

## 摘要

Traceroute 是一种被广泛应用的主动网络测量工具，是用于探测包路由、网络性能的最重要的工具之一。传统的 traceroute 功能比较单一，不能满足日益增加的网络性能分析的需求。本课题在研究现有 GNU traceroute 功能和实现方法的基础上，根据实际网络测量任务中的情况，增加了 IP 地址的物理位置信息显示、丢包率统计和探测包泊松间隔发包的功能。通过对增强型 traceroute 进行功能测试和性能测试，证明该增强型 traceroute 工具测量准确、可靠性好。

关键字：包，路径，模块，探针

## ABSTRACT

---

### ABSTRACT

Traceroute is a widely used tool for networking measuring , it is one of the most important tool for exploiting package routes and measuring the conditions of the network. The traditional traceroute lacks some features , and thus can not satisfy some pacific purposes. This research project is based on the analysis of the implementaion of the GNU traceroute and add some new features according to the practical network measuring . The new features include the display of the physical address of the IP address , statistics of package loss , make sending packags comply to poisson distribution . The pracical work proves the enchanced traceroute is useful.

Keywords : package , routes , module , probe

# 目录

第 1 章 引言 .....	1
背景和研究意义的介绍 .....	1
1.1 研究现状及发展态势 .....	1
1.2 选题依据及意义 .....	1
1.3 课题研究内容 .....	2
1.4 拟解决的关键问题 .....	2
1.5 论文特色或创新点 .....	3
第 2 章 traceroute 测量原理及其功能介绍 .....	4
2.1 Traceroute 工作原理 .....	4
2.3 实例截图 .....	6
2.4 traceroute 的重要的功能选项 .....	8
第 3 章 源代码重要功能的结构化详细分析 .....	11
3.2 描述各项性质的全局变量 .....	13
3.3 main 函数 .....	14
3.3.1 命令行接口 .....	14
3.3.2 两个出错处理函数 .....	19
3.3.4 模块调用分析 .....	19
3.4 发包收包控制模块 .....	24
3.5 各 module 内部实现 .....	27
第 4 章 各功能模块的实现 .....	33
4.1 traceroute 安全性改进 .....	33
4.2 泊松发包 .....	33
4.3 地理位置显示 .....	36
4.4 丢包率统计 .....	39
第五章 功能测试总结 .....	41
5.1 测试环境 .....	41
5.2 IP 地址物理地理信息显示 .....	41

5.3 发包间隔时间图 .....	42
5.4 包统计数 .....	43
结束语 .....	44
参考文献 .....	45
附录 .....	46
英文资料及译文 .....	50
中文译文 .....	57

## 第 1 章 引言

### 背景和研究意义的介绍

#### 1.1 研究现状及发展态势

近年来,随着 Internet 的不断发展,用户对网络的依赖程度不断增加,这无疑使网络的正常运行变得越来越重要,如何对网络中链路及路径上的性能指标及路由等进行测量一直是国内外许多研究机构研究的重点。随着主动网络测量技术研究的展开,研究人员已经开发了大量经典的测量路由、双向时延(Round-trip delay time, RTT, RTT)和链路带宽等的主动测量工具,包括 ping、traceroute、pathchar、pathchirp 等。

在以上一系列的工具有中, traceroute 是网络测量中研究网络状况的最最常用的主动测量工具之一。Traceroute 工具最早是由 Van Jacobson 在 1988 年实现。从那以后,增加了很多选项的扩充,用来支持网络管理员对网络进行探测分析。Traceroute 主要利用 IP 的 TTL 字段和 ICMP 差错报文来完成路由探测,通过计算发送 TTL 递增的数据包到目的地址与返回相应响应包之间的间隔时长,来测量端到端路径上的逐跳路由以及源节点和后继每一跳之间的 RTT。

随着 P2P 业务和新兴应用在网络上的大量展开,网络性能状况的变化越来越复杂,网络运营商以及网络服务提供商等更加迫切的需要了解网络当前性能状况,以保证网络质量以及网络业务的顺利开展。而现有的网络主动测量工具所实现的测量功能已经远远达不到实际测量应用的需求,故开发功能更加强大的网络测量工具的已迫在眉睫。

#### 1.2 选题依据及意义

传统的 traceroute 测量工具所实现的功能比较单一,且测量获取的网络信息量比较少,在网络测量的过程中给结果统计等工作造成一些不便。从严格意义上讲,测量的完整性和正确性得不到保证,性能及测量速度也不能保证最佳。首先, traceroute 是通过发送多个测试包来获得路由,而这些测量包所经过的路径有可能是不同的,在两条链路上使用负载均衡技术会导致路由抖动的发生;其次,同时发出的测量包个数,测量包的大小等条件的设置可能会增加测量中包的排队

延迟，从而干扰测量结果，测量包的响应时间，测量包之间的时间间隔等条件的设置也有可能对当前系统及网络负载产生不同的影响；最后，在 linux 环境下，tracert 工具的一些选项只有超级用户可以使用，而普通用户使用时会受到限制。这些因素都会对使用 tracert 进行主动测量的精确度和性能产生影响。

因此，本课题的主要意义是在现有 tracert 实现的基础上，提出一些可改进的地方，增强 tracert 工具原有的功能，例如：改进测量性能、加快测量速度、并增加其他测量功能、发包种类、发包间隔或统计结果类型等，通过实际的网络测试，确定并优化改进方案，实现增强型的 tracert 网络测量工具，使之更加满足目前网络测量应用中的实际需求。

### 1.3 课题研究内容

本课题的主要研究内容是基于主动测量的原理，实现功能更加强大、更符合当前网络测量需求的 tracert 测量工具。在现有 tracert 的基础上，改进测量性能，加快测量速度，并增加其他测量功能或统计结果，使之更加满足目前网络测量应用中的实际需求。

需要完成的功能主要包括测量包所经历网络中间节点的地理位置的显示功能，测量包发包模式的改进，丢包率统计功能等。另外，需要对 tracert 工具的安全性进行改进。

### 1.4 拟解决的关键问题

在研究中需要解决的关键问题包括深入了解 tracert 的功能及实现方法，了解 ICMP、TCP、UDP 等主要网络协议的原理；同时需要熟练掌握 linux 环境下的网络编程技术。同时，在增强型 tracert 网络工具的设计和实现中，务必要做到合理可行，并具有可扩展性。

本课题的实施方案如下：

研究现有 tracert 的功能及实现方法，了解 ICMP、TCP、UDP 等网络协议的主要工作原理；

通过调查现有网络测量应用的实际需求，提出现有 tracert 中需要改进的不足，并设计改进方案；

在设计方案的基础上实现增强型 tracert 网络测量工具。

## 1.5 论文特色或创新点

本课题结合主动测量技术的原理，在原有 `traceroute` 网络测量工具的基础上实现了一款功能更加强大的网络测量工具，该工具所实现的功能将更加符合当前网络测量的实际需求。

## 第 2 章 traceroute 测量原理及其功能介绍

### 2.1 Traceroute 工作原理

Traceroute 程序的设计是利用 ICMP 及 IP header 的 TTL (Time To Live) 栏位 (field)。首先, traceroute 送出一个 TTL 是 1 的 IP datagram (其实, 每次送出的为 3 个 40 字节的包, 包括源地址, 目的地址和包发出的时间标签) 到目的地, 当路径上的第一个路由器 (router) 收到这个 datagram 时, 它将 TTL 减 1。此时, TTL 变为 0 了, 所以该路由器会将此 datagram 丢掉, 并送回一个「ICMP time exceeded」消息 (包括发 IP 包的源地址, IP 包的所有内容及路由器的 IP 地址), traceroute 收到这个消息后, 便知道这个路由器存在于这个路径上, 接着 traceroute 再送出另一个 TTL 是 2 的 datagram, 发现第 2 个路由器..... traceroute 每次将送出的 datagram 的 TTL 加 1 来发现另一个路由器, 这个重复的动作一直持续到某个 datagram 抵达目的地。当 datagram 到达目的地后, 该主机并不会送回 ICMP time exceeded 消息, 因为它已是目的地了, 那么 traceroute 如何得知目的地到达了昵?

Traceroute 在送出 UDP datagrams 到目的地时, 它所选择送达的 port number 是一个一般应用程序都不会用的号码 (30000 以上), 所以当此 UDP datagram 到达目的地后该主机会送回一个「ICMP port unreachable」的消息, 而当 traceroute 收到这个消息时, 便知道目的地已经到达了。

Traceroute 提取发送 ICMP TTL 到期消息设备的 IP 地址并作域名解析。每次 Traceroute 都打印出一系列数据, 包括所经过的路由设备的域名及 IP 地址, 三个包每次来回所花时间。

Traceroute 有一个固定的时间等待响应 (ICMP TTL 到期消息)。如果这个时间过了, 它将打印出一系列的 \* 号表明: 在这个路径上, 这个设备不能在给定的时间内发出 ICMP TTL 到期消息的响应。然后, Traceroute 给 TTL 计数器加 1, 继续进行。

### 2.2 Traceroute 功能作用



Traceroute 是一个了解路由信息的重要工具。

互联网中，信息的传送是通过网中许多段的传输介质和设备（路由器，交换机，服务器，网关等等）从一端到达另一端。每一个连接在 Internet 上的设备，如主机、路由器、接入服务器等一般情况下都会有一个独立的 IP 地址。通过 Traceroute 我们可以知道信息从你的计算机到互联网另一端的主机是走的什么路径。当然每次数据包由某一同样的出发点（source）到达某一同样的目的地（destination）走的路径可能会不一样，但基本上来说大部分时候所走的路由是相同的。

UNIX, Linux 系统中，使用 Traceroute 命令；MS Windows 中为 Tracert 命令。

### Traceroute 的基本用法

```
[root@linuxserver ~]# traceroute --help
Version 1.4a12
Usage: traceroute [-dFIrnx] [-g gateway] [-i iface] [-f first_ttl]
        [-m max_ttl] [-p port] [-q nqueries] [-s src_addr] [-t tos]
        [-w waittime] [-z pausesecs] host [packetlen]
[root@linuxserver ~]#
```

#### Traceroute 对 icmp 的使用

Traceroute 是用来侦测主机到目的主机之间所经路由情况的重要工具，也是最便利的工具。前面说到，尽管 ping 工具也可以进行侦测，但是，因为 ip 头的限制，ping 不能完全的记录下所经过的路由器。所以 Traceroute 正好就填补了这个缺憾。

Traceroute 的原理是非常非常的有意思，它受到目的主机的 IP 后，首先给目的主机发送一个 TTL=1（还记得 TTL 是什么吗？）的 UDP(后面就知道 UDP 是什么了)数据包，而经过的第一个路由器收到这个数据包以后，就自动把 TTL 减 1，而 TTL 变为 0 以后，路由器就把这个包给抛弃了，并同时产生一个主机不可达的 ICMP 数据报给主机。主机收到这个数据报以后再发一个 TTL=2 的 UDP 数据报给目的主机，然后刺激第二个路由器给主机发 ICMP 数据报。如此往复直

到到达目的主机。这样，tracert 就拿到了所有的路由器 ip。从而避开了 ip 头只能记录有限路由 IP 的问题。

有人要问，我怎么知道 UDP 到没到达目的主机呢？这就涉及一个技巧的问题，TCP 和 UDP 协议有一个端口号定义，而普通的网络程序只监控少数的几个号码较小的端口，比如说 80,比如说 23,等等。而 tracert 发送的是端口号>30000(真变态)的 UDP 报，所以到达目的主机的时候，目的主机只能发送一个端口不可达的 ICMP 数据报给主机。主机接到这个报告以后就知道，主机到了

## 2.3 实例截图

以下发两张在 linux 上运行 tracert 的截图。

第一个探测到 人人网的服务器，并没有到达目的地，后面我们在实现了 IP 地址的物理位置信息显示后再回过来看这个例子，会看到人人网的服务器在哪个地方：

```

xiaoyu@chain:~/Documents/codes/computer system$ traceroute www.renren.com
traceroute to www.renren.com (60.29.242.220), 30 hops max, 60 byte packets
 1 121.48.153.129 (121.48.153.129)  2.811 ms  2.902 ms  3.115 ms
 2 125.71.230.29 (125.71.230.29)  3.525 ms  3.796 ms  3.987 ms
 3 125.71.230.1 (125.71.230.1)  4.219 ms  4.363 ms  4.557 ms
 4 118.121.22.121 (118.121.22.121)  4.681 ms  4.861 ms  5.035 ms
 5 221.237.185.37 (221.237.185.37)  5.251 ms  221.237.185.21 (221.237.185.21)  5.422 ms  221.237.185.37 (221.237.185.37)  5.587 ms
 6 222.213.14.165 (222.213.14.165)  5.799 ms  2.120 ms  3.372 ms
 7 202.97.47.197 (202.97.47.197)  13.588 ms  14.195 ms  14.482 ms
 8 218.30.19.222 (218.30.19.222)  15.160 ms  15.377 ms  15.612 ms
 9 ***
10 *** double send times: 2, double is the return type of get times
11 *** double recv times
12 *** not recv ttl
13 *** not sk
14 *** not seq
15 *** not text
16 *** not err str len: 0, not enough
17 ***
18 *** struct probe struct probe
19 ***
20 ***
21 *** to module struct: 1, 这个 struct 代表一个探针
22 *** struct to module struct next
23 *** struct probe name
24 *** not init: 1, not sock addr, not dest
25 *** not init: 1, not poll seq, not packet len: 1, 数据包长度不够
26 *** not send probe: probe ip: not ttl
27 *** not recv probe: not id, not events
28 *** not create probe: probe ip
29 *** not option options: 1, not module options, if any
30 *** not option options: 1, not module options, if any
xiaoyu@chain:~/Documents/codes/computer system$

```

Figure 2.1 未增强 traceroute 到人人网服务器

第二个是探测 GOOGLE 服务器，这次到达了目的地，后面我们也会回来再次用这例子。

```
xiaoyu@chain:~/Documents/codes/computer system$ traceroute www.google.com
traceroute to www.google.com (74.125.71.104), 30 hops max, 60 byte packets
 1  121.48.153.129 (121.48.153.129)  3.136 ms  3.317 ms  3.540 ms
 2  125.71.230.61 (125.71.230.61)  3.666 ms  4.098 ms  4.291 ms
 3  125.71.230.5 (125.71.230.5)  4.503 ms  4.237 ms  4.424 ms
 4  118.121.22.121 (118.121.22.121)  12.871 ms  13.037 ms  13.230 ms
 5  221.237.185.37 (221.237.185.37)  4.461 ms  4.566 ms  221.237.185.21 (221.237.185.21)  4.685 ms
 6  222.213.9.69 (222.213.9.69)  5.293 ms  2.204 ms  222.213.14.153 (222.213.14.153)  2.991 ms
 7  202.97.66.33 (202.97.66.33)  41.893 ms  42.137 ms  42.553 ms
 8  202.97.53.42 (202.97.53.42)  42.244 ms  42.657 ms  42.831 ms
 9  * * *
10  202.97.53.173 (202.97.53.173)  84.917 ms  85.053 ms *
11  202.97.61.102 (202.97.61.102)  198.994 ms  199.203 ms  199.327 ms
12  202.97.62.214 (202.97.62.214)  212.455 ms  207.524 ms  208.015 ms
13  209.85.241.56 (209.85.241.56)  81.223 ms  209.85.241.58 (209.85.241.58)  55.009 ms  64.778 ms
14  216.239.43.17 (216.239.43.17)  46.163 ms  209.85.253.69 (209.85.253.69)  41.835 ms  42.037 ms
15  * 216.239.48.226 (216.239.48.226)  228.391 ms  228.521 ms
16  hx-in-f104.1e100.net (74.125.71.104)  61.550 ms  61.344 ms  62.231 ms
xiaoyu@chain:~/Documents/codes/computer system$
```

Figure 2.2

## 2.4 traceroute 的重要的功能选项

--help 打印帮助信息，并退出。

-4, -6 显示地指定使用 IPv4 或 IPv6 traceroute。默认情况下，traceroute 会解析给定的主机名，并自动选择合适的协议。如果解析主机名既得到了 IPv4 的地址，又得到了 IPv6 的地址，traceroute 会使用 IPv4。

-I 使用 ICMP ECHO 进行探测。

-T 使用 TCP SYN 进行探测。

-U 使用 UDP 报文进行探测（默认情况）。对于无特权用户来说，只允许使用 UDP 报文进行探测。

-d 允许进行 socket 级别的调试（当 Linux kernel 支持它的时候）Enable socket level debugging (when the Linux kernel supports it)

-F 将“不要分段 Don't Fragment”位置位。这将告诉中间路由器不要将该包分段（当路由器发现该探测包对于网络中 MTU 来说太大的时候）

-f <first\_ttl> 设置第一个检测数据包的存活数值 TTL 的大小。默认是 1。

-g <gateway>

告诉 traceroute 为发出的 packet 增加 IP 源路由选项，以此告诉网络在路由该 packet 时需要通过指定的网关。不是十分有用，大多数的路由器因为安全方面的考虑将源路由设置为失效。

**-i <interface>**

指定 traceroute 发送包时经过的端口。默认的端口是依照路由表选定的。

**-m <max\_ttl>**

指定 traceroute 将要探测的最大跳数（最大的生存时间）。默认值为 30。

**-N <nqueries>**

指定同时发送的探测包数目。同时发送几个探测包可以适当地加快 traceroute 的速度。默认值为 15。注意：有些路由器和主机会使用 ICMP 速率限制，在这种情况下，指定同时发送大量的探测包会导致一些响应丢失。

**-n** 显示的时候无需将 IP 地址和主机名相对应。直接使用 IP 地址而非主机名称。

**-p <port>**

使用 UDP 的跟踪，基础的 traceroute 会使用指定的目的端口（每个探测包的目的端口号会递增）。

使用 ICMP 跟踪，指定初始的 icmp 序列号（每个探测包递增）。

使用 TCP 跟踪，指定要连接的端口号（常数）。

**-t <tos>**

对于 IPv4，设置服务类型（Type of Service, TOS）及优先值。有用的数值有 16（低延迟）和 8（高吞吐量）。注意在使用某些 TOS 优先值时，你必须是超级用户。

对于 IPv6，设置流量控制值。

**-w <waittime>**

设置对探测包响应的等待时间（秒），默认值是 5 秒。

**-q <nqueries>**

设置每一跳的探测包数量。默认是 3。

**-r** 忽略普通的路由表，直接发送到所在网络（attacked network）的远端主机上。如果该主机不是直接附在网络（directly-attached network）中，会返回一个错误。该选项可用于 ping 一个本地主机，而该主机所经过的端口没有路由。

**-s <source\_addr>**

设置本地主机发出数据包的地址。注意你必须选择某一端口的地址，这个地址就是发出数据包的端口所使用的。

**-z <sendwait>**

探测包之间最小的时间间隔（默认值为 0）。如果该值大于 10，则它指定的为毫秒，否则，它指定的为秒（允许使用浮点数）。当某些路由器对 ICMP 报文实行速率限制时有用

后面我们在分析源代码的时候会看到这些选项是如何实现的。

## 第 3 章 源代码重要功能的结构化详细分析

### 3.1 源代码的文件结构

准备工作。

在 GNU 的官网上 [www.gnu.org](http://www.gnu.org) 下载开源的 GNU traceroute, 我使用的是 traceroute-2.0.16, 压缩包的大小为 63KB, 解压之后的大小为 216.1KB, 以下是解压之后的目录文件。

外围的文件中有两个 shell script 文件, `chvers.sh`, `store.sh`, 这两件文会被 Makefile 所调用, 另外 `Make.defines`, `Make.rules`, `default.rules`, `traceroute.spec` 都是和 Makefile 有关的, `COPYING`, `COPYING.LIB` 介绍了 GPL, GUN General Public License (GNU 通用公共许可证) CREDITS 中作者对相关的人表示了感谢, README 中简单介绍了此 traceroute 的情况, 提到了这是为 linux 实现的最新的 traceroute, 以及其实一些主要特性, 当然 traceroute 完整的功能描述可以在其 man page 中看到。

特别要提到 TODO 文件, 软件的作者在里面提到了他对此程序的一些改进完善的想法, 因为这是开源的 GNU traceroute, 我们都可以自己动手去实现这些想法和我们自己的想法。

我们可以看到, 最主要的代码都在 traceroute 这个目录下, 我们要深入剖析的也是这个目录下的代码。

```
-- ChangeLog
-- chvers.sh
-- COPYING
-- COPYING.LIB
-- CREDITS
-- default.rules
-- include
--   |-- version.h
-- libsupp
--   |-- clif.c
--   |-- clif.h
-- Make.defines
-- Makefile
-- Make.rules
-- README
-- store.sh
-- TODO
-- traceroute
--   |-- as_lookups.c
--   |-- csum.c
--   |-- extension.c
--   |-- flowlabel.h
--   |-- mod-icmp.c
--   |-- mod-raw.c
--   |-- mod-tcp.c
--   |-- mod-tcpconn.c
--   |-- mod-udp.c
--   |-- module.c
--   |-- poll.c
--   |-- random.c
--   |-- time.c
--   |-- traceroute.8
--   |-- traceroute.c
--   |-- traceroute.h
-- traceroute.spec
-- VERSION
-- wrappers
--   |-- lft
--   |-- Makefile
--   |-- README.wrappers
--   |-- tcptraceroute
--   |-- tracepath
--   |-- traceproto
--   |-- traceroute-nanog
```

4 directories, 40 files



```

55 #define MAX_HOPS      255 //最大跳数
56 #define MAX_PROBES    10 //每一跳最大发包数
57 #define MAX_GATEWAYS_4 8
58 #define MAX_GATEWAYS_6 127
59 #define DEF_HOPS      30
60 #define DEF_SIM_PROBES 16 /* including several hops */
61 #define DEF_NUM_PROBES 3
62 #define DEF_WAIT_SECS 5.0
63 #define DEF_SEND_SECS 0
64 #define DEF_DATA_LEN  40 /* all but IP header... */
65 #define MAX_PACKET_LEN 65000
66 #ifndef DEF_AF
67 #define DEF_AF        AF_INET
68 #endif
69
70 #define ttl2hops(X)    (((X) <= 64 ? 65 : ((X) <= 128 ? 129 : 256)) - (X))
71

```

Figure 3.1 源代码树

## 3.2 描述各项性质的全局变量

main 函数定义在 traceroute.c 中，traceroute.c 同时也定义了所有模块都要使用的公共函数，和相关的变量

首先我们列出这个文件定义的宏

Figure 3.2 宏定义

全局变量：

这些全局变量很重要，描述了程序和包的各个方面的性质，现在不必弄清楚这些变量是描述什么用的，在后面的函数分析过程中我们会反复提到这些变量，值得注意的是这些全局变量都定义为 static,只在这一个文件中使用，non-static 全局变量应该在头文件中定义。

```

92 static int dontfrag = 0;
93 static int noresolve = 0;
94 static int extension = 0;
95 static int as_lookups = 0;
96 static unsigned int dst_port_seq = 0;
97 static unsigned int tos = 0;
98 static unsigned int flow_label = 0;
99 static int noroute = 0;
100 static unsigned int fwmark = 0;
101 static int packet_len = -1;
102 static double wait_secs = DEF_WAIT_SECS; //the time (in seconds) to wait for a response to a probe (default 5.0 sec)
103 static double send_secs = DEF_SEND_SECS; //send_secs is the time interval of sending probes
104 static int mtudisc = 0;
105 static int backward = 0;
106
107 static struct sockaddr_any dst_addr = {{ 0, }, }; // 这个文件定义的自己的(static) dest 和 source 地址
108 static char *dst_name = NULL; // 应该是目标 域名
109 static char *device = NULL;
110 static struct sockaddr_any src_addr = {{ 0, }, };
111 static unsigned int src_port = 0;
112
113 static const char *module = "default"; // first used in check_progname() 应该指的是调用哪个相应的模块来发包
114 static const struct tr_module *ops = NULL; // 在 do_it 中用到, ops 应该就是表示 我们在命令中要使用的那个 module
115
116 static char *opts[16] = { NULL, }; /* assume enough */ // 这个是用来当指定 -0 时, 存放对模块里面参数的设置
117 static unsigned int opts_idx = 1; /* first one reserved... */
118
119
120 static int af = 0;

```

```

84 static int num_gateways = 0;
85 static unsigned char *rtbuf = NULL;
86 static size_t rtbuf_len = 0;
87 static unsigned int ipv6_rthdr_type = 2; /* IPV6_RTHDR_TYPE_2 */
88
89 static size_t header_len = 0;
90 static size_t data_len = 0;
91
92 static int dontfrag = 0;
93 static int noresolve = 0;
94 static int extension = 0;
95 static int as_lookups = 0;
96 static unsigned int dst_port_seq = 0;
97 static unsigned int tos = 0;
98 static unsigned int flow_label = 0;
99 static int noroute = 0;
100 static unsigned int fwmark = 0;
101 static int packet_len = -1;
102 static double wait_secs = DEF_WAIT_SECS; //the time (in seconds) to wait for a response to a probe (default 5.0 se
103 static double send_secs = DEF_SEND_SECS; //send_secs is the time interval of sending probes
104 static int mtudisc = 0;
105 static int backward = 0;
106

```

Figure 3.5 全局变量定义 2

### 3.3 main 函数

#### 3.3.1 命令行接口

我们从 main 函数开始，首先要看的是命令行的分析

命令解析模块的主要功能是根据用户输入的命令行判断是否符合格式，并根据输入提取有效的信息，并进行设置。

命令解析模块的重点在于提取有效信息，由我们命令行的格式可知道最后两个参数分别为目的站点地址和服务器的 IP 地址，为得到这两个参数我们可以简单的利用 main()函数的传递的 argc 和 argv[7]参数获得。即目的站点的地址为 argv[argc-2]，服务器 IP 地址为 argv[argv-1]。

命令解析模块的流程图如下：

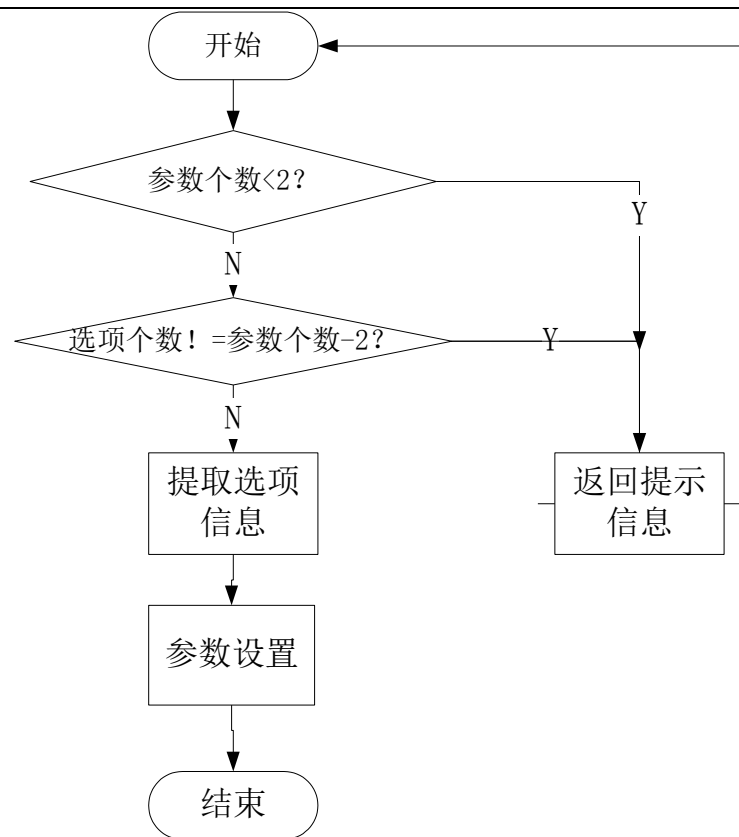


Figure 3.6 命令解析模块流程图

```

558 int main (int argc, char *argv[]) {
559     默认  setlocale (LC_ALL, "");
560     setlocale (LC_NUMERIC, "C"); /* avoid commas in msec printed */
561
562     check_progname (argv[0]);
563
564     if (CLIF_parse (argc, argv, option_list, arg_list,
565                   CLIF_MAY_JOIN_ARG | CLIF_HELP_EMPTY) < 0
566         ) exit (2);
567
568     ops = tr_get_module (module);
569     if (!ops) ex_error ("Unknown traceroute module %s", module);
570
571     if (!ops->user && geteuid () != 0)
572         ex_error ("The specified type of tracerouting "
573                 "is allowed for superuser only");
574
575
576
577

```

Figure 3.7 命令行解析代码

main 函数开始几行定义如上， 第一二行是设置相关语言的  
check\_prognam 是这个在这个文件中自定义的一个函数

```

149 /* Set initial parameters according to how we was called */
150
151 static void check_prognam (const char *name) {
152     const char *p;
153     int l;
154
155     p = strrchr (name, '/');
156     if (p) p++;
157     else p = name;
158
159     l = strlen (p);
160     if (l <= 0) return;
161     l--;
162
163     if (p[l] == '6') af = AF_INET6;
164     else if (p[l] == '4') af = AF_INET;
165
166     if (!strncmp (p, "tcp", 3))
167         module = "tcp";
168     if (!strncmp (p, "tracert", 7))
169         module = "icmp";
170
171     return;
172 }

```

Figure 3.8 解析程序调用名代码

check\_prognam(argv[0])对程序的调用名进行了检查，里面涉及到两个变量的设置，

af, module, 根据程序调用名 argv[0]对这两个变量进行了设置

从 ..图我们可以看到 module 定义 static const char \* module = “default”  
module 是一个字符串，用它来代表发包所用的模块

例如若名字为 traceroute4 则用 ipv4 发包，若名字为 tcptraceroute 则用 tcp

模块发包，这时 `module = "tcp"`，在我自己的电脑上，ubuntu 系统上，不同的名字像 `tcptraceroute` 都是一个 link file,指向唯一的 `traceroute`,然后再在程序里面来判断怎么发包

下面我们看 566-568 行，

```
566         if (CLIF_parse (argc, argv, option_list, arg_list,
567                         CLIF_MAY_JOIN_ARG |
CLIF_HELP_EMPTY) < 0
568             ) exit (2);
```

这个函数是在 `/usr/include/clif.h` 头文件中声明的，（应该并不是属于某个标准的接口函数，能找到的资料很少），这个文件并不是 GNU C library 的一部份在 `clif.h` 中定义了两个 struct

```
13 typedef struct CLIF_option_struct CLIF_option;
14 struct CLIF_option_struct {
15     const char *short_opt;
16     const char *long_opt;
17     const char *arg_name;
18     const char *help_string;
19     int (*function) (CLIF_option *optn, char *arg);
20     void *data;
21     int (*function_plus) (CLIF_option *optn, char *arg);
22     unsigned int flags;
23 };
24 #define CLIF_END_OPTION { 0, 0, 0, 0, 0, 0, 0, 0 }
25
26 typedef struct CLIF_argument_struct CLIF_argument;
27 struct CLIF_argument_struct {
28     const char *name;
29     const char *help_string;
30     int (*function) (CLIF_argument *argm, char *arg, int index);
31     void *data;
32     unsigned int flags;
33 };
34 #define CLIF_END_ARGUMENT { 0, 0, 0, 0, 0 }
```

Figure 3.9 用于命令行解析的数据结构

在 `traceroute.c` 中定义了一个 `struct CLIF_option` 的数组，对每个命令行选项可能出现的参数都有了说明，及相关的函数调用和变量设置。

命令解析模块的异常处理：由于探测时我们必须知道目的站点和服务器的 IP 地址，所有参数的最小个数是 2，如果参数个数小于 2，那么输入命令就是无效的命令。当有选项存在是，参数个数-选项个数应当为 2，这样才是有效的输入格式。如果用户输入的是无效的信息，那么命令解析模块会提示输入错误，返回给用户一个正确的模式，让用户重新输入。

举一个例子 在 shell 中敲下如下命令调用时

```
traceroute -f 3 www.renren.com
```

-f 表示 `first_ttl` 设为 3,即从路径上的第三个路由开始记录，程序中相关的代码为

```
450      { "f", "first", "first_ttl", "Start from the %s hop (instead from 1)",  
451          CLIF_set_uint, &first_hop, 0, 0 },
```

可以看到，若用了 -f，则会调用 `CLIF_set_uint` 把 `first_hop` 的值设为我们 -f 后指定的值

当然调用的函数也可以是我们自己定义的函数

比如

```
452      { "g", "gateway", "gate", "Route packets through the specified gateway "  
453          "(maximum " _TEXT(MAX_GATEWAYS_4) "  
for IPv4 and "  
454          _TEXT(MAX_GATEWAYS_6) " for IPv6)",  
455          add_gateway, 0, 0, CLIF_SEVERAL },
```

-g 设用来设置我们的包所必须经过的路由，`add_gateway` 就是我们自己定义的函数，这个在后面我们会说到

### 3.3.2 两个出错处理函数

下面我们看 traceroute.c 中定义的两个出错处理函数，程序中到处都在用到

```
131 static void ex_error (const char *format, ...) {
132     va_list ap;
133
134     va_start (ap, format);
135     vfprintf (stderr, format, ap);
136     va_end (ap);
137
138     fprintf (stderr, "\n");
139
140     exit (2);
141 }
142
143 void error (const char *str) {
144
145     fprintf (stderr, "\n");
146
147     perror (str);
148
149     exit (1);
150 }
```

### 3.3.4 模块调用分析

```

592 ops = tr_get_module (module); // defined in module.c
593 if (!ops) ex_error ("Unknown traceroute module %s", module);
594
595 if (!ops->user && geteuid () != 0)
596     ex_error ("The specified type of tracerouting "
597               "is allowed for superuser only");
598

```

接下来的几行代码

Figure 3.10 获取使用模块代码

`ops = tr_get_module (module)`，是非常关键的，用全局变量的定义中 我们可以看到 114 `static const tr_module *ops = NULL;`；  
`ops` 是一个 `struct tr_module` 型的变量，`tr_module` 是在 `traceroute.h` 中定义的一个 `struct`，是整个程序中最重要两个 `struct` 之一（另一个是描述 发包探针的，后面我们会提到）

下面介绍一下这个 `struct`，`tr_module` 是用来描述每个发包模块的，每次我们调用 `traceroute`，都会决定用哪一个发包模块来发包，是发 UDP 包（默认），TCP 包（-T），还是 ICMP 包（-I）

`tr_module` 第一个成员变量是指向另一个 `tr_module` 的 `pointer`，这形成了链表，每一个我们程序所实现的 `module` 在这个链表中都会有一个 `struct` 与之对应，而这个链表在 `main` 函数调用之前就已经形成了，下面我们会说这是怎么做到的。

```

36 struct tr_module_struct {
37     struct tr_module_struct *next;
38     const char *name;
39     int (*init) (const sockaddr_any *dest,
40                 unsigned int port_seq, size_t *packet_len); //这里定义了4个函数指针
41     void (*send_probe) (probe *pb, int ttl);
42     void (*recv_probe) (int fd, int revents);
43     void (*expire_probe) (probe *pb);
44     CLIF_option *options; /* per module options, if any */
45     short user; /* whether applicable for non-root users */
46     short one_per_time; /* no simultaneous probes */
47     size_t header_len; /* additional header length (aka for udp) */
48 };
49 typedef struct tr_module_struct tr_module; //
50

```

Figure 3.11

`tr_module` 别外的成员有 `const char * name`，这是 `module` 的名字，也就是



traceroute.c 中全局变量 `module` 所对应的值，在我们的程序中，程序定义了 6 个 `module struct`, `name` 分别为 `default`, `udp`, `udplite`, `tcp`, `tcpconn`, `icmp`, `raw` 分别在 `mod-udp.c`, `mod-tcp.c`, `mod-tcpconn.c`, `mod-icmp.c` 中

45 行有一个成员 `short user;` 表示是否需要 root 权限来运行这个 `module`,

上面我们提到的 6 个 `module` 中，前三个的 `user` 为 1, 不需要 root 权限，后三个 `user` 为 0, 需要 root 权限

`tr_module` 还定义的 4 个函数指针, `init`, `send_probe`, `recv_probe`, `expire_probe` 分别指完成相应功能的函数，这些函数在定义相应 `tr_module` 的文件中定义，这些函数完成初始化，发包，收包，超时处理，非常重要，后面我们会看到

`short one_per_time` 的值都为 1, 表示每个 `probe` 所发的包个数

`header_len` ...??

`CLIF_option` 是每个 `module` 的 `option`, (后面对 `tcp` 举例)

其实在 `main` 函数调用之前，`module.c` 中的，`tr_register_module` 就已经被调用过好几次了，这个函数是把每个 `mod-*.c` 中定义的各自的 `tr_module` 放到一个链表中，这个链表的表末是 `NULL` (`module.c`) 中定义，下面是摘自 `module.c` 中的代码

```
16 static tr_module *base = NULL;
17
18 void tr_register_module (tr_module *ops) {
19
20     ops->next = base;
21     base = ops;
22 }
```

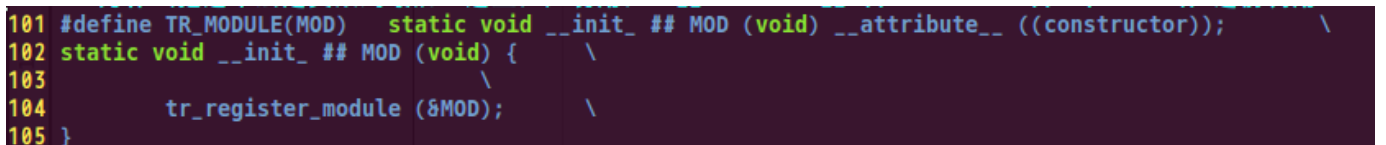
而这个函数是怎么定义的呢，下面我们以 `default` 模块举例，`default module` 是我

们 traceroute 默认使用的 module,  
定义如下, 代码摘自 mod-udp.c

```
203 static tr_module default_ops = {
204     .name = "default",
205     .init = udp_default_init,
206     .send_probe = udp_send_probe,
207     .recv_probe = udp_recv_probe,
208     .expire_probe = udp_expire_probe,
209     .user = 1,
210     .header_len = sizeof (struct udphdr),
211 };
212
213 TR_MODULE (default_ops);
```

我们可以看到, default\_ops 是一个 static 全局变量, 且其大部分成员已经初使化 (struct 以如此方式初使化部分成员时名字前面加一个点.)  
关键在于 宏 TR\_MODULE (default\_ops)

TR\_MODULE 这个宏是在 traceroute.h 的最后定义的(由于直接复制过来的代码不好看, 这里用截图)



```
101 #define TR_MODULE(MOD)    static void __init_## MOD (void) __attribute__((constructor)); \
102 static void __init_## MOD (void) { \
103     \
104     tr_register_module (&MOD); \
105 }
```

Figure 3.12

我们可以看到, 宏 TR\_MODULE 展开后是一个函数声明和一个函数定义, 以 TR\_MODULE (default\_ops)举例:

我们用命令 `gcc -E -o mod-udp.i mod-udp.c`  
为 mod-udp.c 生成预处理后的文件, 来查看宏展开后的样子:

```

3578
3579 static void __init_default_ops (void) __attribute__((constructor));
3580 static void __init_default_ops (void) { tr_register_module (&default_ops); };
3581

```

Figure 3.13

我们可以清晰的看到生成了一个函数声明和一个函数定义，这个函数声明后后面加了 `__attribute__((constructor))`，表示这个函数在 `main` 函数调用之前就要调用。而在每个 `tr_module` 定义后面都有这样一个相应的宏调用，用 `tr_register_module` 把所有定义的 `tr_module` 组成一个链表，这一切都在 `main` 函数调用之前发生。

`ops = tr_get_module(module)` 就是通过下面在 `module.c` 中定义的函数来得到我们此次调用程序将要使用的模块。

```

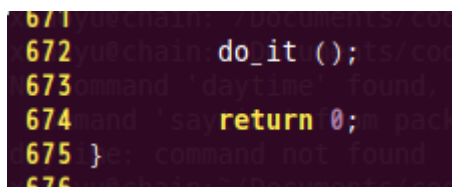
24 const tr_module *tr_get_module (const char *name) {
25     const tr_module *ops;
27     if (!name) return 0;
28
29     for (ops = base; ops; ops = ops->next) {
30         if (!strcasecmp (name, ops->name))// compare ingoring letter case
31             return ops;
32     }
33
34     return NULL;
35 }

```

（这里试着通过 `mod-udp.o` 来显示在 `main` 调用之前调用以上函数）

准备工作做好了之后就要开始发包了，所有的发包收包工作都下面的这个函数中。

完成，这是 main 函数中最后调用的函数 do\_it()



```

671
672 unchain do_it ();
673 command /daytime found
674 and /sa return 0;; pack
675 } command not found
676

```

Figure 3.14

do\_it()最外层是一个 while 循环，控制直到发包收包完成， while 里面是一个 for 循环，控制每一次发的包，根据不同的调用参数设置，在 for 里面实现收包的最长等待时间-w， 发包间隔-z，以及每次同时发几个包出去（程序执行的时间忽略不计） -s

下面是整个 do\_it 的代码（由于代码很长，这里就不完整贴出来的），do\_it 这个函数在我们实现新功能的时候会在里面关键的地方添代码，所以我在附录 A 完整的列出 do\_it 的代码，方便对照查看(截图太小装不下，所以以文字的形式)

### 3.4 发包收包控制模块

这就是我们上面说到的 do\_it()函数,do\_it()函数在 main 函数的最后调用,它返回之后整个程序也就执行完了,do\_it 的完整程序我们放在附录中,这里只列出其关键代码说明其功能,且第四章中我们在说新功能实现的时候,会反复提到 do\_it 函数,且为了测试程序功能,实现新功能,会在 do\_it 中添加很多打印代码等

do\_it 中会调用各 module 中自己的函数实现发包收包,但在 do\_it 中完成了数据包的分析,数据包分析模块负责从网络中接收数据包，判断是否是本次探测所发送的数据包，并分析是中间路由器还是目的地址所恢复的，然后从数据包中提取出有效的信息，并传递 IP 地址信息给地理位置显示模块，并且显示探测结果给用户。

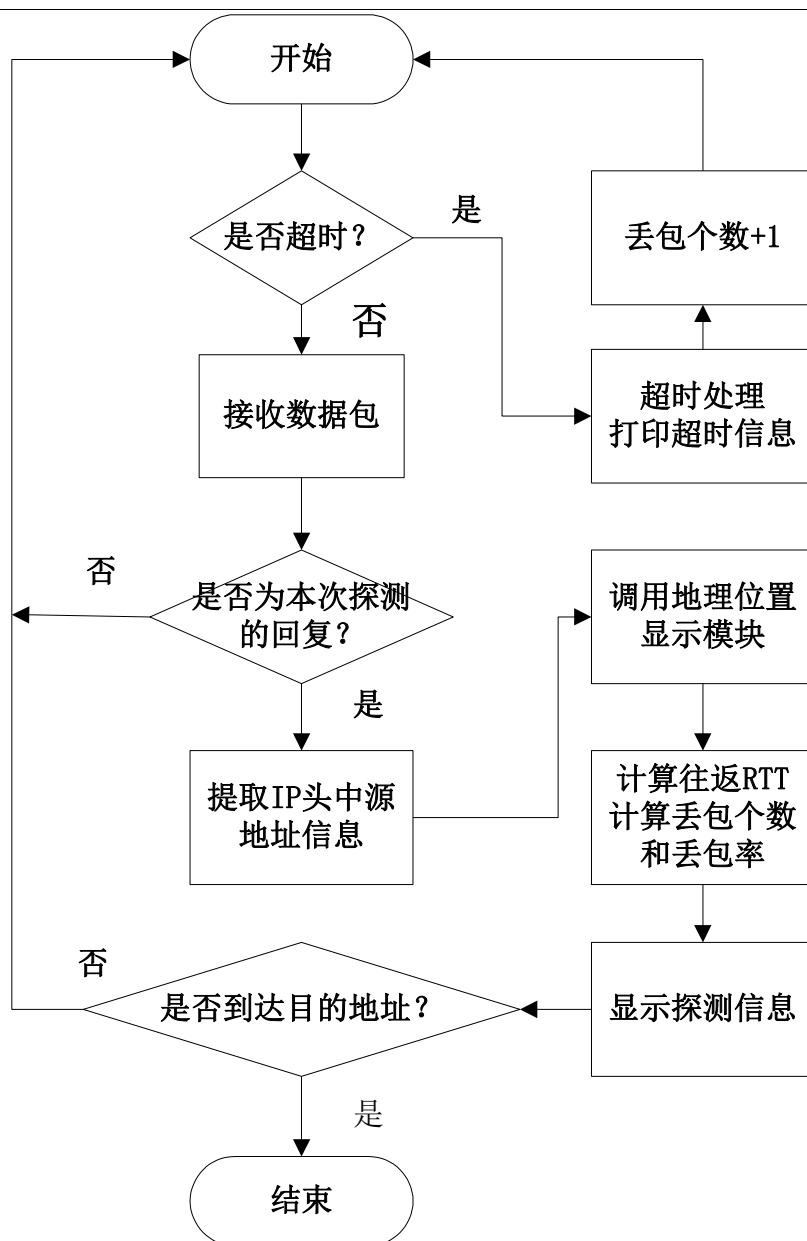


图 3.15 数据包分析模块流程图

数据包分析模块的流程如上图所示：首先启动一个定时器，时刻检验是否发超过设定的超时时间，如果超时，即转到超时处理。如果没有超时，在设定时间内接收到了数据包，那么数据分析模块首先判断这个数据包是不是回复本次探测，如果是就接收，不是就丢弃，继续接收网络上的数据包。接收到数据包后，

数据包分析模块根据各协议的特点，提取出 IP 地址，往返 RTT 的信息等。

在 do\_it 中,首先定义了 start, end 来确定发包的起始数(-f,可以设置使第一跳不为 1)

```
while (start < end)
```

这个 while 会一直运行,直到发包收包完全结束, 每确定收到一个包或确定这个包超时了,start 才会加 1 ,

while 里面有一个 for 循环, 用于一次性发的包控制,可能会因各种原因跳出 for 循环, 比如一次性发包个数到了, 发包间隔时间原因, 它也是从 start 到 end

我们现在来看看代码

超时处理

```
if (!pb->done &&
982                pb->send_time &&
983                now_time - pb->send_time >= wait_secs
984            ) {
985                ops->expire_probe (pb);

986                check_expired (pb);
987            }
```

若包已发出并收到 ( pb->done 进行控制)

```
990            if (pb->done) {
991
992                if (n == start) {    /*  can print it now    */
993                    print_probe (pb);
994                    start++;
995                }
996
997                if (pb->final)  //
998                    end = (n / probes_per_hop + 1) *probes_per_hop;
1000                continue;
1001            }
```

若还未发包

---

```

        if (!pb->send_time) {
1005                int ttl;
1006
1007                if (send_secs && (now_time - last_send) < send_secs) {
1008                        max_time = (last_send + send_secs) - wait_secs;
1009                        break;
1010                }
1012                ttl = n / probes_per_hop + 1;
1014                ops->send_probe (pb, ttl);

```

关键是一句的 调用 module ops->send\_probe  
收包的控制

```

        if (max_time) {
1038                double timeout = (max_time + wait_secs) - now_time;
1040                if (timeout < 0)  timeout = 0;
1042                do_poll (timeout, poll_callback);
1043        }
1045    }

```

这个 if 在 for 循环之后, 当 for 循环被迫中断后, 就会进行收包过程, 全依靠 do\_poll 这个调用, do\_poll 是定义在 poll.c 中的函数, 就是通过 poll 函数检查之前发出的包的 port 哪些可以接收数据了, 若可以, 则一个个接收, 并记录接收时间, 若成功接收 则设置 probe->done(), 在下面一小节的各模块内部实现我们会看到各 module 对包处理的实现

### 3.5 各 module 内部实现

数据包构造模块的主要功能是构造 traceroute 探测所需要的数据包, 包括数据包 IP 头部的构造和各协议数据包的构造。

如下图所示，数据包构造模块的流程流程是：

1. 创建一个原始套接口
2. 利用 `setsockopt` 函数设置选项
3. 初始化 TTL 和发包数
4. 根据当前的 TTL 值初始化 IP 头部
5. 根据探测类型构造数据包(ICMP,UDP,TCP)
6. 发送数据包
7. 休眠 `timeval[]`的时间，其中 `timeval` 是泊松发包模块所提供的泊松间隔



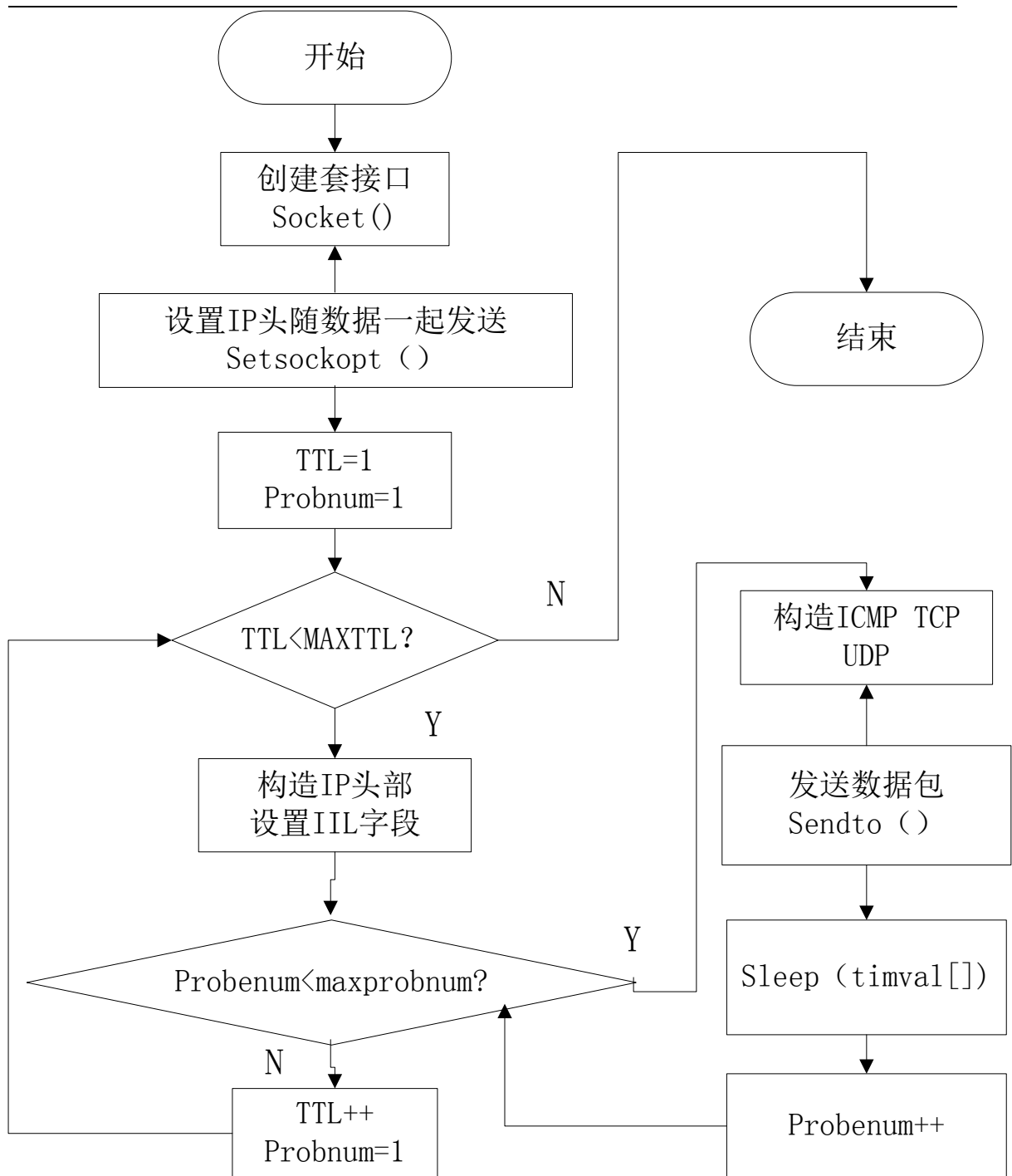


图 3.16 数据包构造模块流程图

为了能够方便的构造上文提到的各种报文，在这里采用原始套接口，自行填充 IP 数据报和内部封装的报文的所有字段。

在这里需要用到 `setsockopt()` 函数，该函数用于设置任意类型，任意套接口的选项值，在这里我们设置选项为 `IP_HDRINCL` 和层次为 `IPPROTO_IP`。

**IP\_HDRINCL**：通过设置该选项，可以自行构造 IP 头。对于生成和发送 UDP 和 TCP 协议报文，这是必须的，但是对于生成和发送 ICMP 报文，就不是必须的，为了统一处理，设置了该选项，不论 IP 数据报中封装的何种协议报文，均需手动构造 IP 头。这样就可以手动设置 IP 的 TTL 字段，进行 `traceroute` 探测。

**IPPROTO\_IP**：由于程序中的原始 socket 可能涉及 ICMP、UDP、TCP 三种协议报文的发送和接收，而他们都是被封装在 IP 数据报中的，所以将该原始 socket 协议的类型设置为 `IPPROTO_IP`，这样就可以兼顾这三种协议报文。

各协议包的实现具体如下：

#### ● IP 头部

IP 头部的构造需要用到库函数中的 `netinet/ip.h` 文件中关于 IP 结构的定义，需设置 TTL 字段为递增的值，关键字段赋值如下：

表 3-1 IP 头部关键字赋值

<code>ip_header-&gt;id</code>	<code>getpid()</code>
<code>ip_header-&gt;ttl</code>	<code>ttn;</code>
<code>ip_header-&gt;protocol</code>	UDP/ICMP/TCP

#### ● ICMP 数据包

ICMP 数据包的构造需要用到库函数中的 `netinet/ip_icmp.h` 文件中关于 ICMP 结构的定义。使用 ICMP 方式进行 `traceroute` 路由探测是，需要构造 ICMP 应答请求包，需要设置 ICMP 包的 TYPE 字段为 8，CODE 字段为 0。关键字段赋值如下：

表 3-2 ICMP 数据包关键字赋值

<code>icmp-&gt;icmp_type</code>	8
<code>icmp-&gt;icmp_code</code>	0
<code>icmp-&gt;icmp_id</code>	<code>getpid()</code>

#### ● UDP 数据包

UDP 数据包的构造需要用到库函数中的 `Netinet/udp.h` 文件中关于 UDP 结构的定义.使用 UDP 方式进行 `traceroute` 路由探测时, 需要把 UDP 的源端口设置为一个和本进程 ID 相关的一个值, 将 UDP 的目的端口设置为一个不可达的端口值。

表 3-3 UDP 数据包关键字赋值

<code>udp-&gt;source</code>	<code>sport(getpid()+3000)</code>
<code>udp-&gt;dest</code>	大端口号

### ● TCP 数据包

TCP 的构造需要用到库函数中的 `Netinet/tcp.h` 文件中关于 ICMP 结构的定义。使用 TCP 方式进行 `traceroute` 路由探测时, 需要把 TCP 的目的端口设置为 80, SYN 字段设置为 1, 以进行 TCP 连接的第一次握手。

表 3-4 TCP 数据包关键字赋值

<code>tcp-&gt;source</code>	<code>sport(getpid()+3000)</code>
<code>tcp-&gt;dest</code>	80
<code>tcp-&gt;syn</code>	1;

在 3.2 的模块调用分中我们已经看到各模块都实现了自己的 `send_probe`, `recv_probe`, `check_expired`(超时处理), `init_module()`

下面我们以 默认的 module 来做一个简单的分析,这个 module 定义在 `mod-udp.c` 中,发送的是 udp 包,

这是模块数据结构定义

```
static tr_module default_ops = {
```

```
204     .name = "default",
```

```
205     .init = udp_default_init,
```

```
206     .send_probe = udp_send_probe,
```

```
207         .recv_probe = udp_recv_probe,  
  
208         .expire_probe = udp_expire_probe,  
  
209         .user = 1,  
  
210         .header_len = sizeof (struct udphdr),  
  
211     };
```

上面各个函数也是在这个文件中定义的。

## 第 4 章 各功能模块的实现

新增功能分析：

经过前期的对现有 traceroute 工具的了解和一些实际网络测量任务中的分析，从以下几个方面对 traceroute 工具进行改进

### 4.1 traceroute 安全性改进

由于 traceroute 程序中需要创建原始套接口，需要用户得到特权，这样如果长时间的运行程序，就有可能因为额外的特权，而造成系统的漏洞，让入侵者有机可趁。

如果我们在创建原始套接口之后，马上收回用户的特权，这样在长时间运行程序中，就不会造成漏洞。所以在本系统中，将对 traceroute 的安全性进行改进。可以利用 `getuid()`取得当前用户的实际有效 ID，再利用 `setuid()`函数设置为当前用户实际 ID，这样就收回了用户的特权。

### 4.2 泊松发包

传统的 traceroute 对于发包间隔没有明确的规定，但是因为每一个数据包处理的时间基本相同，因此可简单认为是周期抽样。

周期抽样是一种最简单的抽样方式，每隔固定时间产生一次抽样。因为简单，所以应用的很多。，但采用周期采样存在两个潜在的问题：

第一，如果进行测量的对象本身具有周期性的行为，那么采样过程将仅仅得到周期性行为的一部分，即只能测到一个相位，这样会使得采样在较大程度上不能真实反映出被测对象的全部特性；

第二，周期性的采样行为使得被测对象受到影响，重复周期性采样的扰动会使网络进入同步状态，虽然一次的影响不至于引起很大的误差，但是周期性的测量行为会加强这种影响，最终会产生太大的误差而引起测量失真。

在 RFC2330 中，推荐泊松抽样，它的时间间隔符合泊松分布。泊松采样具用以下几个特点：

第一，它是无偏的。即使用泊松采样时，每一个随机过程的到达都以同样的

概率被采集到;

第二, 泊松采样是非同步的;

第三, 当新的样本到来时, 泊松采样不能通过预测而确定。

所以, 我们希望在这方面对 traceroute 工具进行改进, 可以使 traceroute 实现泊松发包, 这样可以更好对网络状况进行采样, 对我们分析网络状况有所帮助。

实现泊松发包, 实际上是实现发包间隔时间指数分布, 因为发包是一个不间断的连续性动作, 这样在单位时间内发包的个数会服从泊松分布

下面来看我们实现的代码

首先在 do\_it() 的开头添加如下代码

```
static void do_it (void) {
    int start = (first_hop - 1) * probes_per_hop;
    int end = num_probes;
    double last_send = 0; //用来记录当前最后一个包的 probe->send_time
    int i = 0; //used for time_interval
    num_sending_probes = end - start + 1;
    time_interval = calloc(num_sending_probes, sizeof(double));
    if (!time_interval) error("calloc time_interval");
    for (; i < num_sending_probes; i++)
        time_interval[i] = poisson(1.0);
}
```

Figure 4.1

往后缩进的代码是我自己添加的

首先定义了循环变量 i 用于 for 循环, 然后给 time\_interval 动态分配空间, time\_interval 是我们定义的一个 static 全局变量,

```
static int num_sending_probes = 0;
double * time_interval = 0;
```

并且调用

poisson 这个自定义的函数为 time\_interval 赋值, 这个函数我们定义在 traceroute.c 的结尾处。

```

1642 // 这个函数名虽然叫 poisson,但返回的随机数服从的是 指数分布(也就
1643 //是发包间隔时间),使单位时间发包数量服从poisson分布,miu是 间隔期望值
1644 double poisson (double miu)
1645 {
1646     double result = -miu * log(rand()/RAND_MAX);
1647     if (result < 3*miu)
1648         return result;
1649     else return 3*miu;
1650 }

```

Figure 4.2 自定义的 poisson 函数

这个函数返回服从指数分布的随机数,值得注意的是,由于这个数是我们用于设置发包间隔时间的,而理论上返回数是可以无穷大的,为了避免出现很大的数使的程序老是在等待发包(在实际的工作中遇到了这样的情况,就是程序在某次收包后迟迟没有下个包的信息,开始我还以为是程序哪出错了) 所以我们在当随机数大于  $3 \times \text{miu}$  时( $\text{miu}$  是发包间隔时间的均值),就简单的取  $3 \times \text{miu}$  做为返回值

```

1012 ops->send_probe (pb, ttl); // ttl 是在 set_ttl中, 用setsockopt 用来设为 socket 的属性了
1013 send_secs = time_interval[i++];

```

在程序发包之后,这里我们添上了设置 `send_secs` 的代码,也就是这次发包后到下次发包的间隔时间

`print_end` 这个函数是打印所有收包工作结束后的信息,本来只有一句

`printf("\n");`

我们在 `print_end()`中加上了打印包统计和 依次打印出了所有发出包的间隔时间,代码如下

```

785 int i = 0;
786 for (i=0 ; i<num_sending_probes ; i++)
787 {
788     printf("%5f ", time_interval[i]);
789     if (i % 6 == 0)
790         printf("\n");
791 }

```

Figure 4.3

这里给大家展示下, 每个发送包之间的间隔时间,这是我自己测试的例子, 间

隔时间的均值取的是 2.0 秒。

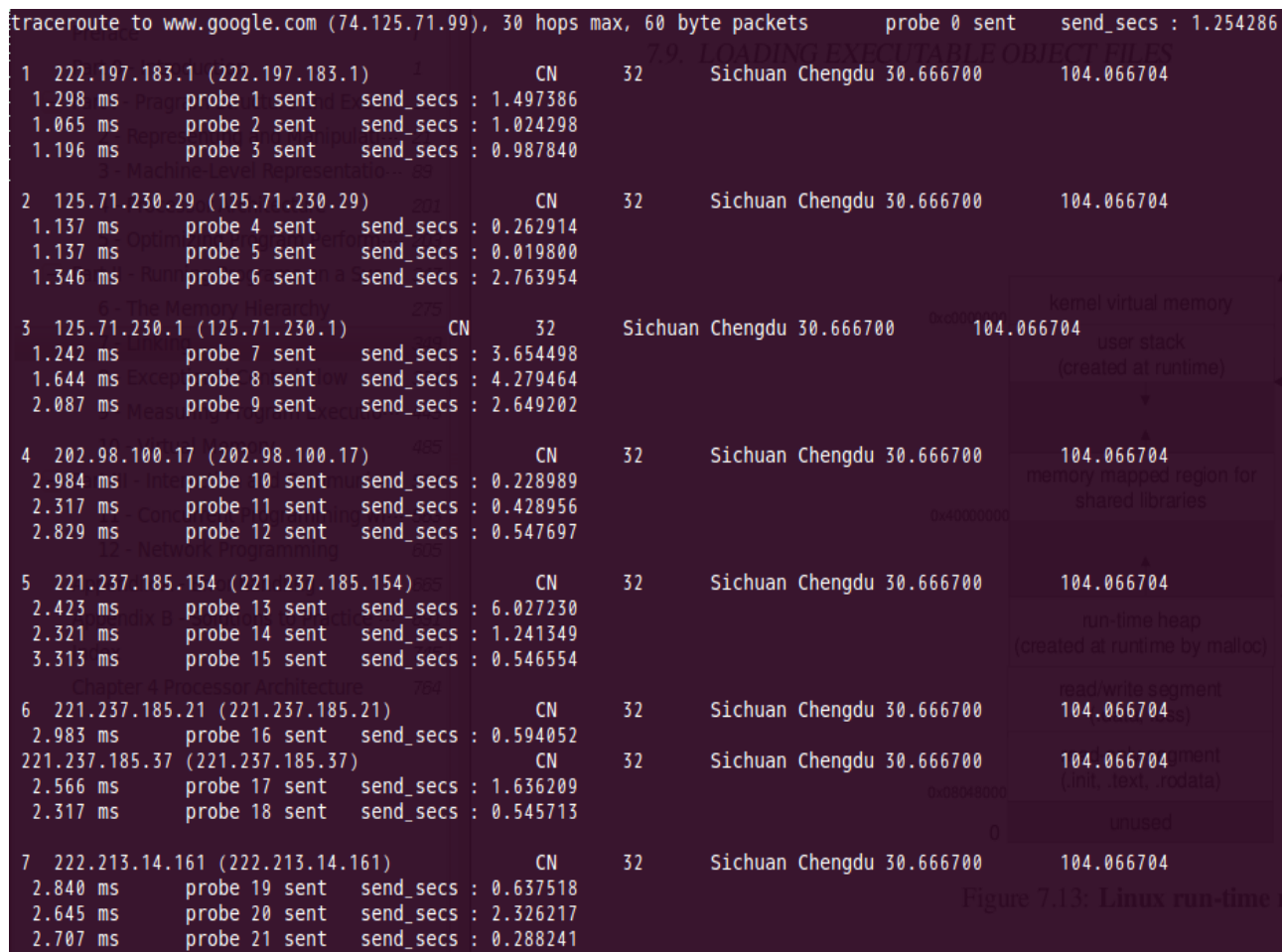


Figure 4.4 poisson 分布发包示例

### 4.3 地理位置显示

传统的 traceroute 的探测结果一般只能显示某一跳的 IP 地址，当用户想知道这个 IP 地址是哪个国家或者哪个城市时，往往需要手动的查找，这样就会给用户造成不方便。因此，

希望增加一个根据 IP 地址显示地理位置的功能。

经过调研，常用的实现 IP 地址到地理位置的数据库有以下两种：QQ 纯真数据库和 MaxMind 数据库。

QQ 纯真数据库的优点是可以显示中文地理位置，并且非常具体，特别是查找



国内的 IP 地址信息，但是对于国外的 IP 数据不是很全，很多国外的 IP 地址无法找到相应的地理位置。另一个缺点是由于记录数太多，查找速度受到影响，并且使用方法复杂，需要了解数据的存储格式去读取，实现查找的代码量大。

相对于 QQ 纯真数据库，MaxMind 数据库的不足之处在于：只提供英文地理位置，免费的数据库只能精确到城市。但是 MaxMind 数据库接口清晰，调用简单，实现简单，提供了现成 linux 环境下的 C 接口，便于程序的实现。同时 MaxMind 数据库还可以提供 IP 地址对应的精度，纬度，邮编等信息，这有利于本系统的课扩展性。故选用了 MaxMind 数据库来实现 IP 地址到地里位置映射这个功能。

下载 MaxMind 的 C 接口并安装后，利用 MaxMind 免费数据库 GEOLITECITY 库，根据 traceroute 探测到的 IP 地址提取地理位置信息，并显示在 traceroute 探测结果后。由于这里我们使用的是 MaxMind 免费数据库，所以只能精确到城市一级。

这里是我们为实现地理位置信息显示而添加的相关代码

我们首先下载如下两个数据库文件到我们的相关目录

```
xiaoyu@chain:~/Documents/traceroute/traceroute-2.0.16/traceroute$ ls -l /usr/local/share/GeoIP/
total 29688
-rw-r--r-- 1 root root 1156240 2011-04-04 18:34 GeoIP.dat
-rw-r--r-- 1 xiaoyu xiaoyu 29202422 2011-04-01 14:49 GeoLiteCity.dat
```

我们在 traceroute.c 中包含如下两个头文件

```
30
31 #include <GeoIP.h>
32 #include <GeoIPCity.h>
```

然后在 traceroute.c 的最后实现如下函数

```

1581 void print_maxmind (sockaddr_any * res)
1582 {
1583     printf("\nprint_maxmind called for ip < %s > ", 10 - 12 - 14 - 16 - 18 - 20 - 22 - 24 - 26 - 28 - 30);
1584     print_addr (res);    printf(">\n");
1585     const char * host ;
1586     host = addr2str (res);
1587
1588     GeoIP *gi;
1589     GeoIPRecord *gir;
1590
1591
1592     gi = GeoIP_open("/usr/local/share/GeoIP/GeoLiteCity.dat", GEOIP_INDEX_CACHE);
1593     if ( NULL == gi ){
1594         fprintf(stderr, "Error opening database\n");
1595         perror("");
1596         exit(1);
1597     }
1598
1599     gir = GeoIP_record_by_name(gi, host);
1600
1601     if (gir != NULL) {
1602         // ret = GeoIP_range_by_ip(gi, (const char *) host);
1603         // time_zone = GeoIP_time_zone_by_country_and_region(gir->country_code, gir->region);
1604         printf("\t\t%s\t%s\t%s\t%s\t%f\t%f\n", // host,
1605             _mk_NA(gir->country_code),
1606             _mk_NA(gir->region),
1607             _mk_NA(GeoIP_region_name_by_code(gir->country_code, gir->region)),
1608             _mk_NA(gir->city),
1609             // _mk_NA(gir->postal_code),
1610             gir->latitude,
1611             gir->longitude
1612             // gir->metro_code,
1613             // gir->area_code,
1614             // _mk_NA(time_zone),
1615             // ret[0],
1616             // ret[1]
1617             );
1618         // GeoIP_range_by_ip_delete(ret);
1619         GeoIPRecord_delete(gir);
1620     }
1621 }

```

Figure 4.5 ip 地址信息打印函数

在 `print_probe`, 即打印每个 `probe` 时, 调用这个函数

```

748
749     if (pb->ext) printf (" <%s>", pb->ext);
750                     print_maxmind(&pb->res);
751     if (backward && pb->recv_ttl) {
752         int hops = ttl2hops (pb->recv_ttl);
753         if (hops != ttl) printf (" '-%d'", hops);

```

## 4.4 丢包率统计

在数据包分析模块中，1.1.3 所述操作中，添加丢包率统计：首先启动一个定时器，如果超时，未收到测量包对应的反馈包，则认为包已丢。如果没有超时，则认为包未丢。

```

1361         if (!err)
1362             memcpy (&pb->res, &from, sizeof (pb->res)); // !! 这里, 把 pb->res
1363
1364             printf("probe %d received\n", pb->probe_number);
1365             num_recieved++;

```

我在 `recv_reply` 这个函数中添加了如下的代码

如图，最后两行是我添加的，`pb->probe_number` 是我为 `probe struct` 添加的一个成员变量(`probe struct` 的介绍见第三章源码分析)

```

21 struct probe_struct {
22     int probe_number;
23     int done;

```

`pb->probe_number` 在发包的时候赋值,在 `do_it()`中

这里的 `n` 是 `for` 循环的循环变量,表示发的第几个包,

```

1014         ops->send_probe (pb, ttl); // ttl 是在 set_ttl中, 用setsockopt 用来设为 socket 的属性了
1015         pb->probe_number = n ;
1016         send_secs = time_interval[n] ; // printf("the ii in send_secs = time_i
1017         printf("        probe %d sent    send_secs : %5f\n" , n , send_secs);

```

接下来运行程序的时候我们就可以看到发包和收包的过程了,以下截一个片断图。

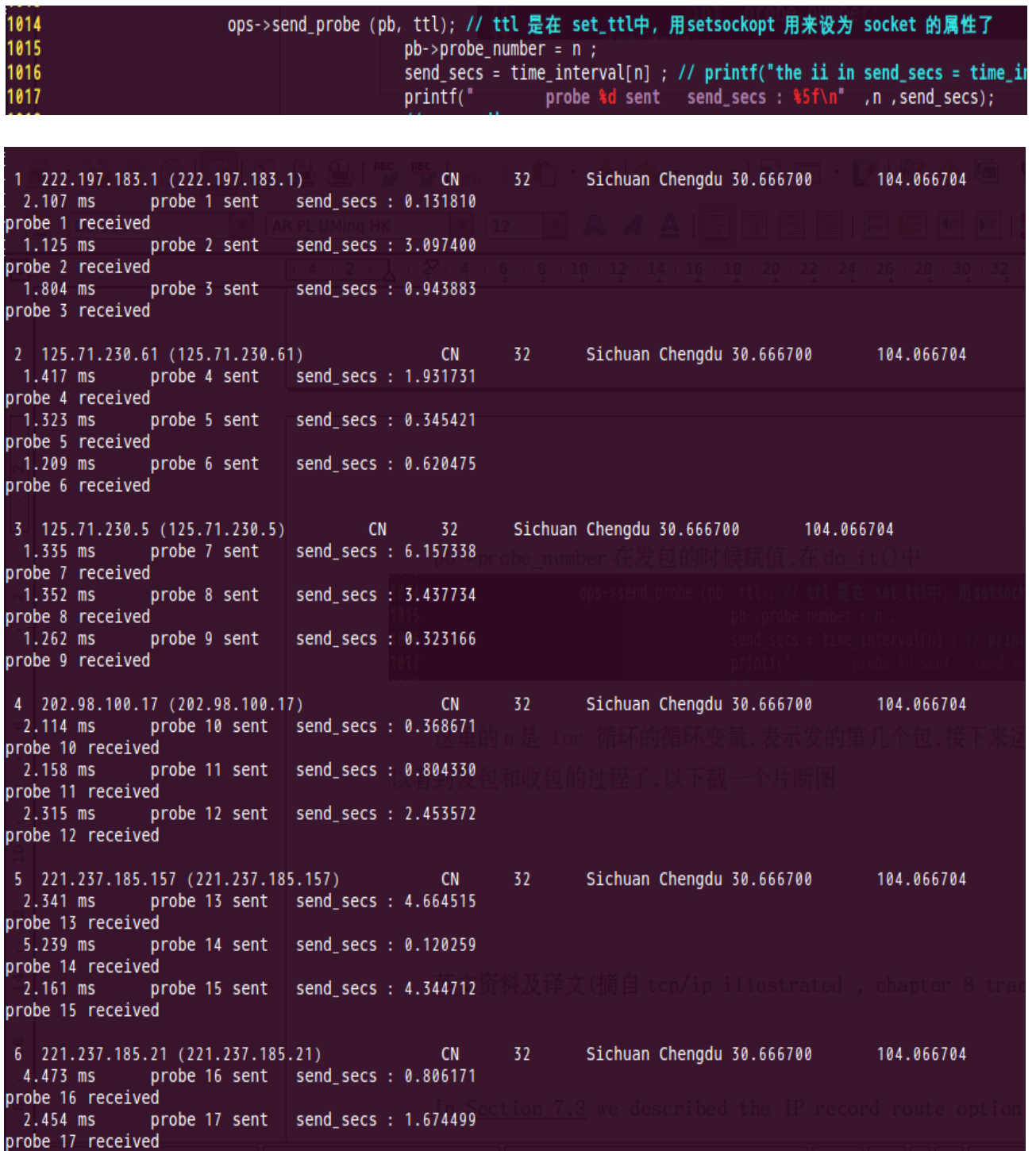


Figure 4.6

## 第五章 功能测试总结

这一章的内容是一些截图, 有些前面没有给出的, 总结实现功能后的测试情况

### 5.1 测试环境

intel X86 architecture

linux 2.6.34 kernel

### 5.2 IP 地址物理地理信息显示

首先我们 traceroute 的是 人人网的服务器, 使用的域名是 [www.renren.com](http://www.renren.com) 我们可以看到最后 traceroute 到人人网的服务器是在北京

```
xiaoyu@chain: ~/Documents/traceroute/traceroute-2.0.16/traceroute$ ./traceroute www.renren.com
set host called, arg is www.renren.com
quit print_addr without any print(addr is empty)
60.29.242.220 (60.29.242.220)
packet length : -1
header length : 28
data length : 32
opts_idx : 1
first hop : 1
probes_per_hop: 3
num_probes : 90
dst_port_seq : 0
num_gateways : 0
send_secs : 0.00
opts[0] : (null)
traceroute to www.renren.com (60.29.242.220), 30 hops max, 60 byte packets
 1  121.49.86.1 (121.49.86.1)      CN      32      Sichuan Chengdu 30.666700      104.066704
    0.542 ms  1.084 ms  1.353 ms
 2  125.71.231.21 (125.71.231.21)  CN      32      Sichuan Chengdu 30.666700      104.066704
    0.335 ms  0.348 ms  0.406 ms
 3  125.71.230.33 (125.71.230.33)   CN      32      Sichuan Chengdu 30.666700      104.066704
    0.320 ms  0.378 ms  0.430 ms
 4  125.71.228.89 (125.71.228.89)  CN      32      Sichuan Chengdu 30.666700      104.066704
    0.466 ms  0.573 ms  0.586 ms
 5  202.98.100.17 (202.98.100.17)  CN      32      Sichuan Chengdu 30.666700      104.066704
    1.653 ms  1.584 ms  2.194 ms
 6  221.237.185.73 (221.237.185.73)  CN      32      Sichuan Chengdu 30.666700      104.066704
    2.493 ms  2.275 ms  2.182 ms
 7  221.237.185.37 (221.237.185.37)  CN      32      Sichuan Chengdu 30.666700      104.066704
    1.626 ms  221.237.185.21 (221.237.185.21)
    2.170 ms  221.237.185.37 (221.237.185.37)
    2.089 ms
 8  222.213.14.153 (222.213.14.153)  CN      32      Sichuan Chengdu 30.666700      104.066704
    15.002 ms  14.998 ms  222.213.5.9 (222.213.5.9)
    3.629 ms
 9  202.97.36.109 (202.97.36.109)  CN      N/A      N/A      N/A      35.000000      105.000000
    15.745 ms  202.97.36.6 (202.97.36.6)
    15.304 ms  15.912 ms
10  202.97.38.162 (202.97.38.162)  CN      N/A      N/A      N/A      35.000000      105.000000
    26.138 ms  218.30.19.193 (218.30.19.193)
    16.458 ms  15.997 ms
11  218.30.69.22 (218.30.69.22)     CN      22      Beijing Beijing 39.928902      116.388298
    16.110 ms  16.038 ms  218.30.69.30 (218.30.69.30)
    25.785 ms
12  * * *
13  * * *
```

Figure 5.1



下面我们 traceroute [www.google.com](http://www.google.com) google 的服务器

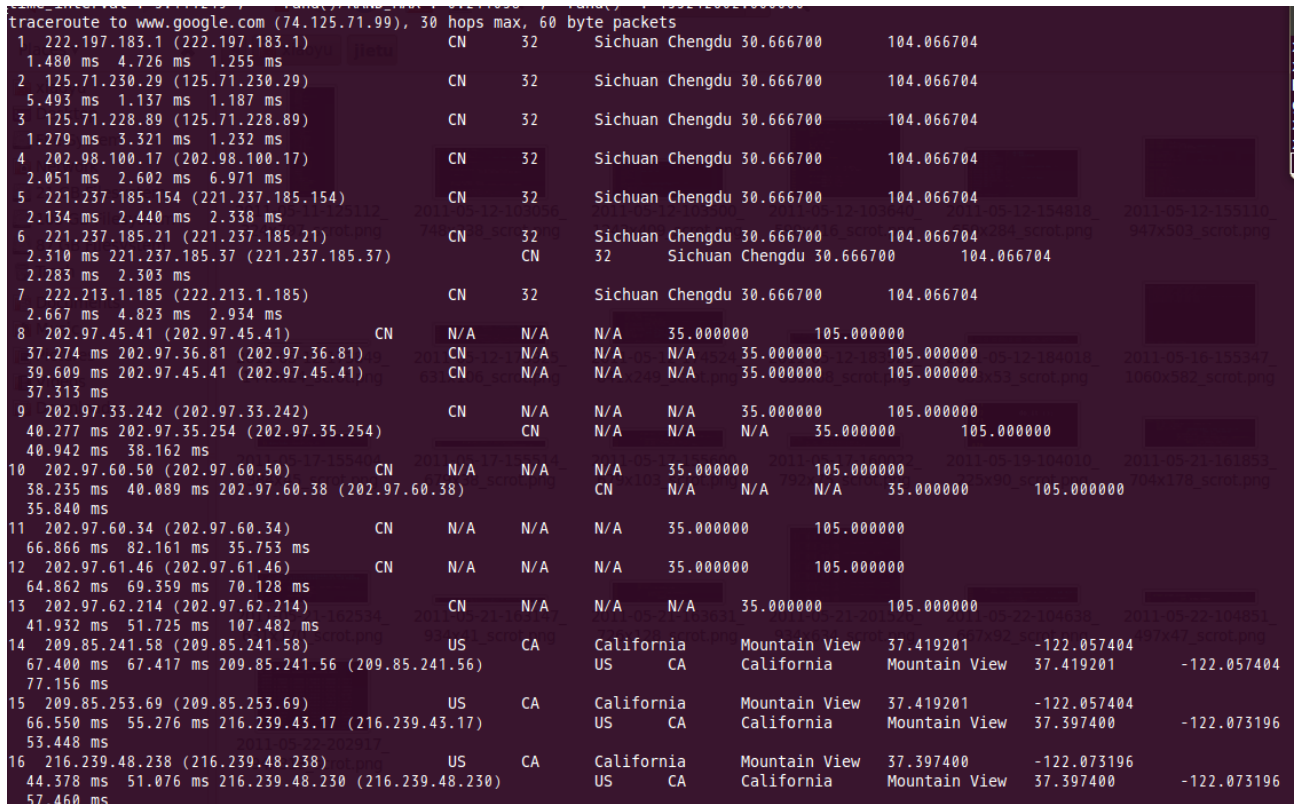
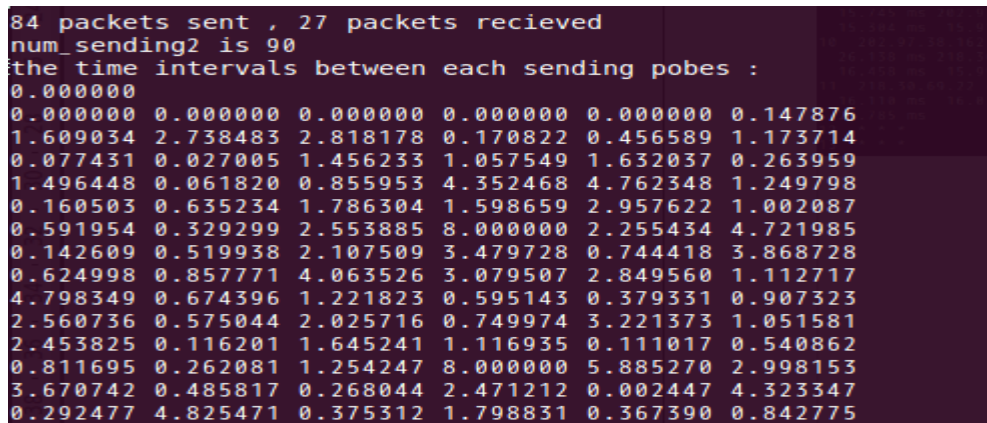


Figure 5.2

可以看到 google 的服务器最后是在 美国加州。

### 5.3 发包间隔时间图

这是一个服从泊松分布的发包过程的发包间隔时间表



## 5.4 包统计数

这是刚才那个 google 的例子, 可以看到第一行是发包以及成功收包数的统计

```
91 packets sent , 48 packets recieved
the time intervals between each sending pokes :
0.057659
0.973552 4.863111 3.038782 0.190601 0.852784 3.372797
0.771406 1.947381 2.455390 1.524522 4.440055 0.738260
6.296328 2.529104 3.566044 1.487846 1.020645 2.528464
4.233822 1.954019 2.309919 0.258872 1.612333 3.980446
5.144664 1.449282 0.953581 1.329932 4.333200 1.839097
1.443616 0.631693 1.440630 0.699915 0.897985 3.939965
0.233285 2.289877 1.318860 3.397955 0.484516 0.937602
0.268878 0.378005 0.192762 6.324268 2.387845 1.352896
2.249144 1.718719 0.244664 0.893007 2.394497 2.208728
0.505849 1.943706 0.406878 1.846131 0.227072 0.144095
0.456316 1.942839 0.832068 2.527465 4.971493 2.421802
1.725232 0.054400 0.968458 0.125448 3.715322 1.827460
1.142020 6.996964 2.949725 1.497013 5.246171 1.262824
0.037363 1.845775 0.091515 0.286940 6.581194 2.715043
3.241317 0.412184 0.906123 8.000000 3.111249 0.000000
```

## 结束语

通过这次毕设我学到很多东西, 包括网络编程知识, 一个成熟的小软件是怎么用 C 语言各文件组织起来的, traceroute 还可以从其它方面进行改进。

最后的成果, 增强型的 traceroute 达到了课题的要求和之前的预定, 准确的显示出 IP 物理地址和实现泊松发包及包统计

本课题进一步需要解决的是还有一些用于发包的协议可以新成新的模块添加到源代码树中且在安全性的改进上还可以再进一步



---

## 参考文献

GARY R. Wright & W.Richard Stevens . TCP/IP illustrated volume 1 protocol 北京:人民邮电出版社 2010

W.Richard Stevens & Bill Fenner & Andrew M.Rudoff . Unix network programming .volume 1 the sockets 北京:清华大学出版社 2006

Eric S. Raymond . The art of unix programming(英文版) .  
北京:人民邮电出版社 2006

W.Richard Stevens & Stephen A.Rago . Advanced Programming in the unix environment(2ed). 北京: 人民邮电出版社 2006

## 附录

do\_it()的代码

```
static void do_it (void) {
    int start = (first_hop - 1) * probes_per_hop;
    int end = num_probes;
    double last_send = 0;
    print_header ();

    while (start < end) {
        int n, num = 0;
        double max_time = 0;
        double now_time = get_time ();

        for (n = start; n < end; n++) {
            probe *pb = &probes[n];

            if (!pb->done &&
                pb->send_time &&
                now_time - pb->send_time >= wait_secs
            ) {
                ops->expire_probe (pb);
                check_expired (pb);
            }

            if (pb->done) {

                if (n == start) { /* can print it now */
```

---

```
    print_probe (pb);
    start++;
}

if (pb->final)

end = (n / probes_per_hop + 1) * probes_per_hop;

continue;

}

if (!pb->send_time) {

    int ttl;

    if (send_secs && (now_time - last_send) < send_secs) {

max_time = (last_send + send_secs) - wait_secs;

break;

    }

    ttl = n / probes_per_hop + 1;

ops->send_probe (pb, ttl);
```

```
    if (!pb->send_time) {

        if (max_time) break;    /* have chances later    */

        else error ("send probe");

    }

    last_send = pb->send_time;

}

if (pb->send_time > max_time)

    max_time = pb->send_time;


num++;

if (num >= sim_probes) break;

}

if (max_time) {

    double timeout = (max_time + wait_secs) - now_time;

    if (timeout < 0) timeout = 0;
```

---

```
do_poll (timeout, poll_callback);

    }

}

print_end ();

return;
```

## 英文资料及译文

摘自 tcp/ip illustrated , chapter 8 traceroute)

In [Section 7.3](#) we described the IP record route option (RR). Why wasn't this used instead of developing a new application? There are three reasons. First, historically not all routers have supported the record route option, making it unusable on certain paths. (Traceroute doesn't require any special or optional features at any intermediate routers.) Second, record route is normally a one-way option. The sender enables the option and the receiver has to fetch all the values from the received IP header and somehow return them to the sender. In [Section 7.3](#) we saw that most implementations of the Ping server (the ICMP echo reply function within the kernel) reflect an incoming RR list, but this doubles the number of IP addresses recorded (the outgoing path and the return path), which runs into the limit described in the next paragraph. (Traceroute requires only a working UDP module at the destination-no special server application is required.) The third and major reason is that the room allocated for options in the IP header isn't large enough today to handle most routes. There is room for only nine IP addresses in the IP header options field. In the old days of the ARPANET this was adequate, but it is far too small nowadays.

Traceroute uses ICMP and the TTL field in the IP header. The TTL field (time-to-live) is an 8-bit field that the sender initializes to some value. The recommended initial value is specified in the Assigned Numbers RFC and is currently 64. Older systems would often initialize it to 15 or 32. We saw in some of the Ping examples in [Chapter 7](#) that ICMP echo replies are often sent with the TTL set to its maximum value of 255.

Each router that handles the datagram is required to decrement the TTL by either one or the number of seconds that the router holds onto the datagram. Since most routers hold a datagram for less than a second, the TTL field has effectively become a hop counter, decremented by one by each router.

RFC 1009 [Braden and Postel 1987] required a router that held a datagram for more

---

than 1 second to decrement the TTL by the number of seconds. Few routers implemented this requirement. The new Router Requirements RFC [Almquist 1993] makes this optional, allowing a router to treat the TTL as just a hop count.

The purpose of the TTL field is to prevent datagrams from ending up in infinite loops, which can occur during routing transients. For example, when a router crashes or when the connection between two routers is lost, it can take the routing protocols some time (from seconds to a few minutes) to detect the lost route and work around it. During this time period it is possible for the datagram to end up in routing loops. The TTL field puts an upper limit on these looping datagrams.

When a router gets an IP datagram whose TTL is either 0 or 1 it must not forward the datagram. (A destination host that receives a datagram like this can deliver it to the application, since the datagram does not have to be routed. Normally, however, no system should receive a datagram with a TTL of 0.) Instead the router throws away the datagram *and* sends back to the originating host an ICMP "time exceeded" message. The key to Traceroute is that the IP datagram containing this ICMP message has the router's IP address as the source address.

We can now guess the operation of Traceroute. It sends an IP datagram with a TTL of 1 to the destination host. The first router to handle the datagram decrements the TTL, discards the datagram, and sends back the ICMP time exceeded. This identifies the first router in the path. Traceroute then sends a datagram with a TTL of 2, and we find the IP address of the second router. This continues until the datagram reaches the destination host. But even though the arriving IP datagram has a TTL of 1, the destination host won't throw it away and generate the ICMP time exceeded, since the datagram has reached its final destination. How can we determine when we've reached the destination? Traceroute sends UDP datagrams to the destination host, but it chooses the destination UDP port number to be an unlikely value (larger than 30,000), making it improbable that an application at the destination is using that port. This causes the destination host's UDP module to generate an ICMP "port unreachable" error (Section 6.5) when the datagram arrives. All Traceroute needs to do is differentiate between the received ICMP

messages-time exceeded versus port unreachable-to know when it's done.

The Traceroute program must be able to set the TTL field in the outgoing datagram. Not all programming interfaces to TCP/IP support this, and not all implementations support the capability, but most current systems do, and are able to run Traceroute. This programming interface normally requires the user to have superuser privilege, meaning it may take special privilege to run it on your host.

### 8.3 LAN Output

We're now ready to run traceroute and see the output. We'll use our simple internet (see the figure on the inside front cover) going from svr4 to slip, through the router bsdi. The hardwired SLIP link between bsdi and slip is 9600 bits/sec.

```
svr4 % traceroute slip
```

```
traceroute to slip (140.252.13.65), 30 hops max. 40 byte packets
```

```
1 bsdi (140.252.13.35) 20 ms 10 ms 10 ms
```

```
2 slip (140.252.13.65) 120 ms 120 ms 120 ms
```

The first unnumbered line of output gives the name and IP address of the destination and indicates that traceroute won't increase the TTL beyond 30. The datagram size of 40 bytes allows for the 20-byte IP header, the 8-byte UDP header, and 12 bytes of user data. (The 12 bytes of user data contain a sequence number that is incremented each time a datagram is sent, a copy of the outgoing TTL, and the time at which the datagram was sent.)

The next two lines in the output begin with the TTL, followed by the name of the host or router, and its IP address. For each TTL value three datagrams are sent. For each returned ICMP message the round-trip time is calculated and printed. If no response is received within 5 seconds for any of the three datagrams, an asterisk is printed instead and the next datagram is sent. In this output the first three datagrams had a TTL of 1 and the ICMP messages were returned in 20, 10, and 10 ms. The next three datagrams were sent with a TTL of 2 and the ICMP messages were returned 120 ms later. Since the TTL of 2 reached the final destination, the program then stopped.

The round-trip times are calculated by the traceroute program on the sending host. They are the total RTTs from the traceroute program to that router. If we're interested in the per-hop time we have to subtract the value printed for TTL *N* from the value printed for



---

TTL  $N+1$ .

Figure 8.1 shows the tcpdump output for this run. As we might have guessed, the reason that the first probe packet to bsdi had an RTT of 20 ms and the next two had an RTT of 10 ms was because of an ARP exchange, tcpdump shows this is indeed the case.

The destination UDP port starts at 33435 and is incremented by one each time a datagram is sent. This starting port number can be changed with a command-line option. The UDP datagram contains 12 bytes of user data, which we calculated earlier when traceroute output that it was sending 40-byte datagrams.

Next, tcpdump prints the comment [ttl 1] when the IP datagram has a TTL of 1. It prints a message like this when the TTL is 0 or 1, to warn us that something looks funny in the datagram. Here we expect to see the TTL of 1, but with some other application it could be a warning that the datagram might not get to its final destination. We should never see a datagram passing by with a TTL of 0, unless the router that put it on the wire is broken.

1	0.0	arp who-has bsdi tell svr4
2	0.000586 (0.0006)	arp reply bsdi is-at 0:0:c0:6f:2d:40
3	0.003067 (0.0025)	svr4.42804 > slip.33435: udp 12 [ttl 1]
4	0.004325 (0.0013)	bsdi > svr4: icmp: time exceeded in-transit
5	0.069810 (0.0655)	svr4.42804 > slip.33436: udp 12 [ttl 1]
6	0.071149 (0.0013)	bsdi > svr4: icmp: time exceeded in-transit
7	0.085162 (0.0140)	svr4.42804 > slip.33437: udp 12 [ttl 1]
8	0.086375 (0.0012)	bsdi > svr4: icmp: time exceeded in-transit
9	0.118608 (0.0322)	svr4.42804 > slip.33438: udp 12
10	0.226464 (0.1079)	slip > svr4: icmp: slip udp port 33438 unreachable
11	0.287296 (0.0608)	svr4.42804 > slip.33439: udp 12
12	0.395230 (0.1079)	slip > svr4: icmp: slip udp port 33439 unreachable

```

13    0.409504 (0.0143)    s      : udp 12
14    0.517430 (0.1079)    s      lp port 33440 unreachable

```

**Figure 8.1** tcpdump output showing the ICMP message format.

The ICMP message "time exceeded" is the one we expect to see from the router bsdi, since it will decrement the TTL. The message comes from the router even though the IP datagram that caused the error is going to slip. There are two different ICMP "time exceeded" messages (Figure 6.3), each with a different *code* field in the ICMP header. Figure 8.2 shows the format of this ICMP error message.

**Figure 8.2** ICMP time exceeded message.

The one we've been describing is generated when the TTL reaches 0, and is specified by a *code* of 0.

It's also possible for a host to send an ICMP "time exceeded during reassembly" when it times out during the reassembly of a fragmented datagram. (We talk about fragmentation and reassembly in Section 11.5.) This error is specified by a *code* of 1. Lines 9-14 in Figure 8.1 correspond to the three datagrams sent with a TTL of 2. These reach the final destination and generate an ICMP port unreachable message.

It is worthwhile to calculate what the round-trip times should be for the SLIP link, similar to what we did in Section 7.2 when we set the link to 1200 bits/sec for the Ping example. The outgoing UDP datagram contains 12 bytes of data, 8 bytes of UDP header, 20 bytes of IP header, and 2 bytes (at least) of SLIP framing (Section 2.4) for a total of 42 bytes. Unlike Ping, however, the size of the return datagrams changes. Recall from Figure 6.9 that the returned ICMP message contains the IP header of the datagram that caused the error and the first 8 bytes of data that followed that IP header (which is a UDP header in the case of traceroute). This gives us a total of 20+8+20+8+2, or 58 bytes. With a data rate of 960 bytes/sec the expected RTT is  $(42 + 58/960)$  or 104 ms. This corresponds to the IIO-ms value measured on svr4.

The source port number in Figure 8.1 (42804) seems high. traceroute sets the source port number of the UDP datagrams that it sends to the logical-OR of its Unix process ID

---

with 32768. In case traceroute is being run multiple times on the same host, each process looks at the source port number in the UDP header that's returned by ICMP, and only handles those messages that are replies to probes that it sent.

There are several points to note with traceroute. First, there is no guarantee that the route today will be in use tomorrow, or even that two consecutive IP datagrams follow the same route. If a route changes while the program is running you'll see it occur because traceroute prints the new IP address for the given TTL if it changes.

Second, there is no guarantee that the path taken by the returned ICMP message retraces the path of the UDP datagram sent by traceroute. This implies that the round-trip times printed may not be a true indication of the outgoing and returning datagram times. (If it takes 1 second for the UDP datagram to travel from the source to a router, but 3 seconds for the ICMP message to travel a different path back to the source, the printed round-trip time is 4 seconds.)

Third, the source IP address in the returned ICMP message is the IP address of the interface on the router on which the UDP datagram *arrived*. This differs from the IP record route option (Section 7.3), where the IP address recorded was the outgoing interface's address. Since every router by definition has two or more interfaces, running traceroute from host A to host B can generate different output than from host B to host A. Indeed, if we run traceroute from host slip to svr4 the output becomes:

```
slip % traceroute svr4
```

```
traceroute to svr4 (140.252.13.34), 30 hops max, 40 byte packets
```

```
1 bsdi (140.252.13.66) 110 ms 110 ms 110 ms
```

```
2 svr4 (140.252.13.34) 110 ms 120 ms 110 ms
```

This time the IP address printed for host bsdi is 140.252.13.66, the SLIP interface, while previously it was 140.252.13.35, the Ethernet interface. Since traceroute also tries to print the name associated with an IP address, the names can change. (In our example both interfaces on bsdi have the same name.)

Consider Figure 8.3. It shows two local area networks with a router connected to each LAN. The two routers are connected with a point-to-point link. If we run traceroute

from a host on the left LAN to a host on the right LAN, the IP addresses found for the routers will be *if1* and *if3*. But going the other way will print the IP addresses *if4* and *if2*. The two interfaces *if2* and *if3* share the same network ID, while the other two interfaces have different network IDs.

**Figure 8.3** Identification of interfaces printed by traceroute.

Finally, across wide area networks the traceroute output is much easier to comprehend if the IP addresses are printed as readable domain names, instead of as IP addresses. But since the only piece of information traceroute has when it receives the ICMP message is an IP address, it does a "reverse name lookup" to find the name, given the IP address. This requires the administrator responsible for that router or host to configure their reverse name lookup function correctly (which isn't always the case). We describe how an IP address is converted to a name using the DNS in Section 14.5.

---

## 中文译文

在 7.3 节中，我们描述了 IP 记录路由选项（RR）。为什么不使用这个选项而另外开发一个新的应用程序？有三个方面的原因。首先，原先并不是所有的路由器都支持记录路由选项，因此该选项在某些路径上不能使用。（Traceroute 程序不需要中间路由器具备任何特殊的或可选的功能。）

其次，记录路由一般是单向的选项。发送端设置了该选项，那么接收端不得不从收到的 IP 首部中提取出所有的信息然后全部返回给发送端。在 7.3 节中，我们看到大多数 Ping 服务器的实现（内核中的 ICMP 回显回答功能）把接收到的 RR 清单返回，但是这样使得记录下来的 IP 地址翻了一番（一来一回），这样会受到一些限制，这一点我们在下一段讨论。（Traceroute 程序只需要目的端运行一个 UDP 模块---其他不需要任何特殊的服务器应用 T 程序。）

最后一个原因也是最主要的原因是，IP 首部中留给选项的空间有限，不能存放当前大多数的路径。在 IP 首部选项字段中最多只能存放 9 个 IP 地址。在原先的 ARPANET 中这是足够的，但是对现在来说是远远不够的。

Traceroute 程序使用 ICMP 报文和 IP 首部中的 TTL 字段。TTL 字段（生存周期）是由发送端初始设置一个 8 bit 字段。推荐的初始值由分配数字 RFC 指定，当前值为 64。较老版本的系统经常初始化为 15 或 32。我们从第 7 章中的一些 ping 程序例子中可以看出，发送 ICMP 回显回答时经常把 TTL 设为最大值 255。

每个处理数据报的路由器都需要把 TTL 的值减 1 或减去数据报在路由器中停留的秒数。由于大多数的路由器转发数据报的时延都小于 1 秒钟，因此 TTL 最终成为一个跳站的计数器，所经过的每个路由器都将其值减 1。

RFC 1009 [Braden and Postel 1987]指出，如果路由器转发数据报的时延超过 1 秒，那么它将把 TTL 值减去所消耗的时间（秒数）。但很少有路由器这么实现。新的路由器需求文档 RFC [Almquist 1993]为此指定它为可选择功能，允许把 TTL 看成一个跳站计数器。

TTL 字段的目的是防止数据报在路由选择时无休止地在网络中流动。例如，

当路由器瘫痪或者两个路由器之间的连接丢失时，路由选择协议有时会去检测丢失的路由并一直进行下去。在这段时间内，数据报可能在循环回路被终止。TTL 字段就是在这些循环传递的数据报上加上一个生存上限。

当路由器收到一份 IP 数据报，如果其 TTL 字段是 0 或 1，则路由器不转发该数据报。（接收到这种数据报的目的主机可以将它交给应用程序，这是因为不需要转发该数据报。但是在通常情况下，系统不应该接收 TTL 字段为 0 的数据报。）相反地，路由器将该数据报丢弃并给信源机发一份 ICMP“超时”信息。Traceroute 程序的关键在于包含这份 ICMP 信息的 IP 报文的信源地址是该路由器的 IP 地址。

我们现在可以猜想一下 Traceroute 程序的操作过程。它发送一份 TTL 字段为 1 的 IP 数据报给目的主机。处理这份数据报的第一个路由器将 TTL 值减 1，丢弃该数据报，并发回一份超时 ICMP 报文。这样就得到了该路径中的第一个路由器的地址。然后 Traceroute 程序发送一份 TTL 值为 2 的数据报，这样我们就可以得到第二个路由器的地址。继续这个过程直至该数据报到达目的主机。但是目的主机哪怕接收到 TTL 值为 1 的 IP 数据报，也不会丢弃该数据报并产生一份超时 ICMP 报文，这是因为数据报已经到达其最终目的地。那么我们该如

何判断已经到达目的主机了呢？

Traceroute 程序发送一份 UDP 数据报给目的主机，但它选择一个不可能的值作为 UDP 端口号（大于 30,000），使目的主机的任何一个应用程序都不可能使用该端口。因为，当该数据报到达时，将使目的主机的 UDP 模块产生一份“端口不可到达”错误（见 6.5 节）的 ICMP 报文。这样，Traceroute 程序所要做的就是区分接收到的 ICMP 信息是超时还是端口不

可到达，以判断什么时候结束。

Traceroute 程序必须可以为发送的数据报设置 TTL 字段。并非所有与 TCP/IP 接口的程序都支持这项功能，同时并非所有的实现都支持这项能力，但目前大部分系统都支持这项功能，并可以运行 Traceroute 程序。这个程序界面通常要求用户具有超级用户权限，这意味着它可能需要特殊的权限以在你的主机上运行该程序。

### 8.3 局域网输出

我们现在已经做好运行 Traceroute 程序并观察其输出的准备了。我们将使用从 svr4 到 slip, 经路由器 bsdi 的简单互连网（见内封面）。bsdi 和 slip 之间是 9600 b/s 的 SLIP 链路。

输出的第一个无标号行给出了目的主机名和其 IP 地址, 指出 traceroute 程序最大的 TTL 字段值为 30。40 字节的数据报包含 20 字节 IP 首部, 8 字节的 UDP 首部和 12 字节的用户数据。（12 字节的用户数据包含每发一个数据报就加 1 的序号, 送出 TTL 的副本以及发送数据报的时间。）

输出的后面两行以 TTL 开始, 接下来是主机或路由器名, 以及其 IP 地址。对于每个 TTL 值, 发送 3 份数据报。每接收到一份 ICMP 报文, 就计算并打印出往返时间。如果在 5 秒种内仍未收到 3 份数据报的任意一份的响应, 则打印一个星号, 并发送下一份数据报。在上述输出结果中, TTL 字段为 1 的前三份数据报的 ICMP 报文分别在 20, 10 和 10 ms 收到。TTL 字段为 2 的 3 份数据报的 ICMP 报文则在 120 ms 后收到。由于 TTL 字段为 2 到达最终目的主机, 因此程序就此停止。

往返时间是由发送主机的 traceroute 程序计算的。它是指从 traceroute 程序到该路由器的总往返时间。如果我们对每段路径的时间感兴趣, 可以用 TTL 字段为 N+1 所打印出来的时间减去 TTL 字段为 N 的时间。

图 8.1 给出了 tcpdump 的运行输出结果。正如我们所预想的那样, 第一个发往 bsdi 的探测数据报的往返时间是 20 ms 而后面两个数据报往返时间是 10 ms 的原因是发生了一次 ARP 交换。tcpdump 结果证实了确实是这种情况。

目的主机 UDP 端口号最开始设置为 33435, 且每发送一个数据报加 1。可以通过命令行选项来改变开始的端口号。UDP 数据报包含 12 个字节的用户数据, 我们在前面 traceroute 程序输出的 40 字节数据报中已经对其进行了描述。

后面 tcpdump 打印出了 TTL 字段为 1 的 IP 数据报的注释[ttl 1]。当 TTL 值为 0 或 1 时, tcpdump 打印出这条信息, 以提示我们数据报中有些不太寻常之处。在这里我们可以预见到 TTL 值为 1, 而在其它一些应用程序中, 它可以警告我们

数据报可能无法到达其最终目的主机。我们不可能看到路由器传送一个 TTL 值为 0 的数据报，除非发出该数据报的该路由器已经崩溃。

图 8.1 从 svr4 到 slip 的 traceroute 程序示例的 tcpdump 输出结果

因为 bsdi 路由器将 TTL 值减到 0，因此我们预计它将发回“传送超时”的 ICMP 报文。即使这份被丢弃的 IP 报文发送往 slip，路由器也会发回 ICMP 报文。

有两种不同的 ICMP“超时”报文（见 p.71 页的图 6.3），它们的 ICMP 报文中 code 字段不同。图 8.2 给出了这种 ICMP 错误报文的格式。

图 8.2 ICMP 超时报文

我们所讨论的 ICMP 报文是在 TTL 值等于 0 时产生的，其 code 字段为 0。

主机在组装分片时可能发生超时，这时，它将发送一份“组装报文超时”的 ICMP 报文。（我们将在 11.5 节讨论分片和组装。）这种错误报文将 code 字段置 1。

图 8.1 的第 9-14 行对应于 TTL 为 2 的 3 份数据报。这 3 份报文到达最终目的主机，并产生一份 ICMP 端口不可到达报文。

计算出 SLIP 链路的往返时间是很有意义的，就象我们在 7.2 节中所举的 Ping 例子，将链路值设置为 1200 b/s 一样。发送出动的 UDP 数据报共 42 个字节，包括 12 字节的数据，8 字节 UDP 首部，20 字节的 IP 首部以及（至少）2 字节的 SLIP 帧（2.4 节）。但是与 Ping 不一样的是，返回的数据报大小是变化的。从图 6.9 可以看出，返回的 ICMP 报文包含发生差错的数据报的 IP 首部以及紧随该 IP 首部的 8 字节数据（在 traceroute 程序中，即 UDP 首部）。这样，总共就是  $20 + 8 + 20 + 8 + 2$ ，即 58 字节。在数据速率为 960 B/s 的情况下，预计的 RTT 就是  $(42 + 58/960)$ ，即 104 ms。这个值与 svr4 上所估算出来的 110 ms 是吻合的。

图 8.1 中的源端口号（42804）看起来有些大。traceroute 程序将其发送的 UDP 数据报的源端口号设置为 Unix 进程号与 32768 之间的逻辑或值。对于在同一台主机上多次运行 traceroute 程序的情况，每个进程都查看 ICMP 返回的 UDP 首部的源端口号，并且只处理那些对自己发送回答的报文。



---

关于 `traceroute` 程序还有一些必须指出的事项。首先，并不能保证现在的路由也是将来所要采用的路由，甚至两份连续的 IP 数据报都可能采用不同的路由。如果在运行程序时，路由发生改变，你就会观察到这种变化，这是因为对于一个给定的 TTL，如果其路由发生变化，`traceroute` 程序将打印出新的 IP 地址。

第二，不能保证 ICMP 报文的路由与 `traceroute` 程序发送的 UDP 数据报采用同一路由。这表明所打印出来的往返时间可能并不能真正体现数据报发出和返回的时间差。（如果 UDP 数据报从信源到路由器的时间是 1 秒，而 ICMP 报文用另一条路由返回信源用了 3 秒时间，则打印出来的往返时间是 4 秒。）

第三，返回的 ICMP 报文中的信源 IP 地址是 UDP 数据报到达的路由器接口的 IP 地址。这与 IP 记录路由选项（7.3 节）不同，记录的 IP 地址指的是发送接口地址。由于每个定义的路由器都有 2 个或更多的接口，因此，从 A 主机到 B 主机上运行 `traceroute` 程序和从 B 主机到 A 主机上运行 `traceroute` 程序所得到的结果可能是不同的。事实上，如果从 `slip` 主机到 `svr4` 上运行 `traceroute` 程序，其输出结果变成了：

这次打印出来的 `bsdi` 主机的 IP 地址是 140.252.13.66，对应于 `SLIP` 接口，而上次的地址是 140.252.13.35，是以太网接口地址。由于 `traceroute` 程序同时也打印出与 IP 地址相关的主机名，因而主机名也可能变化。（在我们的例子中，`bsdi` 上的两个接口都采用相同的名字。）

考虑图 8.3 的情况。它给出了两个局域网通过一个路由器相连的情况。两个路由器通过一个点对点的链路相连。如果我们在左边 LAN 的一个主机上运行 `traceroute` 程序，那么它将发现路由器的 IP 地址为 `if1` 和 `if3`。但在另一种情况下，就会发现打印出来的 IP 地址为 `if4` 和 `if2`。`if2` 和 `if3` 有着同样的网络号，而另两个接口则有着不同的网络号。

图 8.3 `traceroute` 程序打印出的接口标识

最后，在广域网情况下，如果 `traceroute` 程序的输出是可读的域名形式，而不是 IP 地址形式，那么会更好理解一些。但是由于 `traceroute` 程序接收到 ICMP 报文时，它所获得的唯一信息就是 IP 地址，因此，在给定 IP 地址的情况下，它做一个“反向域名查看”工作来获得域名。这就需要路由器或主机的主管人员正确配置其反向域名查看功能（并非所有的情况下都是如此）。我们将在 14.5 节描述如何使用 DNS 将一个 IP 地址转换成域名。