

[源码赏析] NetSpeeder: 简单粗暴的服务端网络加速器

前言

首先简单分析一下这个小东西的原理：它就是直接调用 `libpcap` 来监听数据链路层的网络数据包，然后使用特定语法（也就是 `tcpdump` 的语法）来过滤出感兴趣的包，再调用 `libnet` 库暴力地把这些包重发一遍，实现发包翻倍的效果，从而能够起到降低丢包率，优化网络连接的作用。但如果你在VPS上使用这类东西，它会占用大量的国际出口带宽，本质是损人利己的行为，所以其实并不建议使用，这里只是分析一下源码而已。

然而吐个槽：就这么个简单的一百来行的小东西，扔到github上就骗了三百多个赞，而比它复杂得多高级得多的 `WinDivert` 这样的底层网络库居然一个赞都没有！于是可见轮子哥说得确实很对：当你真正解决了专家们所面对的问题的时候，并不会收获多少关注，因为有能力关注的人太少了.....

PS: 项目地址在这里 (<https://github.com/snooda/net-speeder>)。

代码注释

```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
/*
 * 简单贴一段介绍:
 * libnet是一个小型的接口函数库, 主要用C语言写成, 提供了低层网络数据包的构造、处理和发送功能。
 * libnet的开发目的是: 建立一个简单统一的网络编程接口以屏蔽不同操作系统底层网络编程的差别, 使得程序
 */
#include <libnet.h>

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 65535

#ifdef COOKED
#define ETHERNET_H_LEN 16
#else
#define ETHERNET_H_LEN 14
#endif

#define SPECIAL_TTL 88

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

void print_usage(void);

/*
 * 打印帮助你信息, Linux程序的编程习惯了, 这里没什么可说的
 */
void print_usage(void) {
    printf("Usage: %s [interface] [\"filter rule\"]\n", "net_speeder");
    printf("\n");
    printf("Options:\n");
    printf("    interface    Listen on <interface> for packets.\n");
    printf("    filter       Rules to filter packets.\n");
    printf("\n");
}

/*
 * 每次捕获到数据包, 就会自动调用这个回调函数
 * 函数的参数分别为:
 * 1. 指向用户自定义数据的指针
 * 2. 一个用来描述当前捕获的数据包的pcap自定义结构体;
 * 其中包含了时间戳、已捕获长度、整体长度等信息, 可以用于计算数据偏移, 不过一般不怎么用
 * 3. 指向当前捕获的数据的指针, 当然数据缓冲区是libpcap自己分配并管理的
 * 注意这个指针所指向数据的内容取决于接口的类型, 一般情况下是普通的IEEE 802.3 Ethernet数据链路层
 * 具体的协议类型以及规范可以参考: http://www.tcpdump.org/linktypes.html
 */
```

```

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    static int count = 1;
    struct libnet_ipv4_hdr *ip;

    // 首先我们取出传递进来的libnet句柄, 待会可以用来发包
    libnet_t *libnet_handler = (libnet_t *)args;
    count++;

    // 数据起始地址再加上以太网头长度, 就得到了IP包的头地址
    ip = (struct libnet_ipv4_hdr *)(packet + ETHERNET_H_LEN);

    if (ip->ip_ttl != SPECIAL_TTL) {
        // 为这个包设置特殊的TTL值, 这是为了表示区分
        // 因为当我们把这个包重发以后, libpcap会再次抓到这个包
        // 如果不加以区分, 就会把任何一个包无限次地重复发送, 导致大量的无意义流量
        ip->ip_ttl = SPECIAL_TTL;
        // 调用libnet的函数强行再发一遍, 如此暴力!
        int len_written = libnet_adv_write_raw_ipv4(libnet_handler, (u_int8_t *)ip, ntohs(ip->ip_len));
        // 当然发送有可能失败—网络上什么操作都可能有意外发生
        // 因此就把错误信息打印出来
        if (len_written < 0) {
            printf("packet len:[%d] actual write:[%d]\n", ntohs(ip->ip_len), len_written);
            printf("err msg:[%s]\n", libnet_geterror(libnet_handler));
        }
    } else {
        // 如果抓到的包是本程序自己重发的, 直接忽略就行了
        // The packet net_speeder sent. nothing todo
    }
    return;
}

// 这么短的初始化函数非要单独拆分出来, 也是有点画蛇添足
libnet_t *start_libnet(char *dev) {
    char errbuf[LIBNET_ERRBUF_SIZE];
    libnet_t *libnet_handler = libnet_init(LIBNET_RAW4_ADV, dev, errbuf);

    if (NULL == libnet_handler) {
        printf("libnet_init: error %s\n", errbuf);
    }
    return libnet_handler;
}

#define ARGV_NUM 3

int main(int argc, char **argv) {
    // 设备名
    char *dev = NULL;
    // 错误信息缓冲区 (这是libpcap的函数调用约定, 多数函数都要接受一个缓冲区地址, 用于存储缓冲区数据)
    char errbuf[PCAP_ERRBUF_SIZE];
    // 这个当然就是libpcap一个会话的句柄, 需要调用pcap的对应函数进行初始化
    pcap_t *handle;

    // 指向过滤规则字符串的指针, 这个字符串用于传入给libpcap来生成相应的BPF规则, 从而实现过滤出用户需
    // 详情可以参考https://developer.apple.com/library/mac/documentation/Darwin/Reference
    char *filter_rule = NULL;
    // 刚才说libpcap会把字符串形式的规则转换为BPF所识别的规则, 因此这个结构体就是用来存储生成的规则

```

```
struct bpf_program fp;
bpf_u_int32 net, mask;

// 这里检查参数, 并为设备名和过滤规则赋初值
if (argc == ARGV_NUM) {
    dev = argv[1];
    filter_rule = argv[2];
    printf("Device: %s\n", dev);
    printf("Filter rule: %s\n", filter_rule);
} else {
    // 如果参数不对就报错退出
    print_usage();
    return -1;
}

// 一般情况下以太网帧头的长度是14字节, 但看起来有些应用的长度会是16字节
// 这里会把这个信息打印出来
printf("ethernet header len: [%d](14:normal, 16:cooked)\n", ETHERNET_H_LEN);

// 调用libpcap的库函数来检查设备的子网掩码和IP地址
// 当然调用有可能失败并且这是正常的, 比如对于某些虚拟设备如Mac上的pktap, 自然是没有真实地址的
// 此时直接把地址和掩码都置零就可以了
if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
    printf("Couldn't get netmask for device %s: %s\n", dev, errbuf);
    net = 0;
    mask = 0;
}

// 打开一个pcap会话句柄, 相关参数可以参考手册
printf("init pcap\n");
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
if (handle == NULL) {
    printf("pcap_open_live dev:[%s] err:[%s]\n", dev, errbuf);
    printf("init pcap failed\n");
    return -1;
}

// 同样地, 初始化一个libnet会话句柄
printf("init libnet\n");
libnet_t *libnet_handler = start_libnet(dev);
if (NULL == libnet_handler) {
    printf("init libnet failed\n");
    return -1;
}

// 下面继续调用libpcap的函数, 把过滤规则转换为BPF底层规则
if (pcap_compile(handle, &fp, filter_rule, 0, net) == -1) {
    printf("filter rule err:[%s][%s]\n", filter_rule, pcap_geterr(handle));
    return -1;
}

// 然后把这条规则应用到打开的pcap会话上面
if (pcap_setfilter(handle, &fp) == -1) {
    printf("set filter failed:[%s][%s]\n", filter_rule, pcap_geterr(handle));
    return -1;
}
```

```
// 终于完成了冗长的初始化工作, 可以开始调用pcap_loop()函数监听数据包了
// 这个函数本身接受四个参数:
// 1. pcap会话句柄
// 2. 捕获多少个包后退出循环
// 3. 捕获每个包后, 以指向这个包的指针以及其他一些参数来调用的回调函数
// 4. 用户自定义的数据结构
// 我们可以看出, 为了让捕获每个数据包的时候能够调用libnet重发一遍,
// 这里把libnet的句柄作为用户自定义数据传给了pcap_loop()
// 但其实并没有必要用这样蛋疼的循环方式, 因为只要第二个参数是负数, pcap_loop()就会无限循环.....
while (1) {
    pcap_loop(handle, 1, got_packet, (u_char *)libnet_handler);
}

// 虽然pcap_loop()是一个无限循环, 但在某种情况下, 比如注册了signal handler之后, 是可以被打断的
// 尽管这个程序中没有实现相应的机制, 但还是应该把清理工作做好, 比如关闭对应的句柄, 这是好的编程习惯
/* cleanup */
pcap_freecode(&fp);
pcap_close(handle);
libnet_destroy(libnet_handler);
return 0;
}
```

0条评论

 陈亚奇 ▼

陈

开始讨论...



分享


最佳 最新 最早

来做第一个留言的人吧!



订阅

隐私

不要出售我的数据

 发布时间:

2015-08-31

 分类:研究 (<https://blog.finaltheory.me/research/index.html>) 标签:C (<https://blog.finaltheory.me/tag/c.html>)⁷ libpcap (<https://blog.finaltheory.me/tag/libpcap.html>)¹

+ 目录

[源码赏析] NetSpeeder: 简单粗暴的服务端网络加速器

1. 前言
2. 代码注释
3. 评论

Powered by Pelican (<https://github.com/getpelican/pelican>), theme built with Bootstrap3 (<http://getbootstrap.com>) modified by FinalTheory (<https://github.com/FinalTheory/pelican-theme>), icons by Font Awesome (<http://fontawesome.github.io/Font-Awesome>).

COPYRIGHT © 2013-2022 FinalTheory (<https://blog.finaltheory.me>) / LICENSE (/LICENSE.txt)