态？

这是因为，此时 TCP 出现了一种同时关闭的情况。即 PC0 在接受到 Server 发送的 FIN，并发送了 ACK 之后，还没有接收到 Server 对自己的 FIN 的 ACK，此时就进入了 CLOSING 状态。这也与实际情况相符，一次 HTTP 请求结束，客服端和服务器同时关闭 TCP 连接。

8. 本实验中 TCP 连接建立后的数据通信阶段，PC 向 Server 发送的了多少数据？Server 向 PC 发送的数据呢？

HTTP 请求和应答的报文如下图：

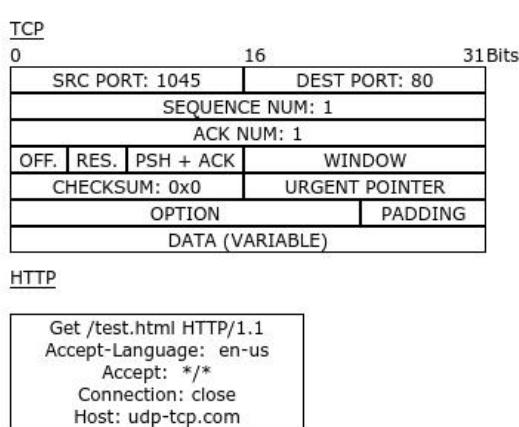


图 5.23 HTTP 请求

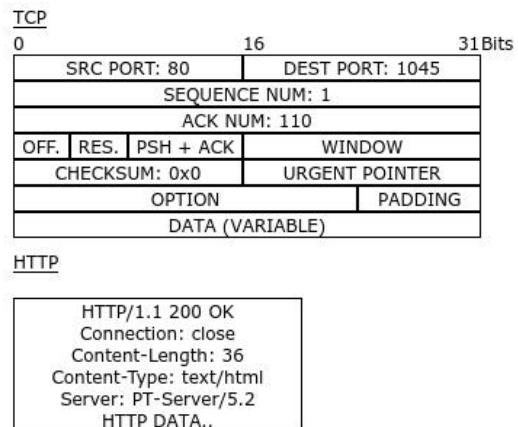


图 5.24 HTTP 应答

将上述两个报文的 Sequence Num 和 TCP 连接释放报文的 Sequence Num 相比，即可计算 PC 向 Server 发送的数据为长度 $110 - 1 = 109$ 字节，Server 向 PC 发送的数据长度为 $138 - 1 = 137$ 字节。

5.5.2 实验总结

本次实验相比之前的实验复杂了一些，主要是涉及到 TCP 相关的各种细节，处理要比之前麻烦一些，而且也涉及到一些值的计算。在实验中一开始怎么也找不到 TCP 连接释放时的四次握手，查阅资料后才发现原来是存在一种同时关闭的状况，并查阅资料了解了具体的情况。

通过本次实验，我实际见证了 TCP 的连接建立的具体的一步步的情况，熟悉了 TCP 和 UDP 的工作模式，对上课讲过的东西有了更深一步的认识。

实验七 DNS 解析实验

6.1 实验环境

1. 实验环境：运行 Arch Linux x86_64 操作系统的 PC 机一台
2. PacketTracer 版本：7.0.0.0202

6.2 实验目的

1. 理解 DNS 系统的工作原理。
2. 熟悉 DNS 服务器的工作过程。
3. 熟悉 DNS 报文格式。
4. 理解 DNS 缓存的作用。

6.3 实验内容及步骤

6.3.1 网络拓扑配置

打开 PacketTracer，配置网络拓扑如下图。

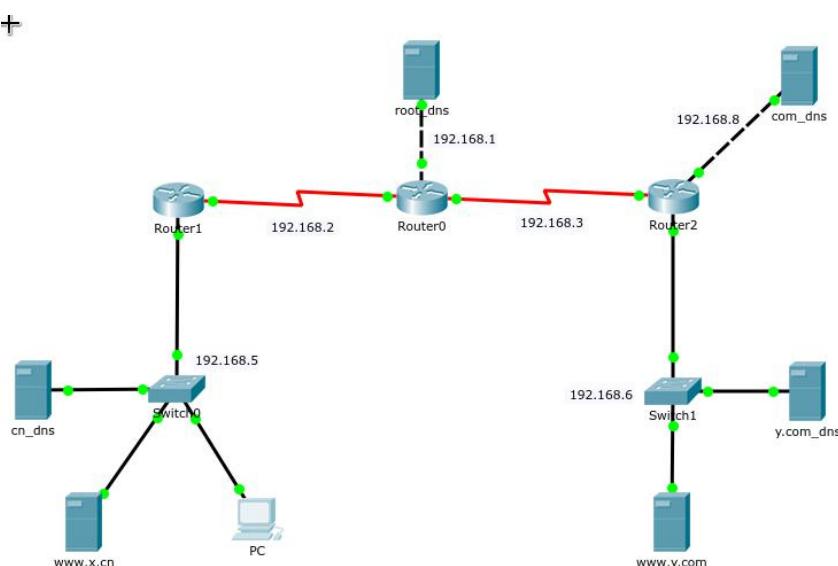


图 6.1 网络拓扑配置

其中，各台主机 IP 地址配置如下表：

表 6.1 实验拓扑中各台设备 IP 地址

设备	IP 地址
cn_dns	192.168.5.1
www.x.cn	192.168.5.2
PC	192.168.5.3
root_dns	192.168.1.1
y.com_dns	192.168.5.1
www.y.com	192.168.5.2
com_dns	192.168.8.1

接着为三台路由器配置网口 IP 值以及路由转发表，配置完成之后打开网口，可以看到所有的指示灯变成绿色，网络拓扑配置完毕。

6.3.2 观察本地域名解析过程

首先点击 Simulation，进入模拟模式。

点击 PC/Desktop，点击浏览器 Web Browser，输入域名 www.x.cn/index.html，观察 DNS 报文的发送和 DNS 解析的过程。

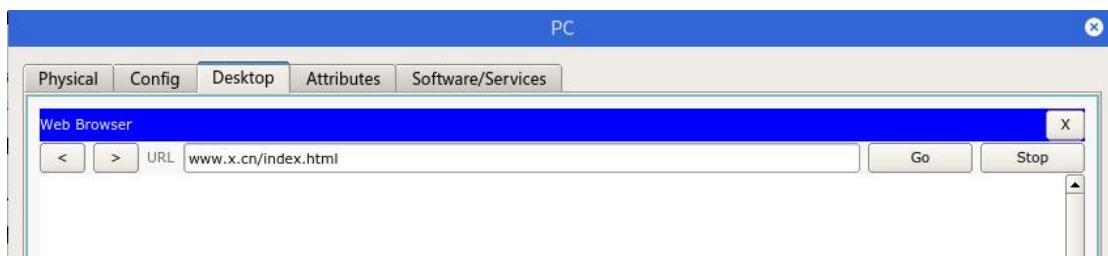


图 6.2 浏览器输入域名 www.x.cn/index.html

完成后，单击 Reset Simulation，重置模拟，将原有事件全部清空，同时关闭 PC 的 Web Browser 窗口。

6.3.3 观察外网域名解析过程

重新打开 PC 的 Web Browser，输入外网域名 www.y.com/index.html，再次观察 DNS 报文的发送和域名解析的过程。

完成后，仍然重置模拟，同时关闭 PC 的 Web Browser 窗口。

6.3.4 观察 DNS 缓存的作用

重复 3.3 步骤，在已有 DNS 缓存的情况下，观察此次外网域名解析的过程，并分析 DNS 的作用。

6.4 实验结果

6.4.1 观察本地域名解析过程

当在浏览器中输入域名之后, Event List 中出现一个 DNS 报文, 不断点击 Capture/Forward 按钮, 一步步观察报文发送的步骤, 当浏览器显示出请求的网页之后, 捕获到的 Event List 如下图:

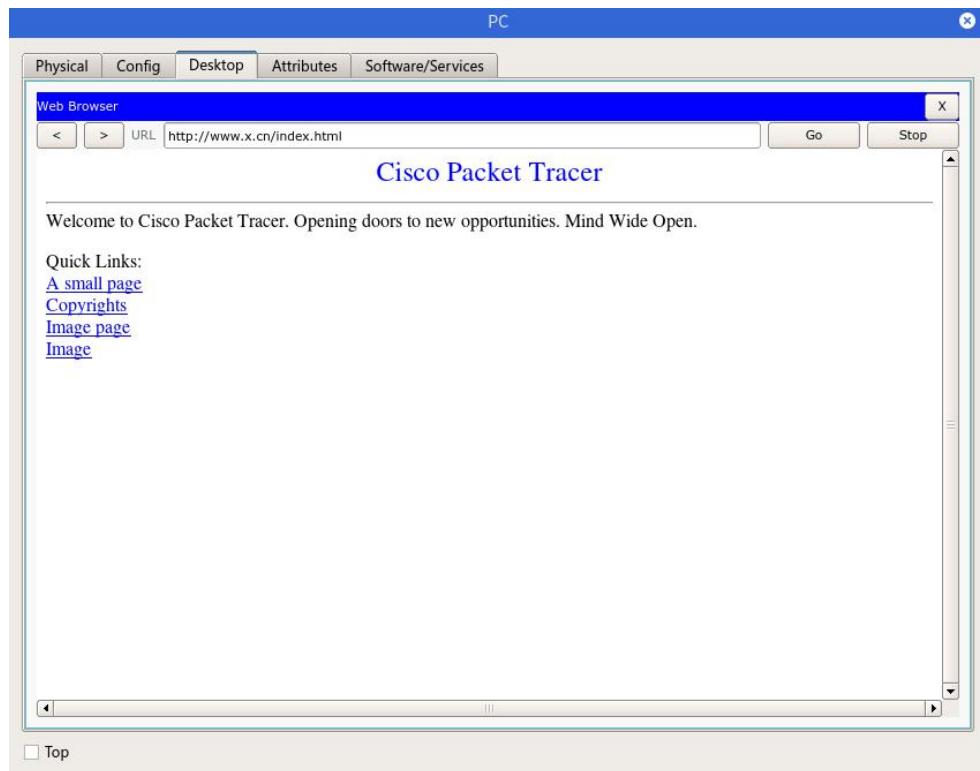


图 6.3 浏览器显示网页

Vis.	Time(sec)	Last Device	At Device	Type	Info
	0.000	--	PC	DNS	
	0.001	PC	Switch0	DNS	
	0.002	Switch0	cn_dns	DNS	
	0.003	cn_dns	Switch0	DNS	
	0.004	Switch0	PC	DNS	
	0.004	--	PC	TCP	
	0.005	PC	Switch0	TCP	
	0.006	Switch0	www.x.cn	TCP	
	0.007	www.x.cn	Switch0	TCP	
	0.008	Switch0	PC	TCP	

图 6.4 浏览器显示网页

从 Event List 中, 我们可以看到, 报文发送的过程如下: (忽略 ARP)

1. PC -> cn_dns, DNS, PC 从本地 DNS 服务器查询域名。
2. cn_dns -> PC, DNS, 本地 DNS 服务器在本地文件中找到了对应域名的

记录，返回查询的结果给 PC。

3. PC → www.x.cn, TCP, PC 已经从 DNS 服务器知道了网站服务器的 IP 地址，于是就可以和网站服务器之间建立 TCP 连接了。

DNS 查询报文和响应报文如下图：

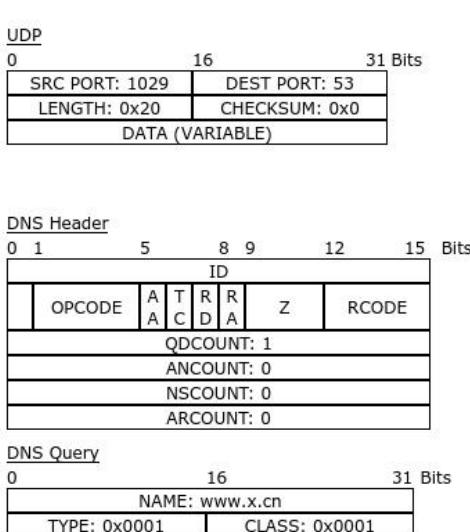


图 6.5 DNS 请求报文

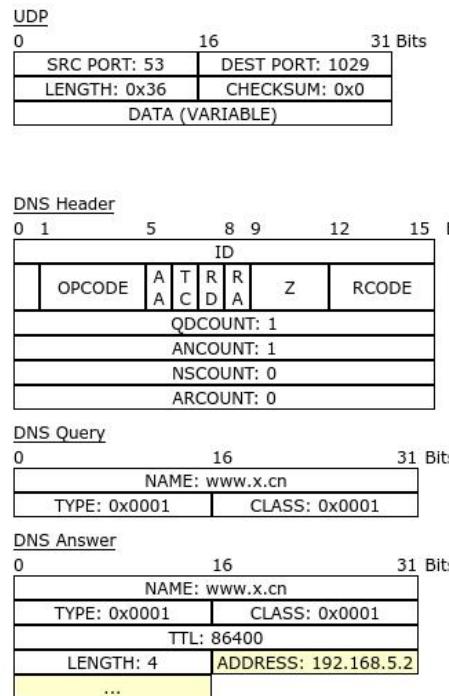


图 6.6 DNS 响应报文

DNS 响应报文中，包括请求报文的全部内容，还包括 TTL（缓存在主机上的时间），LENGTH（数据长度），ADDRESS（域名对应的 IP 地址）。

在 DNS 首部中，查询记录数 QDCOUNT 为 1，应答记录数 ANCOUNT 为 1。

6.4.2 观察外网域名解析过程

重新观察 PC 请求 www.y.com/index.html 的过程，根据捕获到的 Event List，分析 DNS 服务器之间的域名解析的过程如下：

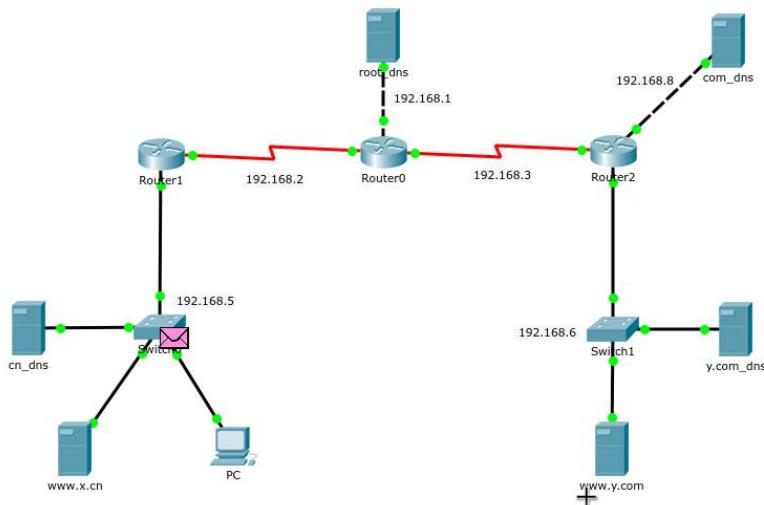


图 6.7 网络拓扑

1. PC -> cn_dns, DNS, PC 向其 DNS 服务器查询域名。
2. cn_dns -> root_dns, DNS, 本地 DNS 服务器 cn_dns 发现无法解析这个域名，在本地文件中找到根 DNS 服务器的记录，于是向根 DNS 服务器请求查询。
3. root_dns -> com_dns, DNS, 根 DNS 服务器 root_dns 收到 cn_dns 发来的 DNS 查询请求之后，在本地文件中未能直接解析出域名 www.y.com，但找到能解析.com 域名的顶级域名服务器 com_dns，于是 root_dns 向 com_dns 发送 DNS 查询。
4. com_dns -> y.com_dns, DNS, 顶级域名服务器 com_dns 收到 DNS 查询请求之后，在本地文件中未能直接解析出域名 www.y.com，但找到能解析 y.com 后缀的权威域名服务器 y.com_dns，于是 com_dns 向 y.com_dns 发送 DNS 查询。
5. y.com_dns -> com_dns, DNS, 权威域名服务器 y.com_dns 收到 DNS 查询后，在本地文件中找到了对应域名的记录，将查询结果写入应答报文中返回给 com_dns。
6. com_dns -> root_dns, DNS, com_dns 收到 DNS 应答报文之后，取出查询结果并写入应答报文中返回给 root_dns。
7. root_dns -> cn_dns, DNS, 根 DNS 服务器 root_dns 收到 DNS 应答报文之后，取出查询结果并写入应答报文中返回给 cn_dns。
8. cn_dns -> PC, DNS, 本地 DNS 服务器 cn_dns 收到 DNS 应答报文之后，取出查询结果并写入应答报文中返回给 PC。
9. PC -> www.y.com, TCP, PC 和网站服务器建立 TCP 连接。

记录各个 DNS 应答中字段的值，结果如下图：

DNS Header									
0	1	5	8	9	12	15	Bits	ID	
OPCODE		A	T	R	R	Z	RCODE		
		A	C	D	A	Z	RCODE		
		QDCOUNT: 1		ANCOUNT: 2		NSCOUNT: 0		ARCOUNT: 0	
DNS Query									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
DNS Answer									
0								NAME: y.com	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 9	NSDNAME: y.com_dns
								...	
DNS Answer									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 4	ADDRESS: 192.168.6.2
								...	
DNS Answer									
0								NAME: y.com	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 9	NSDNAME: y.com_dns
								...	

图 6.8 DNS: y.com_dns 返回

DNS Header									
0	1	5	8	9	12	15	Bits	ID	
OPCODE		A	T	R	R	Z	RCODE		
		A	C	D	A	Z	RCODE		
		QDCOUNT: 1		ANCOUNT: 2		NSCOUNT: 0		ARCOUNT: 0	
DNS Query									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
DNS Answer									
0								NAME: com	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 7	NSDNAME: com_dns
								...	
DNS Answer									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 4	ADDRESS: 192.168.6.2
								...	
DNS Answer									
0								NAME: com	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 7	NSDNAME: com_dns
								...	
DNS Answer									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 4	ADDRESS: 192.168.6.2
								...	
DNS Answer									
0								NAME: .	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 8	NSDNAME: root_dns
								...	
DNS Answer									
0								NAME: www.y.com	31 Bits
								TYPE: 0x0001	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 4	ADDRESS: 192.168.6.2
								...	
DNS Answer									
0								NAME: .	31 Bits
								TYPE: 0x0002	CLASS: 0x0001
								TTL: 86400	
								LENGTH: 8	NSDNAME: root_dns
								...	

图 6.10 DNS: root_dns 返回

图 6.11 DNS: cn_dns 返回

观察发现，不同 DNS 应答报文的首部中查询记录数（QDCOUNT）及应答记录数（ANCOUNT）不一样，第一个报文只有一个应答，后面的几个报文内容都有两个应答条目。

我们再记录下各个 DNS 服务器内的 DNS 缓存作对比，如下图：

DNS Cache Table for cn_dns				
Name	Record Type	Record Value	Time stamp	
com	NS	server: com_dns 周二 11月 6 11:39:43 2018		
www.y.com	A	IP:192.168.6.2 周二 11月 6 11:39:43 2018		

图 6.12 cn_dns 内的 DNS 缓存

DNS Cache Table for root_dns				
Name	Record Type	Record Value	Time stamp	
www.y.com	A	IP:192.168.6.2 周二 11月 6 11:39:43 2018		
y.com	NS	server: y.com_dns 周二 11月 6 11:39:43 2018		

图 6.13 root_dns 内的 DNS 缓存

DNS Cache Table for com_dns				
Name	Record Type	Record Value	Time stamp	
www.y.com	A	IP:192.168.6.2 周二 11月 6 11:39:35 2018		

图 6.14 com_dns 内的 DNS 缓存

将 DNS 缓存表和应答报文对比就发现，缓存的条目刚好就是应答报文中的几个条目。

其中 NS 类型指 Name Server，用来指定该域名由哪个 DNS 服务器来解析，A 类型值 Address，即指定主机名（或域名）对应的 IP 地址记录。

查阅资料可知，DNS 各字段的含义如下：

QR (1bit)：查询/响应标志，0 为查询，1 为响应。

opcode (4bit)：0 表示标准查询，1 表示反向查询，2 表示服务器状态请求。

AA (1bit)：表示授权回答。

TC (1bit)：表示可截断的。

RD (1bit)：表示期望递归。

RA (1bit)：表示可用递归。

Z (3bit)：用 0 填充的位。

RCODE (4bit)：表示返回码，0 表示没有差错，3 表示名字差错，2 表示服务器错误（Server Failure）。

QDCOUNT (16bit)：表示查询记录数。

ANCOUNT (16bit)：表示应答记录数。

NSCOUNT (16bit)：表示授权区域数。

ARCOUNT (16bit) : 表示附加区域数。

6.4.3 观察 DNS 缓存的作用

重复上一个步骤，在已有 DNS 缓存的情况下，请求 www.y.com/index.html，观察到如下的结果：

Simulation Panel					
Event List					
Vis.	Time(sec)	Last Device	At Device	Type	Info
0.000	--	PC	DNS	DNS	
0.001	PC	Switch0	DNS	DNS	
0.002	Switch0	cn_dns	DNS	DNS	
0.003	cn_dns	Switch0	DNS	DNS	
0.004	Switch0	PC	DNS	DNS	
0.004	--	PC	TCP	TCP	
0.005	PC	Switch0	TCP	TCP	
0.006	Switch0	Router1	TCP	TCP	
0.007	Router1	Router0	TCP	TCP	
0.008	Router0	Router2	TCP	TCP	

图 6.15 Event List

此时，PC 只向本地 DNS 服务器发送了一个 DNS 查询，就获取了网站服务器的 IP 地址，这是因为根据图 12 中的 DNS 缓存表，PC 就直接获了 www.y.com 的 IP 地址。

6.5 实验中的问题及心得

6.5.1 实验中的问题

1. 除了 PC 需要配置 DNS Server 外，Web 服务器是否需要 DNS Server 配置？如果需要，为什么？

不需要，因为 Web 服务器是一种被动接受主机连接的服务器，不需要自己去向其他的域名请求数据，因此不需要配置 DNS Server。

2. 图 7.1 中路由器之间采用串口连接，路由器为什么要采用这种连接方式。查阅资料了解串口连接的优缺点。

路由器之间采用串口连接和用以太网口连接相比保密性和可靠性较高，但速度不如以太网口连接。

3. DNS 协议使用运输层的什么协议？

UDP 协议。

4. DNS 缓存有什么用？在 Packet Tracer 中如何清空 DNS 缓存？

DNS 缓存用来存放最近解析过的域名信息，如果又需要查询同一个域名，就可以快速的返回查询结果，提高解析的效率。清空缓存时，可以进入该 DNS 服务器的 Config 窗口，点击窗口下方的 DNS Cache 按钮，在弹出的窗口中单击下方的 Clear Cache 值就即可把 DNS 缓存清空。

5. 本实验中 PC 与本地域名服务器 `cn_dns` 之间的解析是递归还是迭代？本地域名服务器 `cn_dns` 与根域名服务器 `root_dns` 之间呢？若后者用另一种解析方法，则域名服务器之间的 DNS 请求和应答的交互过程应如何？

本实验中 PC 与本地域名服务器 `cn_dns` 之间的解析的递归查询，本地域名服务器 `cn_dns` 与根域名服务之间的解析也是递归查询。

如果是另一种解析方法，则当 `root_dns` 查询到下一步该向谁查询时，会让 `cn_dns` 自己向目标进行查询。

6.5.2 实验总结

通过本次实验，我理解了 DNS 系统的工作原理，熟悉了 DNS 的工作过程以及 DNS 报文格式，同时也更加熟练了网络拓扑的 IP 配置，路由器转发表配置，把以前几次实验中还有没有搞懂的地方都解决了。

实验八 利用 C++ 开发网络应用程序

7.1 实验环境

1. 实验环境：运行 Arch Linux `x86_64` 操作系统的 PC 机一台
2. 编辑器：`Visual Studio Code`
3. 编译器：`GCC version 8.2.1`
4. 自动编译工具：`CMake version 3.12.4`

7.2 实验目的

1. 基本掌握 C++ 网络应用开发环境调试应用程序的方法。
2. 理解基于套接字开发网络应用程序的过程，深入理解 Ping 工作原理。
3. 深入理解 HTTP 协议的格式或工作过程，理解 Web 代理服务器工作过程。

7.3 实验内容及步骤

7.3.1 编写 Ping 网络程序

在计算机上执行 `ping www.baidu.com` 命令，观察标准的 ping 的输出：

```

panyue@zxcypy ~ ping www.baidu.com
PING www.a.shifen.com (61.135.169.125) 56(84) bytes of data.
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=1 ttl=55 time=43.1 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=2 ttl=55 time=48.8 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=3 ttl=55 time=47.6 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=4 ttl=55 time=36.8 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=5 ttl=55 time=26.4 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=6 ttl=55 time=26.3 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=7 ttl=55 time=28.9 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=8 ttl=55 time=27.7 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=9 ttl=55 time=26.10 ms
64 bytes from 61.135.169.125 (61.135.169.125): icmp_seq=10 ttl=55 time=46.1 ms
^C
--- www.a.shifen.com ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 22ms
rtt min/avg/max/mdev = 26.311/35.861/48.803/9.157 ms
panyue@zxcypy ~

```

图 7.1 计算机标准 ping 命令的输出

观察到，ping 命令一共有一下几个重要的输出。

- 支持域名和 IP 地址输入，其中输入域名时可以查询到 IP 地址。
- 能打印分组的 byte 大小，ttl 和用时。
- 在 ping 命令终止之后可以统计所有包的发送情况，统计时间指标。

首先，设计 ping 类如下：

代码段 7.1 ping 类设计

```

class my_ping {
    string domain;
    string ip_address;
    int my_ping_id;
    bool timeout;
    // Count packet
    int timeout_num;
    int send_num;
    int recv_num;
    // Time statistics
    double min_time;
    double avg_time;
    double max_time;
    double total_time;
    // Socket
    int sock_fd;
    struct sockaddr_in from_addr;
    struct sockaddr_in dst_addr;
    // Packet
    char send_packet[PACKET_SIZE];
}

```

```

char recv_packet[PACKET_SIZE];
// Receive time
struct timeval recv_tval;
public:
    my_ping(const char *ip, int timeout);
    unsigned short get_checksum(unsigned short *packet, int len);
    bool create_socket();
    bool close_socket();
    void send_icmp();
    void recv_icmp();
    int unpack(char *buf, int len);
    bool ping(int times);
    void statistics();
    ~my_ping();
};

}

```

下面对该类各方法做一个说明：

1. **my_ping(const char *ip, int timeout);**

构造函数，初始化 ping 类的各个值，其中最重要的是初始化域名或者 ip。

2. **unsigned short get_checksum(unsigned short *packet, int len);**

计算校验和，主要用于 ICMP 报文中的校验和字段。

3. **bool create_socket();**

创建 socket，并判断输入的是域名还是 ip 地址，若是域名，则利用 gethostbyname 函数获取对应的 ip 地址。

4. **bool close_socket();**

关闭 socket。

5. **void send_icmp();**

构建并发送 ICMP 请求报文。

6. **void recv_icmp();**

接收 ICMP 应答报文。

7. **int unpack(char *buf, int len);**

拆解 ICMP 应答报文，并从中获取需要的信息。

8. bool ping(int times);

执行 ping 操作。

9. void statistics();

统计数据，计算平均 rtt 时间。

在编写 ping 程序之前，我们首先要熟悉 ICMP 的协议结构，查阅资料知道，ICMP 报文的格式如下图所示：

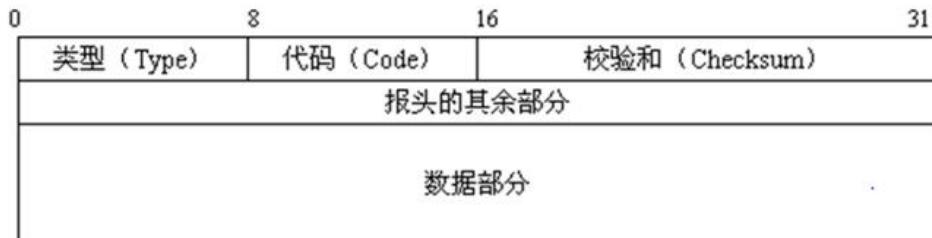


图 7.2 计算机标准 ping 命令的输出

ICMP 结构体使用头文件<netinet/ip_icmp.h>中的结构体 struct icmp。其中类型字段，要填写 ICMP_ECHO，Code 字段，要填写 0，校验和字段，在构件好除此以外的全部数据后计算，算法和 IP,TCP 等的校验和一致。

此外，还应该在 ICMP 结构体中填写上序列号和 ID，序列号从程序启动之后从 1 开始计数，ID 则利用进程号和当前的时间组合构建一个唯一确定的 ID。

最后的数据部分填上时间即可，用于计算 rtt。

而接收并拆包就轻松了许多，我们只需要将收到的 ICMP 包中的数据部分获得，解析之后就能得到发送时间，再用当前时间减去发送时间就可以得到 rtt。

本实验的源码如下：

代码段 7.2 main.c 文件

```
/*
 * HUST-IOT-Network-Labs Ping::main
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */
```

```
#include "ping.h"
```

```
int main(int argc, char **argv) {
```

```

if (argc != 2) {
    printf("Usage: ./my_ping <domain>/<ip address>\n");
    exit(0);
}
my_ping new_ping(argv[1], 100);
new_ping.ping(20);
return 0;
}

```

代码段 7.3 ping.h 文件

```

/*
 * HUST-IOT-Network-Labs Ping::headfile
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */

#ifndef ZXCPYP_PING
#define ZXCPYP_PING

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/ip_icmp.h>
#include <sys/time.h>
#include <signal.h>

#define ICMP_HEAD_LEN 8
#define ICMP_DATA_LEN 56
#define ICMP_PACKET_LEN (ICMP_HEAD_LEN + ICMP_DATA_LEN)
#define PACKET_SIZE 4096
#define PACKET_TEST_NUM 10

```

```

using namespace std;

class my_ping {
    string domain;
    string ip_address;
    int my_ping_id;
    bool timeout;
    // Count packet
    int timeout_num;
    int send_num;
    int recv_num;
    // Time statistics
    double min_time;
    double avg_time;
    double max_time;
    double total_time;
    // Socket
    int sock_fd;
    struct sockaddr_in from_addr;
    struct sockaddr_in dst_addr;
    // Packet
    char send_packet[PACKET_SIZE];
    char recv_packet[PACKET_SIZE];
    // Receive time
    struct timeval recv_tval;
public:
    my_ping(const char *ip, int timeout);
    unsigned short get_checksum(unsigned short *packet, int len);
    bool create_socket();
    bool close_socket();
    void send_icmp();
    void recv_icmp();
    int unpack(char *buf, int len);
    bool ping(int times);
    void statistics();
    ~my_ping();
};

#endif // !ZXCPYP_PING

```

```

/*
 * HUST-IOT-Network-Labs Ping::implementation
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */

#include "ping.h"

my_ping::my_ping(const char *target, int timeout) {
    domain = target;
    ip_address = "";
    my_ping_id = 0;
    timeout = false;
    timeout_num = timeout;
    send_num = 0;
    recv_num = 0;
    min_time = 0;
    avg_time = 0;
    max_time = 0;
    total_time = 0;
    sock_fd = 0;
}

unsigned short my_ping::get_checksum(unsigned short *packet, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short answer = 0;
    while (nleft > 1) {
        sum += *packet++;
        nleft -= 2;
    }
    if (nleft == 1) {
        *(unsigned char *)(&answer) = *(unsigned char *)packet;
        sum += answer;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}

```

```

bool my_ping::create_socket() {
    close_socket();

    unsigned long in_addr = 0l;
    struct hostent *hostinfo;
    struct protoent *protocol;

    if ((protocol = getprotobynumber("icmp")) == NULL) {
        printf("[ERROR] my_ping: Getprotobynumber failed!\n");
        return false;
    }

// Using raw socket (NEED ROOT!)
    if ((sock_fd = socket(AF_INET, SOCK_RAW, protocol->p_proto)) < 0) {
        printf("[ERROR] my_ping: Create socket failed!\n");
        return false;
    }

// Set sockaddr
    bzero(&dst_addr, sizeof(dst_addr));
    dst_addr.sin_family = AF_INET;

// If use domain
    if ((in_addr = inet_addr(domain.c_str())) == INADDR_NONE) {
        bool ret;
        hostinfo = gethostbyname(domain.c_str());
        if (ret) {
            printf("[ERROR] my_ping: Gethostbyname failed!\n");
            return false;
        }
        memcpy((char *)&dst_addr.sin_addr.s_addr, hostinfo->h_addr_list[0],
               hostinfo->h_length);
    }
    else
        dst_addr.sin_addr.s_addr = in_addr;

    ip_address = inet_ntoa(dst_addr.sin_addr);
    return true;
}

bool my_ping::close_socket() {

```

```

if (sock_fd != 0)
    close(sock_fd);
sock_fd = 0;
return true;
}

void my_ping::send_icmp() {
    struct icmp *icmp_packet;
    struct timeval *tval;
    while (send_num < PACKET_TEST_NUM) {
        send_num++;
        icmp_packet = (struct icmp *)send_packet;
        icmp_packet->icmp_type = ICMP_ECHO;
        icmp_packet->icmp_code = 0;
        icmp_packet->icmp_cksum = 0;
        icmp_packet->icmp_seq = send_num;
        icmp_packet->icmp_id = my_ping_id;
        // Set time as data
        tval = (struct timeval *)icmp_packet->icmp_data;
        gettimeofday(tval, NULL);
        // Set checksum
        icmp_packet->icmp_cksum =
            get_checksum((unsigned short *)icmp_packet, ICMP_PACKET_LEN);
        // Send data
        if (sendto(sock_fd, send_packet, ICMP_PACKET_LEN, 0,
                   (struct sockaddr *)&dst_addr, sizeof(dst_addr)) < 0) {
            printf("[ERROR] my_ping: Send failed!\n");
            continue;
        }
        usleep(2);
    }
}

void my_ping::recv_icmp() {
    int read_num;
    int addrlen = sizeof(struct sockaddr_in);
    struct timeval tval;
    while (recv_num < send_num) {
        read_num = recvfrom(sock_fd, recv_packet, sizeof(recv_packet), 0, (struct
        sockaddr *)&from_addr, (socklen_t *)&addrlen);
        if (read_num <= 0) {
            printf("[ERROR] my_ping: Receive failed!\n");
        }
    }
}

```

```

    return;
}
gettimeofday(&recv_tval, NULL);
if (unpack(recv_packet, read_num) == -1)
    continue;
recv_num++;
}
}

int my_ping::unpack(char *buf, int len) {
    int ip_header_len;
    struct ip *ip_packet;
    struct icmp *icmp_packet;
    struct timeval *send_tval;
    double rtt;

    ip_packet = (struct ip *)buf;
    ip_header_len = ip_packet->ip_hl * 4;
    icmp_packet = (struct icmp *)(buf + ip_header_len);
    len -= ip_header_len;
    if (len < 8) {
        printf("[ERROR] my_ping: ICMP packet length error!\n");
        return -1;
    }
    if ((icmp_packet->icmp_type == ICMP_ECHOREPLY) &&
        (icmp_packet->icmp_id == my_ping_id) &&
        (inet_ntoa(from_addr.sin_addr) == ip_address)) {
        send_tval = (struct timeval *)icmp_packet->icmp_data;
        // Get time
        if( (recv_tval.tv_usec->send_tval->tv_usec)< 0) {
            --recv_tval.tv_sec;
            recv_tval.tv_usec+=1000000;
        }
        recv_tval.tv_sec-=send_tval->tv_sec;
        // Get rtt
        rtt = recv_tval.tv_sec * 1000.0 + recv_tval.tv_usec / 1000.0;
        total_time += rtt;
        if (max_time == 0 || max_time < rtt)
            max_time = rtt;
        if (min_time == 0 || min_time > rtt)
            min_time = rtt;
        printf("%d byte from %s: icmp_seq=%d ttl=%d rtt=%f ms\n", len,

```

```

        inet_ntoa(from_addr.sin_addr), icmp_packet->icmp_seq,
        ip_packet->ip_ttl, rtt);
    }
    return 0;
}

void my_ping::statistics() {
    printf("\n----- %s ping statistics -----\\n", ip_address.c_str());
    printf("%d packets transmitted, %d received , %%%d packet loss,
time %.1fms\\n", send_num,
           recv_num, (send_num - recv_num) / send_num * 100, total_time);
    timeout_num = send_num - recv_num;
    avg_time = total_time / recv_num;
    printf("rtt min/avg/max = %.3f/%.3f/%.3f ms\\n", min_time, avg_time,
max_time);
    close(sock_fd);
}

bool my_ping::ping(int times) {
    bool ret;
    int i = 0;
    int pid;
    while (i < times) {
        ret = create_socket();
        if (!ret) {
            printf("[ERROR] my_ping: Create socket failed!\\n");
            return false;
        }
        int pid = getpid();
        pid = pid << 8;
        time_t t;
        time(&t);
        my_ping_id = (pid & 0xff00) | (t & 0xff);
        printf('PING %s(%s): %d bytes data in ICMP packets.\\n", domain.c_str(),
               ip_address.c_str(), ICMP_DATA_LEN);
        i++;
        send_num = 0;
        recv_num = 0;
        send_icmp();
        recv_icmp();
        statistics();
        if (recv_num > 0)

```

```

        break;
    }
    timeout = false;
    if (recv_num > 0) {
        timeout_num = send_num - recv_num;
        avg_time = total_time / recv_num;
    }
    else {
        timeout_num = send_num;
        avg_time = -1;
        return false;
    }
    return true;
}

my_ping::~my_ping() {
    close_socket();
}

```

7.3.2 编写 Web 代理服务器程序

要编写一个 Web 代理服务器，首先要明确 HTTP 代理的原理是什么，其应该包含以下几个模块：

1. 监听端口，接收客户端发来的请求。
2. 解析请求报文，获取目的地址和目的端口。
3. 转发报文给远端服务器，并接收远端服务器发来的响应。
4. 转发响应给客户端。

除此以外，一个完善的代理服务器还应该做到：

1. 程序鲁棒性，不能因为遇到部分可能出现的错误就终止运行。
2. 程序正确性，要保证通过代理可以使得用户正常上网。
3. 程序高效性，程序要有一定的并发度，提高网络访问速度。

据此，我们设计 `HttpProxy` 类如下：

代码段 7.5 `HttpProxy` 类设计

```

class HttpProxy {
    const string port;
public:
    HttpProxy(char *listen_port) : port(listen_port){};

```

```

void *get_in_addr(struct sockaddr *sa); // Get sockaddr, support for IPv4 and
IPv6
int Listen(const char *port); // Start listen at "port"
int Accept(int sockfd_listen); // Accept connections
bool Send(const int sockfd, const string &http_msg); // Make sure everything
is sent
int ReadHttpHeader(string &http_msg, string &address, string &dst_port); // 
Read HTTP header and get some info
int HandleRequest(const int sockfd_client); // Handle HTTP Requests
int Connect(const string host, const string dst_port); // Connect to server
bool Forward(string &host, string &dst_port, const int sockfd_client,
const string &http_msg); // Forward HTTP data between
client and server
int Start(); // Start the proxy
};

```

下面对该类各方法做一个说明：

1. `HttpProxy(char *listen_port) : port(listen_port){};`

构造函数，只初始化监听的端口即可。

2. `void *get_in_addr(struct sockaddr *sa);`

获取 sockaddr 中的地址，支持 IPv4 和 IPv6.

3. `int Listen(const char *port);`

创建 socket，并启动监听端口。

4. `int Accept(int sockfd_listen);`

接收客户端发来的 HTTP 请求。

5. `bool Send(const int sockfd, const string &http_msg);`

发送数据，应使用循环的形式确保所有数据发送完毕。

6. `int ReadHttpHeader(string &http_msg, string &address, string &dst_p
ort);`

读取 HTTP 请求头，并从中获取目的地址和目的端口，如果 HTTP 请求头
中没有写目的端口，则是使用 HTTP 的熟知端口 80。

7. `int HandleRequest(const int sockfd_client);`

处理 HTTP 请求，主要有接受客户端的数据，读取 HTTP 请求头和转发报

文。

8. int Connect(const string host, const string dst_port);

建立连接，这个方法用于在获取了客户端的请求后，根据目的地址和目的端口与 Web 服务器建立连接。

9. bool Forward(string &host, string &dst_port, const int sockfd_client, const string &http_msg);

转发报文，程序核心功能，用以实现客户端和服务器之间的数据转发。

10. int Start();

启动代理，调用前面的方法来构建代理服务器的整个过程。需要注意的是，这个函数应该实现并发操作，这里我使用了 fork 模型，在每次确认接收到合理的客户端发来的 HTTP 请求之后，就创建一个子进程，由子进程来处理这个请求，而父进程继续循环，监听客户端的连接。这样可以为程序提升很多的性能。

在编写 web 代理服务器程序之前，我们首先要熟悉 HTTP 报文头部的结构，查阅资料知道，HTTP 报文头部的格式如下图所示：



图 7.3 计算机标准 ping 命令的输出

我们读取 HTTP 头部时需要做的内容如下：

1. 检查是否有 host 字段，若没有则关闭连接，把 Connection 字段设置为 close。

2. 找到 host 字段，并获取目的地址和目的端口，注意若目的端口不存在，则填上 HTTP 周知端口 80。

3. 检查是否是 GET 请求，若是 GET 请求，则抹掉 GET 请求后面的 URL 中与 host 重复的部分。

接着，转发操作的算法如下：

1. 与服务器建立连接。
2. 把修改过 HTTP 请求头的 HTTP 报文发送给服务器。
3. 接收服务器发来的响应。
4. 将 HTTP 响应发送回客户端。

为了程序的高效，还要考虑并发性，在本程序中，每当确定建立了与客户端的一个连接后，就 fork 一个子进程，由子进程来处理这个连接，父进程继续监听端口。这样，就可以加快转发的速度，进而加快网络访问的速度。

本实验的源码如下：

代码段 7.6 proxy.h

```
/*
 * HTTP Proxy::headfile
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */

#ifndef ZXCPYP_PROXY
#define ZXCPYP_PROXY

#include <iostream>
#include <stdlib.h>
#include <cstring>
#include <string>
#include <sstream>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <thread>

#define HTTP_PORT "80"
```

```
#define BUFFER_SIZE 8192

using namespace std;

class HttpProxy {
    const string port;
public:
    HttpProxy(char *listen_port) : port(listen_port){};
    void *get_in_addr(struct sockaddr *sa); // Get sockaddr, support for IPv4 and
                                            // IPv6
    int Listen(const char *port); // Start listen at "port"
    int Accept(int sockfd_listen); // Accept connections
    bool Send(const int sockfd, const string &http_msg); // Make sure everything
    is sent
    int ReadHTTPHeader(string &http_msg, string &address, string &dst_port); // Read
    // HTTP header and get some info
    int HandleRequest(const int sockfd_client); // Handle HTTP Requests
    int Connect(const string host, const string dst_port); // Connect to server
    bool Forward(string &host, string &dst_port, const int sockfd_client,
                const string &http_msg); // Forward HTTP data between
    client and server
    int Start(); // Start the proxy
};

#endif // !ZXCPYP_PROXY
```

代码段 7.7 proxy.cpp

```
/*
 * HUST-IOT-Network-Labs HTTP Proxy::implementation
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */

#include "proxy.h"

// Get sockaddr, support for IPv4 and IPv6
void *HttpProxy::get_in_addr(struct sockaddr *sa) {
    // IPv4
    if (sa->sa_family == AF_INET)
```

```

    return &((struct sockaddr_in *)sa)->sin_addr;
// IPv6
    return &((struct sockaddr_in6 *)sa)->sin6_addr;
}

// Start listen at 'port'
int HttpProxy::Listen(const char *port) {
    int sockfd_listen;
    struct addrinfo hints, *servinfo, *addr;

// Set addrinfo
    bzero(&hints, sizeof(hints));
    hints.ai_family = AF_UNSPEC;          // IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM;      // TCP
    hints.ai_flags = AI_PASSIVE;         // IP
    hints.ai_protocol = 0;

// Get addrinfo
    int ret = getaddrinfo(NULL, port, &hints, &servinfo);
    if (ret != 0) {
        printf("[ERROR] HttpProxy: getaddrinfo for client failed!\n");
        return -1;
    }

// Loop, create and bind
    for (addr = servinfo; addr != NULL; addr = addr->ai_next) {
        sockfd_listen = socket(addr->ai_family, addr->ai_socktype,
                               addr->ai_protocol);
        if (sockfd_listen == -1) {
            printf("[ERROR] HttpProxy: socket for client create error!\n");
            continue;
        }

        int sockopt = setsockopt(sockfd_listen, SOL_SOCKET, SO_REUSEADDR,
                               &ret, sizeof(ret));
        if (sockopt == -1) {
            printf("[ERROR] HttpProxy: setsockopt failed!\n");
            exit(-1);
        }

        ret = bind(sockfd_listen, addr->ai_addr, addr->ai_addrlen);
        if (ret == -1) {
    }
}

```

```

        printf("[ERROR] HttpProxy: bind error!\n");
        close(sockfd_listen);
        continue;
    }

    // Once bind succeed, over the loop
    break;
}

// All addr create socket or bind failed
freeaddrinfo(servinfo);
if (addr == NULL) {
    printf("[ERROR] HttpProxy: bind failed!\n");
    return -1;
}

// Listen
ret = listen(sockfd_listen, 20);
if (ret == -1) {
    printf("[ERROR] HttpProxy: listen failed!\n");
    return -1;
}

cout << "[SUCCEED] HttpProxy: Listener successfully created!\n";
return sockfd_listen;
}

// Accept connections
int HttpProxy::Accept(int sockfd_listen) {
    struct sockaddr_storage from_addr;
    socklen_t sin_size = sizeof(from_addr);
    char s[INET6_ADDRSTRLEN];
    int sockfd_client;

    // Wait for connections
    cout << "[SUCCEED] HttpProxy: Waiting for connections...\n";
    sockfd_client = accept(sockfd_listen, (struct sockaddr *)&from_addr,
    &sin_size);
    if (sockfd_client == -1) {
        printf("[ERROR] HttpProxy: accept failed!\n");
        return -1;
    }
}

```

```

// Get address
inet_ntop(from_addr.ss_family, get_in_addr((struct sockaddr *)&from_addr), s,
sizeof(s));
cout << "[SUCCEED] HttpProxy: Got connection from " << s << "\n";

return sockfd_client;
}

// Make sure everything is sent
bool HttpProxy::Send(const int sockfd, const string &http_msg) {
    const char *buf = http_msg.c_str();
    int send_num;
    size_t i = 0;
    size_t size = http_msg.size();
    while (i < size) {
        send_num = send(sockfd, buf, size - i, 0);
        if (send_num == -1) {
            printf("[ERROR] HttpProxy: send failed!\n");
            return false;
        }
        i += send_num;
    }
    return true;
}

// Format HTTP request header
int HttpProxy::ReadHTTPHeader(string &http_msg, string &address, string
&dst_port) {
    // Find host
    string header = "Host:";
    size_t pos = http_msg.find(header);
    if (pos != string::npos) {
        // Find hostname
        char hostname[100];
        size_t end = http_msg.find_first_of('\r', pos);
        size_t len = http_msg.copy(hostname, end - pos - header.size() - 1, pos +
header.size() + 1);
        hostname[len] = '\0';
        // Get address
        address = string(hostname, hostname + len);
        // Find port
    }
}

```

```

header = ":";

pos = address.find(header);
if (pos != string::npos) {
    char port_str[6];
    size_t end = address.find_first_of('\r', pos);
    size_t len = address.copy(port_str, end - pos - header.size() - 1, pos +
header.size());
    port_str[len] = '\0';
    cout << "port: " << dst_port << endl;
    dst_port = string(port_str, port_str + len);
}
else {
    dst_port = HTTP_PORT;
}

// Find GET
header = "GET";
pos = http_msg.find(header);
if (pos != string::npos) {
    // Find URI
    char uri_str[1024];
    size_t end = http_msg.find_first_of('\r', pos);
    size_t len = http_msg.copy(uri_str, end - pos - header.size() - 1, pos +
header.size() + 1);
    uri_str[len] = '\0';
    string uri = string(uri_str, uri_str + len);
    size_t check = uri.find(address);
    if (check != string::npos)
        http_msg.replace(pos + header.size() + 1, check + address.size() - pos,
"");

    else {
        cout << "[INFO] No " << address << " after GET.\n";
        return -1;
    }
}
else {
    cout << "[INFO] Hostname not found!\n";
    return -1;
}

// Set connection to close
header = "Connection:";
```

```

pos = http_msg.find(header);
if (pos != string::npos) {
    size_t end = http_msg.find_first_of("\r", pos);
    http_msg.replace(pos + header.size() + 1, end - pos - header.size() - 1,
"close");
}
else {
    cout << "[INFO] Connection header not found!\n";
    return -1;
}

if (address.empty() || http_msg.size() < 8)
    return -1;

return 0;
}

// Handle HTTP Requests
int HttpProxy::HandleRequest(const int sockfd_client) {
    int recv_num;
    char buf[BUFFER_SIZE];
    string request = "";

    recv_num = recv(sockfd_client, buf, sizeof(buf), 0);
    if (recv_num == -1) {
        printf("[ERROR] HttpProxy: recv from client failed!\n");
        return -1;
    }

    cout << "[SUCCEED] Bytes received from client: " << recv_num << '\n';

    request += string(buf, buf + recv_num);
    request += '\0';

    string hostname = "";
    string dst_port = "";
    if (ReadHttpHeader(request, hostname, dst_port) == -1) {
        printf("[ERROR] HttpProxy: handle request error!\n");
        return -1;
    }
    if (!Forward(hostname, dst_port, sockfd_client, request)) {
        printf("[ERROR] HttpProxy: forward data error!\n");
    }
}

```

```

        return -1;
    }

    return 0;
}

// Connect to server
int HttpProxy::Connect(const string host, const string dst_port) {
    int ret;
    struct addrinfo hints, *servinfo, *res;
    char s[INET6_ADDRSTRLEN];
    int sockfd_server;

    bzero(&hints, sizeof(hints));
    hints.ai_family = AF_INET;           // IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM;     // TCP
    hints.ai_flags = AI_PASSIVE;         // IP
    hints.ai_protocol = 0;

// Change string to cstr
    char *chost = (char*)host.c_str();
    char *cport = (char*)dst_port.c_str();

// Get addrinfo
    ret = getaddrinfo(chost, cport, &hints, &servinfo);
    if (ret != 0) {
        printf("[ERROR] HttpProxy: getaddrinfo for server failed!\n");
        return -1;
    }

// Loop, create and bind
    for (res = servinfo; res != NULL; res = res->ai_next) {
        sockfd_server = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (sockfd_server == -1) {
            printf("[ERROR] HttpProxy: socket for server create error!\n");
            continue;
        }
        int sockopt = setsockopt(sockfd_server, SOL_SOCKET, SO_REUSEADDR,
&ret, sizeof(ret));
        if (sockopt == -1) {
            printf("[ERROR] HttpProxy: setsockopt failed!\n");
            exit(-1);
    }
}

```

```

    }

    ret = connect(sockfd_server, res->ai_addr, res->ai_addrlen);
    if (ret == -1) {
        printf("[ERROR] HttpProxy: connect to servet error!\n");
        close(sockfd_server);
        continue;
    }

    // Once bind succeed, over the loop
    break;
}

// All addr create socket or bind failed
freeaddrinfo(servinfo);
if (res == NULL) {
    printf("[ERROR] HttpProxy: connect to server failed!\n");
    return -1;
}

// Get address
inet_ntop(res->ai_family, get_in_addr((struct sockaddr *)res->ai_addr), s,
sizeof(s));
cout << "[SUCCEED] Connecting to server " << host << "\n";

return sockfd_server;
}

// Forward HTTP data between client and server
bool HttpProxy::Forward(string &host, string &dst_port, const int sockfd_client,
const string &http_msg) {
    string sever_response = "";

    // Connect to server
    int sockfd_server = Connect(host, dst_port);
    if (sockfd_server == -1) {
        printf("[ERROR] HttpProxy: connect to server error!\n");
        close(sockfd_server);
        return false;
    }

    // Send HTTP message to server
    if (Send(sockfd_server, http_msg) == false)
        return false;
}

```

```

int total_byte_num = 0;
while (true) {
    int recv_num;
    char buf[8192];

    recv_num = recv(sockfd_server, buf, sizeof(buf), 0);
    if (recv_num == -1) {
        printf("[ERROR] HttpProxy: recv from server failed!\n");
        return -1;
    }
    sever_response = string(buf, buf + recv_num);
    total_byte_num += recv_num;

    if (recv_num == 0)
        break;

    // Send HTTP message to client
    if (Send(sockfd_client, sever_response) == false)
        return false;
}

cout << "[SUCCEED] Forward data succeed! Bytes: " << total_byte_num
<< "\n";

close(sockfd_server);
close(sockfd_client);
return true;
}

// Start the proxy
int HttpProxy::Start() {
    cout << "[INFO] HTTP Proxy start at port: " << port << "\n";
    int sockfd_listen;
    sockfd_listen = Listen(port.c_str());
    if (sockfd_listen == -1)
        return -1;
    while (true) {
        int sockfd_client = Accept(sockfd_listen);
        if (sockfd_client >= 0) {
            if (!fork())

```

```

        close(sockfd_listen);
        HandleRequest(sockfd_client);
        exit(0);
    }
}
close(sockfd_client);
}
return 0;
}

```

代码段 8 main.cpp

```

/*
 * HUST-IOT-Network-Labs HTTP Proxy::main
 *
 * Created by zxcpyp
 *
 * Github: zxc479773533
 */

#include "proxy.h"

int main(int argc, char **argv) {
    int port;
    if (argc != 2) {
        printf("Usage: ./http_proxy <port>\n");
        exit(0);
    }
    port = atoi(argv[1]);
    if (port < 1024 || port > 65535) {
        printf("[ERROR] HttpProxy: Please set a port between 1024 to 65535");
        exit(0);
    }

    // Start Http Proxy
    HttpProxy proxy(argv[1]);
    proxy.Start();

    return 0;
}

```

7.4 实验结果

7.4.1 ping 程序编译运行

在目录终端下执行 `g++ main.c ping.cpp -o my_ping` 进行编译，得到可执行文件 `my_ping`，执行 `sudo ./my_ping www.baidu.com`，可以看到如下的输出：（由于这里程序的编写使用了原始套接字，因此需要 root 权限来执行程序）

```
panyue@zxcypy: ~/code/my_github/HUST-IOT-Network-Labs/ping % master • sudo ./my_ping www.baidu.com
PING www.baidu.com(61.135.169.125): 56 bytes data in ICMP packets.
64 byte from 61.135.169.125: icmp_seq=1 ttl=55 rtt=30.239 ms
64 byte from 61.135.169.125: icmp_seq=2 ttl=55 rtt=30.368 ms
64 byte from 61.135.169.125: icmp_seq=3 ttl=55 rtt=30.298 ms
64 byte from 61.135.169.125: icmp_seq=4 ttl=55 rtt=30.230 ms
64 byte from 61.135.169.125: icmp_seq=5 ttl=55 rtt=30.174 ms
64 byte from 61.135.169.125: icmp_seq=6 ttl=55 rtt=30.112 ms
64 byte from 61.135.169.125: icmp_seq=7 ttl=55 rtt=30.050 ms
64 byte from 61.135.169.125: icmp_seq=8 ttl=55 rtt=32.663 ms
64 byte from 61.135.169.125: icmp_seq=9 ttl=55 rtt=32.638 ms
64 byte from 61.135.169.125: icmp_seq=10 ttl=55 rtt=32.584 ms

----- 61.135.169.125 ping statistics -----
10 packets transmitted, 10 received, 0% packet loss, time 309.4ms
rtt min/avg/max = 30.050/30.936/32.663 ms
panyue@zxcypy: ~/code/my_github/HUST-IOT-Network-Labs/ping % master •
```

图 7.4 ping 程序运行结果

可以看到，我们的 ping 程序执行正常，并且输出了正确的运行结果，验证了程序的正确性。

和计算机中真实的 ping 程序对比发现，计算机原本 ping 程序的 rtt 一般在 2-5ms 左右，而我的 ping 程序 rtt 在 30ms 左右，推测计算机中的 ping 程序作为一个工业产品中的应用，应该有更多的优化，并且程序编写的更加高效。

7.4.2 Web 代理服务器程序编译运行

在目录终端下执行 `g++ main.cpp proxy.cpp -o http_proxy` 进行编译，可以得到可执行文件 `http_proxy`，执行 `./http_proxy 8001`，让服务器监听 8001 端口启动。可以看到如下的输出：

```
panyue@zxcypy: ~/code/my_github/HUST-IOT-Network-Labs
[INFO] HTTP Proxy start at port: 8001
[SUCCEED] HttpProxy: Listener successfully created!
[SUCCEED] HttpProxy: Waiting for connections...
```

图 7.5 web 代理服务器程序运行结果

接着配置浏览器，在 Chrome 浏览器中安装 SwitchySharp 插件，并在其中新建一个情景模式，配置 HTTP 代理如下图所示：



图 7.6 浏览器配置代理

配置完成后，在浏览器中输入 `hust.edu.cn`，观察结果：

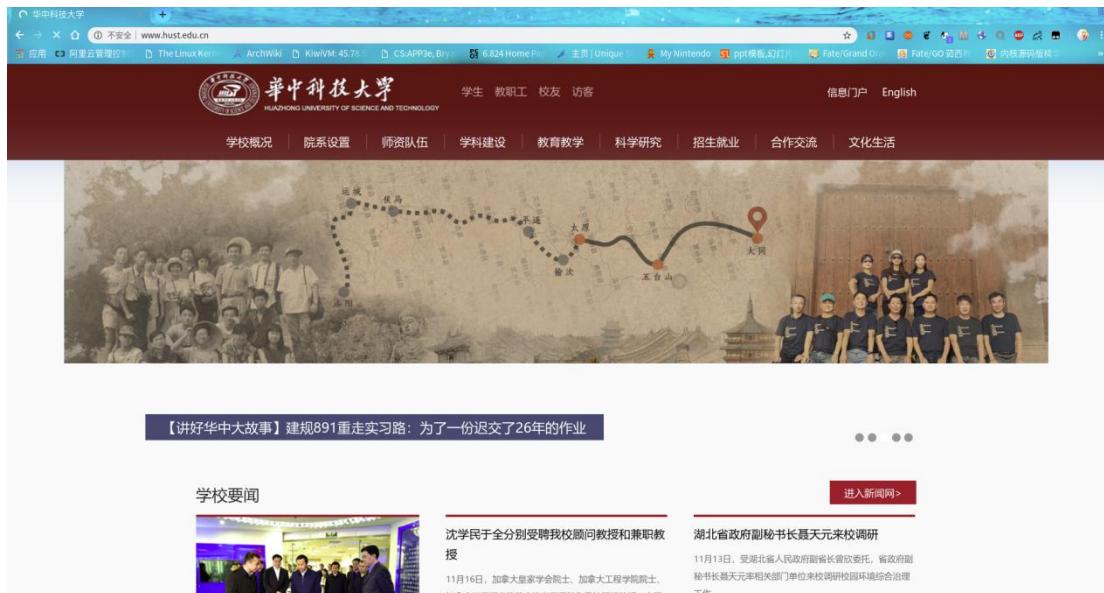
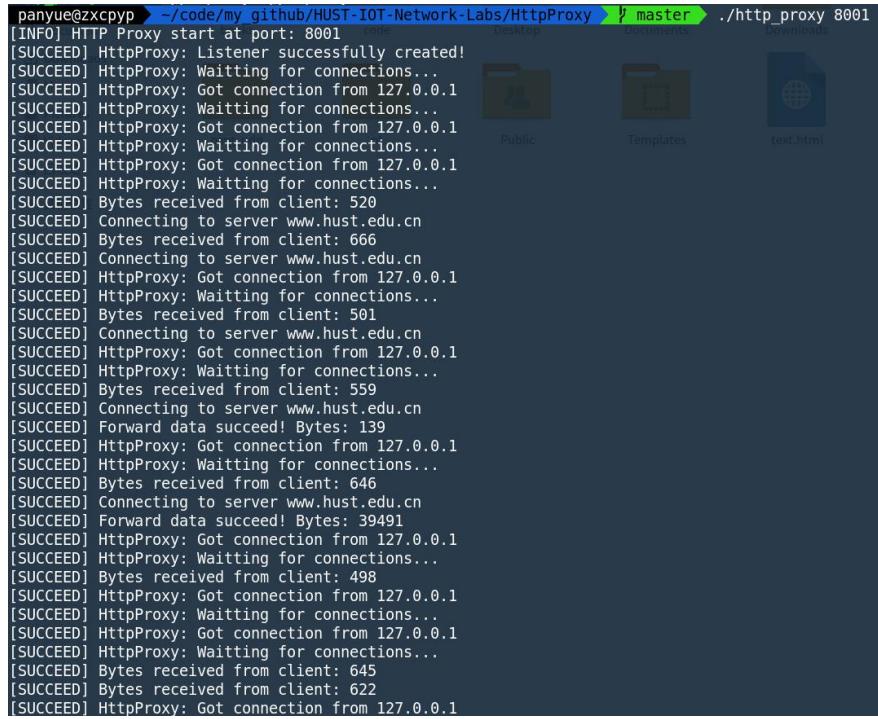


图 7.7 使用我们编写的 Web 代理访问华科官网

可以看到我们可以通过实现的代理服务器正常浏览使用 HTTP 协议的网址，并且在关闭代理服务器之后，无法正常上网，即验证我们的程序的正确性。



```

panyue@zxcypy:~/code/my_github/HUST-IOT-Network-Labs/HttpProxy$ master$ ./http_proxy 8001
[INFO] HTTP Proxy start at port: 8001
[SUCCEED] HttpProxy: Listener successfully created!
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 520
[SUCCEED] Connecting to server www.hust.edu.cn
[SUCCEED] Bytes received from client: 666
[SUCCEED] Connecting to server www.hust.edu.cn
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 501
[SUCCEED] Connecting to server www.hust.edu.cn
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 559
[SUCCEED] Connecting to server www.hust.edu.cn
[SUCCEED] Forward data succeed! Bytes: 139
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 646
[SUCCEED] Connecting to server www.hust.edu.cn
[SUCCEED] Forward data succeed! Bytes: 39491
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 498
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] HttpProxy: Got connection from 127.0.0.1
[SUCCEED] HttpProxy: Waitting for connections...
[SUCCEED] Bytes received from client: 645
[SUCCEED] Bytes received from client: 622
[SUCCEED] HttpProxy: Got connection from 127.0.0.1

```

图 7.8 web 代理服务器输出结果

同时，在代理服务器的终端输出下，可以看到数据转发的过程和转发的数据大小。并且在使用一段时间的过程中程序也是稳定的，没有发生故障错误，完全达到了实验的要求。

但是，使用我们实现的 web 代理服务器上网的时候，明显感觉到比正常的上网稍微有点慢，分析可能原因是转发的过程编写的不够完善，程序的性能有待优化。同时数据通过代理转发的开销也是一个可能的因素。

7.5 实验中的问题及心得

7.5.1 实验中的问题

本次实验由于是自己重头到尾使用 C++ 编写，所以在编程的过程中还是遇到了很多的问题的，这里列举几个遇到的主要问题。

1. 编写 Ping 程序的时候，没有注意到校验和要最后计算的问题，写好程序后查了好久才发现这个顺序错了，这一位应该在完整构建 ICMP 报文后在计算才对。

2. 编写 Web 代理服务器时，可以接收到客户端的请求，可以转发至服务器，可以收到服务器的响应，但是浏览器却没法看到网页。分析知道应该是代理和浏览器之前的通信发生了故障，分析分析源码发现在 Send 函数里 size 没有初始化，导致在第二次转发时，被系统随机赋予了一个垃圾值，导致了报文转发的失败。

3. 编写 Web 代理服务器时，一开始使用的是多线程模型，但是总因为各种

同样的问题导致程序直接崩溃。后来分析代码后意识到了存在共享变量的问题，没有考虑并发安全的问题，最后把程序改成了 fork 模型，就顺利通过测试了。

7.5.2 实验总结

本次实验是计算机网络的最后一次实验，本来实验是给了 Java 代码，让同学们体验一下程序的编写和调试。但出于学习的目的，我决定自己用 C++ 来实现这些功能。感觉整个写下来还是费了不少功夫的，不仅要熟悉课本的知识（比如 ICMP 报文结构、HTML 头部格式等），还参考了 UNIX 网络编程这本书，看了很多，了解了常见的网络编程的模型，API 的使用等等，收获不小。总之，能看到自己完全实现了一个可以用的 HTTP 代理的时候还是非常开心的。

参考文献

- 1 陈鸣著. 计算机网络: 原理与实践. 北京: 高等教育出版社, 2013.
- 2 [美] Jams F. Kurose, Keith W. Ross 著, 陈鸣译. 计算机网络 自顶向下方法(原书第 6 版). 北京: 机械工业出版社, 2016.
- 3 [美] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff 著, UNIX 网络编程 卷 I: 套接字联网 API. 北京: 人民邮电出版社, 2015.
- 4 Cisco 网络技术学院, Packet Tracer 5 使用手册, CCNA 配置手册.