第二章 网络信息获取

1、被动获取技术

第二章　网络信息获取

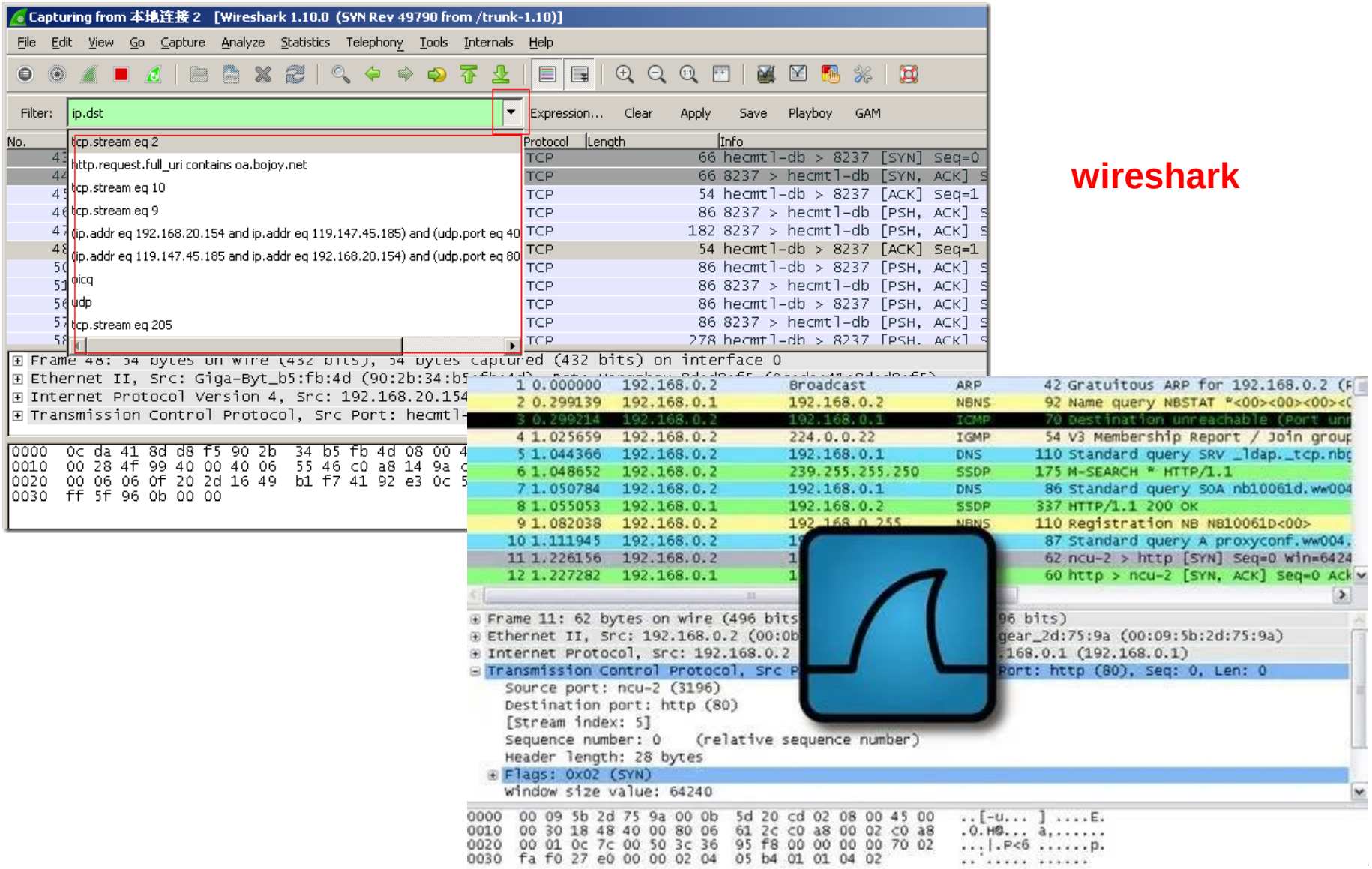
1、被动获取

- 网络基础知识
  - ✓ 以太网广播
  - ✓ TCP/IP协议族
- 被动流量监测的关键技术
- 常用网络开发包
- 高性能捕包
  - ✓零拷贝
  - ✓bpf/xdp
  - ✓dpdk

2、主动获取

- Web信息获取
- 社交网络信息获取
- P2P网络信息获取

什么是被动捕包?



wireshark

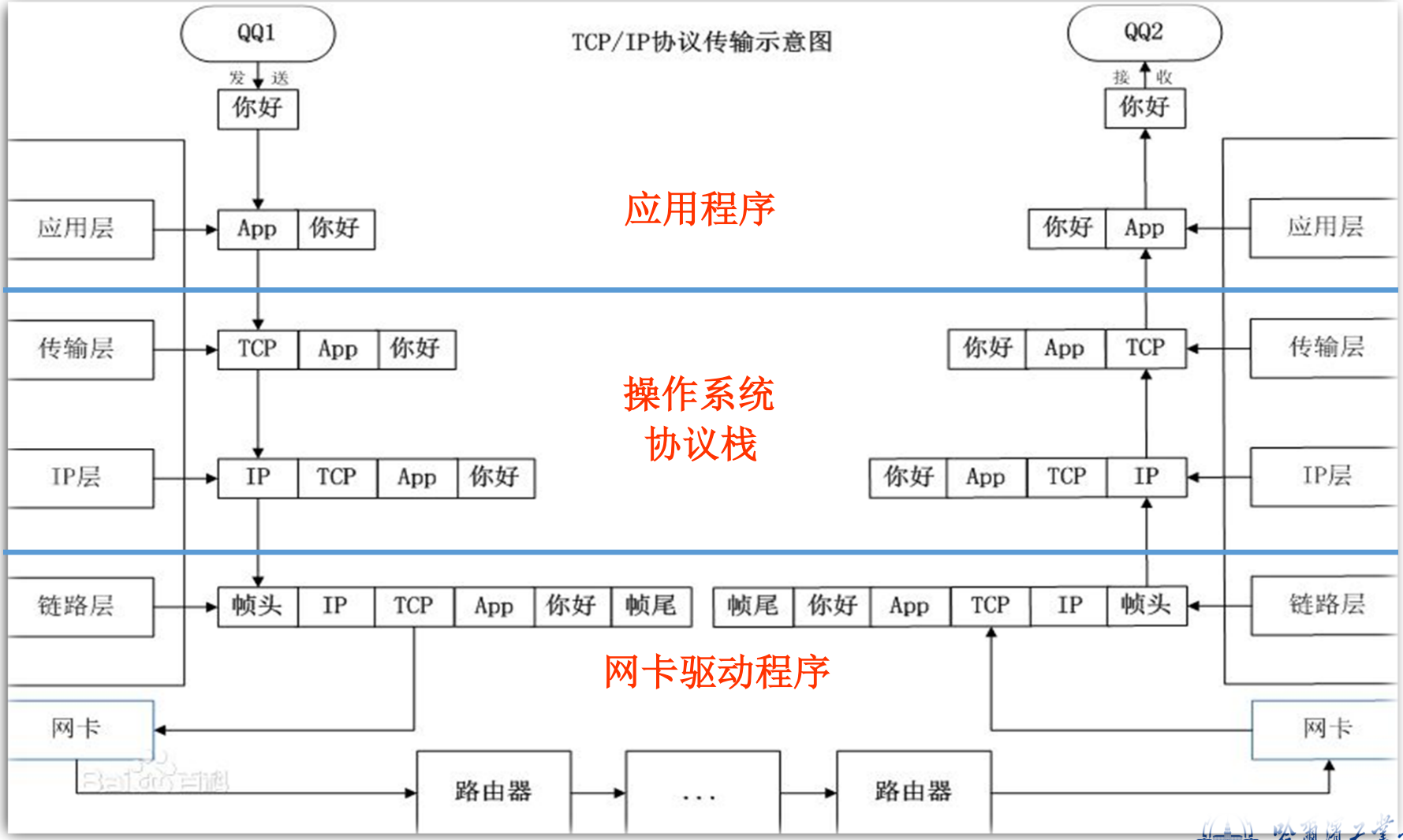
什么是被动捕包?

```
[root@localhost ~]# tcpdump host 192.168.1.254 -c 3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
13:52:47.182023 IP 192.168.1.2 > 192.168.1.254: ICMP echo request, id 10681, seq 1, length 64
13:52:47.190937 IP 192.168.1.254 > 192.168.1.2: ICMP echo reply, id 10681, seq 1, length 64
13:52:48.183178 IP 192.168.1.2 > 192.168.1.254: ICMP echo request, id 10681, seq 2, length 64
3 packets captured
3 packets received by filter
0 packets dropped by kernel
[root@localhost ~]#
```

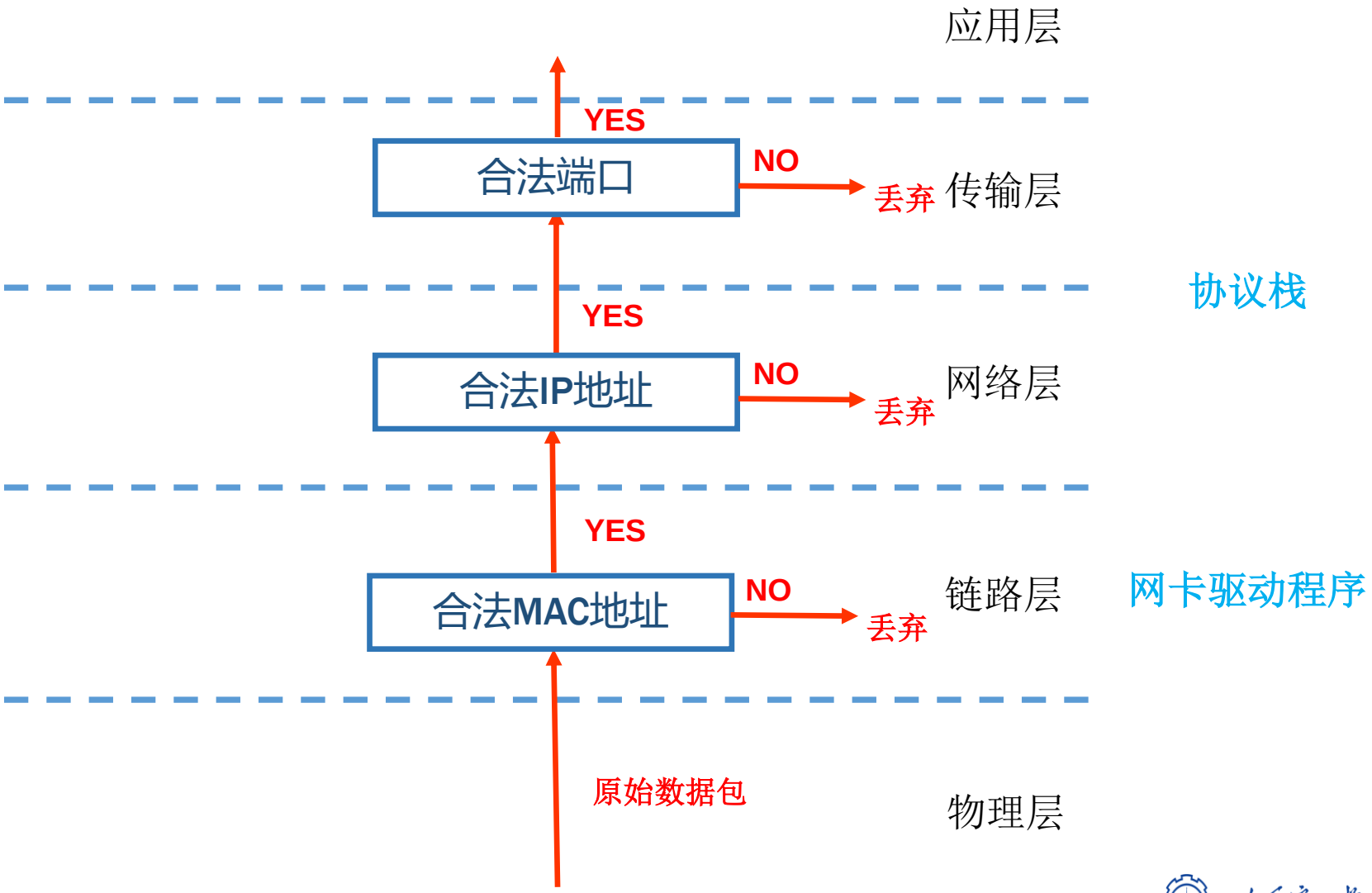
tcpdump

```
root@ubuntu:/home/peng# tcpdump -v
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
01:51:30.100405 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto ICMP (1), length 84)
    ubuntu > 39.156.69.79: ICMP echo request, id 4726, seq 1253, length 64
01:51:30.103766 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto UDP (17), length 71)
    ubuntu.29231 > 192.168.43.20.domain: 64543+ PTR? 79.69.156.39.in-addr.arpa. (43)
01:51:30.160929 IP (tos 0x0, ttl 64, id 36315, offset 0, flags [DF], proto UDP (17), length 71)
    192.168.43.20.domain > ubuntu.29231: 64543 NXDomain* 0/0/0 (43)
01:51:30.197365 IP (tos 0x74, ttl 46, id 0, offset 0, flags [DF], proto ICMP (1), length 84)
    39.156.69.79 > ubuntu: ICMP echo reply, id 4726, seq 1253, length 64
01:51:30.197609 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto UDP (17), length 71)
    ubuntu.53096 > 192.168.43.20.domain: 45581+ PTR? 79.69.156.39.in-addr.arpa. (43)
01:51:30.223801 IP (tos 0x0, ttl 64, id 36318, offset 0, flags [DF], proto UDP (17), length 71)
    192.168.43.20.domain > ubuntu.53096: 45581 NXDomain* 0/0/0 (43)
01:51:35.163660 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto UDP (17), length 72)
    ubuntu.40879 > 192.168.43.20.domain: 40635+ PTR? 83.43.168.192.in-addr.arpa. (44)
```

TCP/IP协议


TCP/IP协议传输示意图
QQ1
发
送
你好
应用层
App
你好
传输层
TCP
App
你好
IP层
IP
TCP
App
你好
链路层
帧头
IP
TCP
App
你好
帧尾
网卡
路由器
...
路由器
网卡
链路层
帧尾
你好
App
TCP
IP
帧头
IP层
你好
App
TCP
IP
传输层
你好
App
TCP
应用层
你好
App
接
收
QQ2
应用程序
操作系统
协议栈
网卡驱动程序

数据包接收过程


应用层
YES
合法端口
NO
丢弃
传输层
协议栈
YES
合法IP地址
NO
丢弃
网络层
YES
合法MAC地址
NO
丢弃
链路层
网卡驱动程序
原始数据包
物理层

二、以太网

以太网最初是由XEROX公司研制,并且在1980年由数据设备公司DEC(DIGIAL EQUIPMENT CORPOR ATION)、INTEL公司和XEROX公司共同使之规范成形。后来它被作为802.3标准为电气与电子工程师协会（IEEE）所采纳。

以太网是最为流行的网络传输系统之一。以太网的基本特征是采用一种称为载波监听多路访问/冲突检测CSMA/CD (Carrier Sense Multiple Access/ Collision Detection)的共享访问方案。

TCP/IP与以太网

- 以太网和TCP/IP可以说是相辅相成的。
  - 以太网在一二层提供物理上的连线，使用48位的MAC地址
  - TCP/IP工作在上层，使用32位的IP地址
  - 两者间使用ARP和RARP协议进行相互转换。

- 载波监听
  - 指在以太网中的每个站点都具有同等的权利，在传输自己的数据时，首先监听信道是否空闲，如果空闲，就传输自己的数据，如果信道被占用，就等待信道空闲。

- 冲突检测
  - 为了防止发生两个站点同时监测到网络没有被使用时而产生冲突。以太网采用广播机制，所有与网络连接的工作站都可以看到网络上传递的数据。

以太网的广播通讯

- 在以太网中，所有的通讯都是广播的
  - 通常在同一个网段的所有网络接口都可以访问在物理媒体上传输的所有数据

- 网卡的MAC地址
  - 每一个网络接口都有一个唯一的硬件地址，这个硬件地址也就是网卡的MAC地址 。
  - 大多数系统使用48比特的地址，这个地址用来表示网络中的每一个设备
  - 一般来说每一块网卡上的MAC地址都是不同的
  - 每个网卡厂家得到一段地址，然后用这段地址分配给其生产的每个网卡一个地址。

以太网的广播通讯

- 在正常的情况下，一个网络接口应该只响应这样的两种数据帧：
  1.与自己mac地址相匹配的数据帧。
  2.发向所有机器的广播数据帧。

- 数据的收发是由网卡来完成的
  - 网卡接收到传输来的数据，网卡内的单片程序接收数据帧的目的MAC地址，根据计算机上的网卡驱动程序设置的接收模式判断该不该接收。
  - 认为该接收就接收后产生中断信号通知CPU
  - 认为不该接收就丢掉不管，所以不该接收的数据网卡就截断了，计算机根本就不知道
  - CPU得到中断信号产生中断，操作系统就根据网卡的驱动程序设置的网卡中断程序地址调用驱动程序接收数据
  - 驱动程序接收数据后放入信号堆栈让操作系统处理。

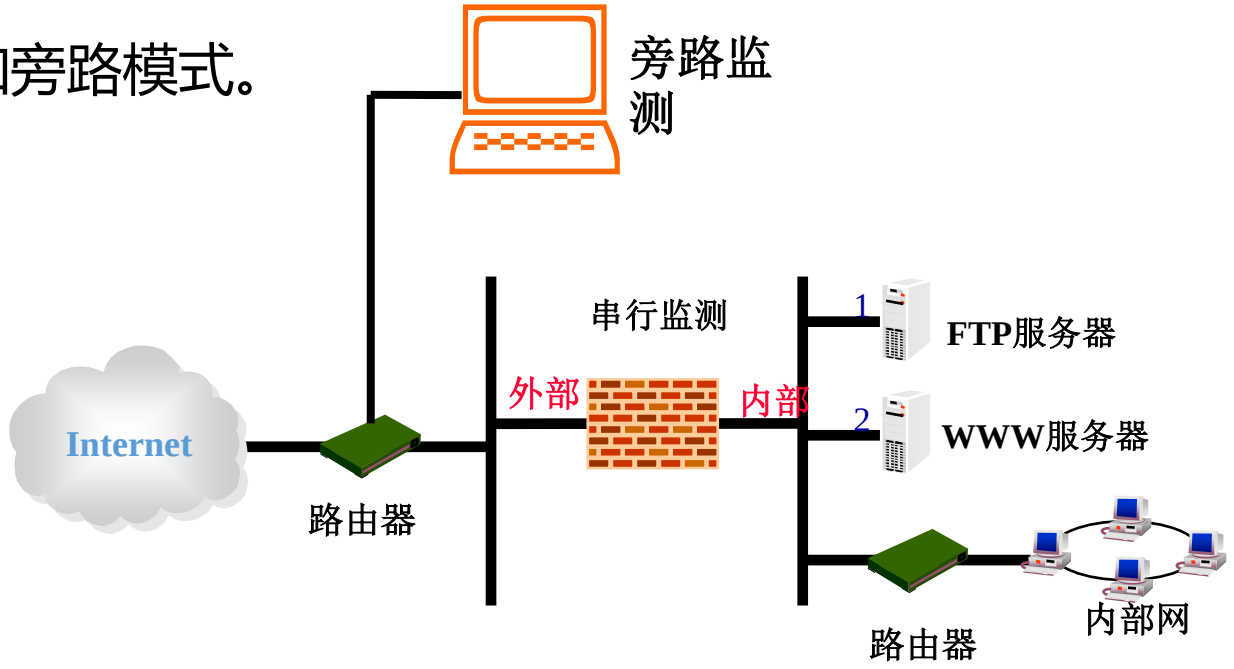
以太网的广播通讯

- 网卡来说一般有四种接收模式：
  ① 广播方式：该模式下的网卡能够接收网络中的广播信息。
  ② 组播方式：设置在该模式下的网卡能够接收组播数据。
  ③ 直接方式：在这种模式下，只有目的网卡才能接收该数据。
  ④ 混杂模式：在这种模式下的网卡能够接收一切通过它的数据，而不管该数据是否是传给它的。

- 总结一下
  - 以太网中是基于广播方式传送数据的，也就是说，所有的物理信号都要经过各机器
  - 网卡的一种模式叫混杂模式（promiscuous），在这种模式下工作的网卡能够接收到一切通过它的数据

网络信息被动获取

- 网络信息被动获取是指通过物理线路接入到实际的网络中，实现获取该网络当前传输的所有信息，即获取当前传输的数据包，并根据信息的源主机、目标主机、服务协议和端口等信息简单过滤掉不关心的垃圾数据，然后提交给上层应用程序进行进一步处理。
- 所谓被动是指只能通过监听和嗅探的模式，收集流经设备的数据流量。
- 分为串行和旁路模式。



常见网络设备

集线器

集线器，差不多就是个多端口的中继器，把每个输入端口的信号放大再发到别的端口去，集线器可以实现多台计算机之间的互联，因为它有很多的端口，每个口都能连计算机。

交换机

交换机工作在数据链路层，比集线器智能一些，能分辨出帧中的源MAC地址和目的MAC地址，因此可以在数据帧的始发者和目标接收者之间建立临时的交换路径，使数据帧直接由源地址到达目的地址。

旁路

网桥

网桥工作在数据链路层，将两个LAN连起来，根据MAC地址来转发帧，可以看作一个“低层的路由器”。

网关，路由器

网关，连接两个不同网络的接口，比如局域网的共享上网服务器就是局域网和广域网的接口。

路由器的基本功能是，把数据（IP报文）传送到正确的网络

串行

网络流监控模式

- 串联监控模式
  - 一般是通过网关或者网桥的模式来进行监控
- 旁路监控模式
  - 一般是指通过交换机等网络设备的“端口镜像”功能来实现监控

旁路部署起来比较灵活方便，不会影响现有的网络结构，串行需要对现有网络结构进行变动

旁路模式对原始传递的数据包不会造成延时，不会对网速造成任何影响。而串联模式是串联在网络中的，那么所有的数据必须先经过监控系统，通过监控系统的分析检查之后，才能够发送到各个客户端，所以会对网速有一定的延时。

旁路监控设备一旦故障或者停止运行，不会影响现有网络的正常原因。而串联监控设备如果出现故障，会导致网络中断，导致网络单点故障。

旁路监控模式局限性

- 数据获取：旁路需要交换机支持端口镜像才可以实现监控。

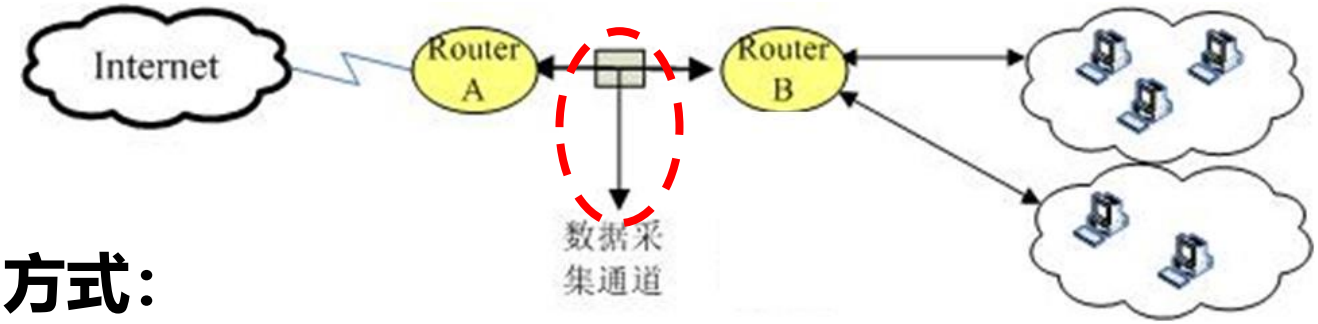
- 数据管控：旁路模式采用发送RST包的方式来断开TCP连接，不能禁止UDP通讯。对于UDP应用，一般还需要在路由器上面禁止UDP端口进行配合。而串联模式不存在该问题。

串行适用的网络产品：防火墙， 数据包过滤设备

旁路适用的网络产品：内容审计，内容分析，监测设备

旁路监测技术

数据分流



分流的方式：

1. **分光器** 将光缆上的光信号直接备份出一份，最简单的方式，设备要求最高
2. **路由交换** 交换机需要通过镜像端口
3. HUB 集线器能够看到所有端口的报文

网络数据包捕获技术

不同的操作系统实现的底层包捕获机制可能是不一样的，但从形式上看大同小异。数据包常规的传输路径依次为网卡接口、设备驱动层、数据链路层、IP层、传输层、最后到达应用程序。而包捕获机制是在数据链路层增加一个旁路处理，对发送和接收到的数据包做过滤/缓冲等相关处理，最后直接传递到应用程序。

网络数据包捕获技术

- 包捕获机制并不影响操作系统对数据包的网络栈处理。
- 对用户程序而言，包捕获机制提供了一个统一的接口，使用户程序只需要简单的调用若干函数就能获得所期望的数据包。
- 针对特定操作系统的捕获机制对用户透明，使用户程序有比较好的可移植性。包过滤机制是对所捕获到的数据包根据用户的要求进行筛选，最终只把满足过滤条件的数据包传递给用户程序。

网络数据包捕获技术

旁路处理机制

- 基于socket的网络的编程方法
- 数据链路提供者接口（DLPI）
- 伯克利数据包过滤器（BPF） Lipcap
- 零拷贝技术……

网络数据包捕获技术-程序设计

1）基于socket的网络编程已成为当今不可替代的编程方法。

- 这种编程思想将网络通讯当作“文件”描述字进行处理，对这个“网络文件”(即socket，套接字/套接口)的操作从编程者的角度来讲与普通的文件操作(如读、写、打开、关闭等)大同小异，从而极大地简化了网络程序开发过程。
- Linux内核版本2.0之前的一种套接字类型，可以接收网络上所有的数据包

网络数据包捕获技术

2）数据链路提供者接口（DLPI）

Data Link Provider Interface，定义了数据链路层向网络层提供的服务，是数据链路服务的提供者和使用者间的一种标准接口，在实现上基于UNIX的流机制。数据链路服务的使用者既可以是用户的应用程序，也可以是访问数据链路服务的高层协议，如TCP/IP等。

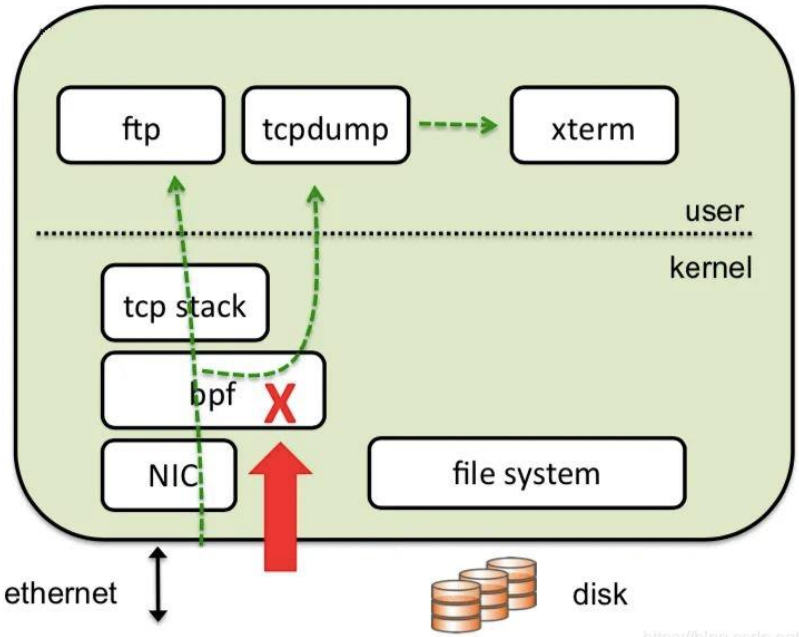
网络数据包捕获技术

3） 伯克利数据包过滤器（BPF）

- 最初构想提出于 1992 年，其目的是为了提供一种过滤包的方法，并且要避免从内核空间到用户空间的无用的数据包复制行为。最初是由从用户空间注入到内核的一个简单的字节码构成，它在那个位置利用一个校验器进行检查 —— 以避免内核崩溃或者安全问题 —— 并附着到一个套接字上，接着在每个接收到的包上运行。几年后它被移植到 Linux 上，并且应用于一小部分应用程序上（例如，tcpdump）。其简化的语言以及存在于内核中的即时编译器（JIT），使 BPF 成为一个性能卓越的工具。
- Berkeley Packet Filter,是一个高效的数据包捕获机制，工作在操作系统的内核层。
- BPF主要由网络转发部分和数据包过滤两部分组成。网络转发部分是从链路层捕获数据包并把它们转发给数据过滤部分，数据包过滤部分是从接收到的数据包中接收过滤规则决定的网络数据包，其他数据包被丢弃
- 在操作系统的内核中完成，效率很高。使用了数据缓存机制，使捕获数据包缓存在内核中，达到一定数量再传递给应用程序
- 实际应用中，使用libpcap实现

基于BPF技术的tcpdump的原理

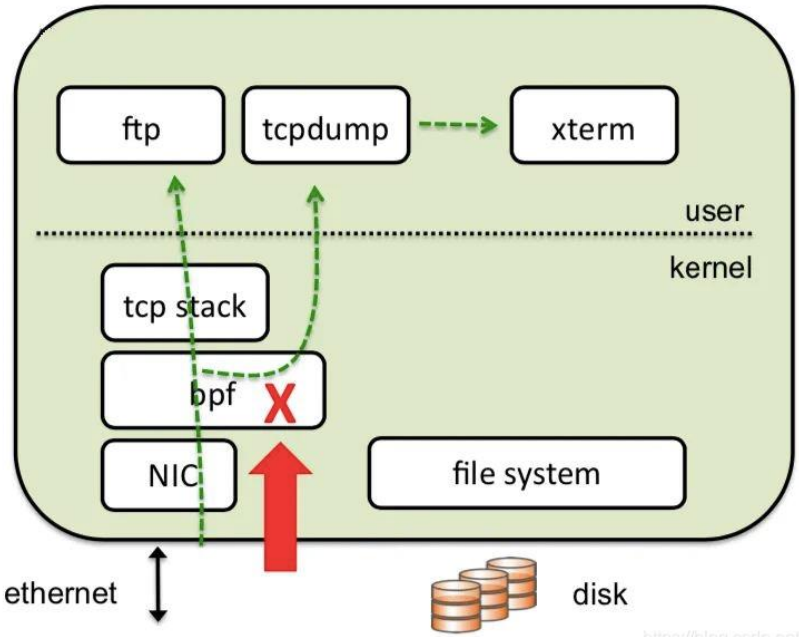
数据包到达网卡后，经过数据包过滤器（BPF）筛选后，拷贝至用户态的 tcpdump 程序，以供 tcpdump 工具进行后续的处理工作，输出或保存到 pcap 文件。数据包过滤器（BPF）主要作用，就是根据用户输入的过滤规则，只将用户关心的数据包拷贝至 tcpdump，这样能够减少不必要的数据包拷贝，降低抓包带来的性能损耗。



思考：如果某些数据包被 iptables 封禁，是否可以通过 tcpdump 抓到包？

基于BPF技术的tcpdump的原理

数据包到达网卡后，经过数据包过滤器（BPF）筛选后，拷贝至用户态的 tcpdump 程序，以供 tcpdump 工具进行后续的处理工作，输出或保存到 pcap 文件。数据包过滤器（BPF）主要作用，就是根据用户输入的过滤规则，只将用户关心的数据包拷贝至 tcpdump，这样能够减少不必要的数据包拷贝，降低抓包带来的性能损耗。



思考：如果某些数据包被 iptables 封禁，是否可以通过 tcpdump 抓到包？

通过上图，我们可以很轻易的回答此问题。因为 Linux 系统中netfilter是工作在协议栈阶段的，tcpdump 的过滤器（BPF）工作位置在协议栈之前，所以当然是可以抓到包了！

网络协议分析技术

■协议分析的过程主要包括三部分:

1） 捕获数据包

Libpcap ， Winpcap .....

2） 过滤数据包

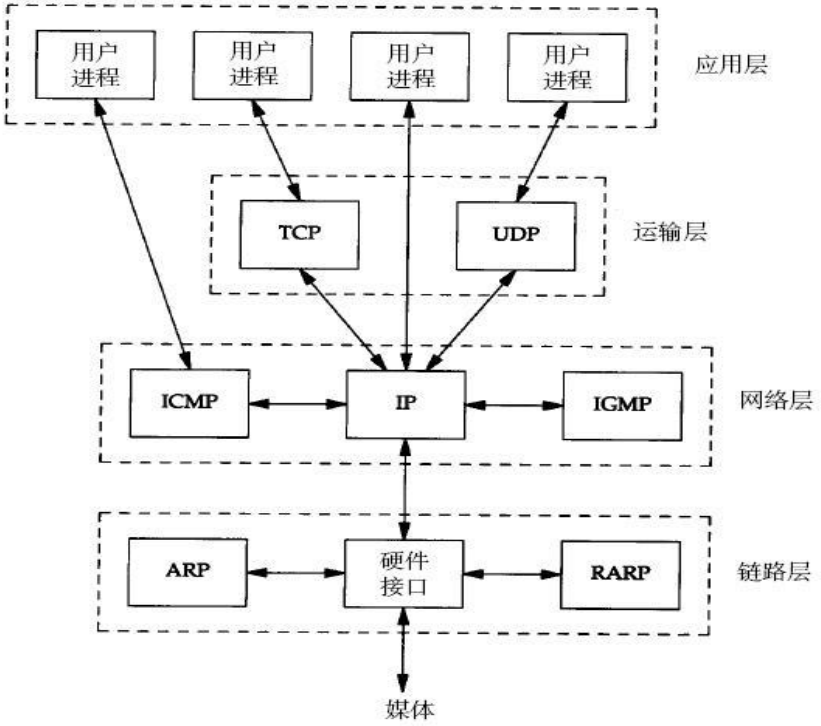
在内核层过滤：效率高 Libpcap

在应用层过滤：从内核层到应用层之间的转换费时费力

3）具体协议分析

链路层-〉网络层-〉传输层-〉应用层

Tcpdump，Windump，Ethereal.....



网络数据包生成技术

指人工构造数据包，然后把数据包发送到网络上

■应用：

- 网络安全扫描系统（构造数据包，探测远程主机，根据返回信息检查远程主机的漏洞）
- 网络安全测试系统（构造各种各样的数据包来检测防火墙、入侵检测系统的性能）
- 也是攻击者用的最多的一种技术（SynFlood、 ping of death……）

■技术代表： Libnet

常用的网络开发包简介

- 网络安全开发包是指用于网络安全研究和开发的一些专业开发函数库
- 在网络安全工具开发中，目前最为流行的C API library有libnet、libpcap、libnids和libicmp等。
- 它们分别从不同层次和角度提供了不同的功能函数。使网络开发人员能够忽略网络底层细节的实现，从而专注于程序本身具体功能的设计与开发。

- libnet 提供的接口函数主要实现和封装了数据包的构造和发送过程。
- libpcap 提供的接口函数主要实现和封装了与数据包截获有关的过程。
- libnids 提供的接口函数主要实现了开发网络入侵监测系统所必须的一些结构框架。
- libicmp 等相对较为简单，它封装的是ICMP数据包的主要处理过程(构造、发送、接收等)。

常用的网络开发包简介

一些在实际使用中常用的网络安全开发包：

- 网络数据包捕获开发包**Libpcap**
- **Windows**平台专业数据包捕获开发包**WinPcap**
- 网络数据包构造和发送开发包**Libnet**
- 网络入侵检测开发包**Libnids**
- 通用网络安全开发包**Libdnet**
- **ICMP**协议数据包处理开发包**Libicmp**

（1）libpcap

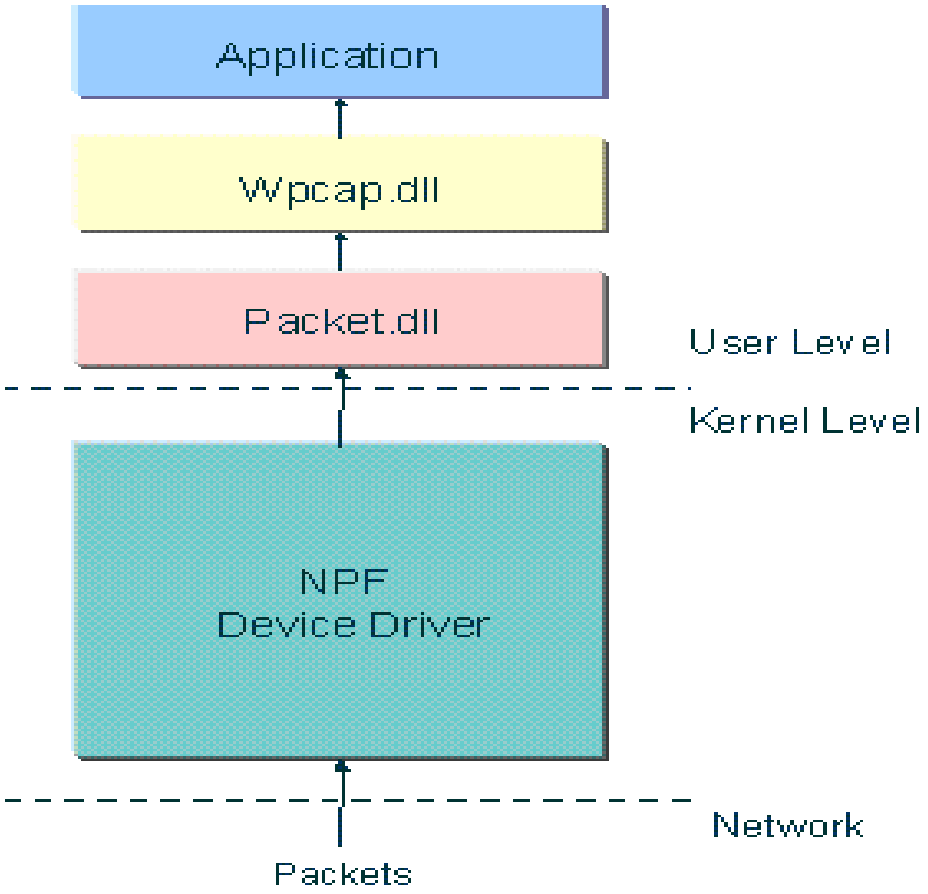
- libpcap的英文意思是 Packet Capture library，即数据包捕获函数库。
  - 该库提供的C函数接口可用于需要捕获经过网络接口（只要经过该接口，目标地址不一定为本机）数据包的系统开发上。
  - 由Berkeley大学Lawrence Berkeley National Laboratory研究院的Van Jacobson、Craig Leres和Steven McCanne编写。

Windows平台下的抓包技术

- 内核本身没有提供标准的接口
- 通过增加一个驱动程序或者网络组件来访问内核网卡驱动提供的数据包
  - 在Windows不同操作系统平台下有所不同
- 不同sniffer采用的技术不同
  - WinPcap是一个重要的抓包工具，它是libpcap的Windows版本

WinPcap

- WinPcap包括三个部分
  - 第一个模块NPF(Netgroup Packet Filter)，是一个虚拟设备驱动程序文件。它的功能是过滤数据包，并把这些数据包原封不动地传给用户态模块，这个过程中包括了一些操作系统特有的代码
  - 第二个模块packet.dll为win32平台提供了一个公共的接口。不同版本的Windows系统都有自己的内核模块和用户层模块。Packet.dll用于解决这些不同。调用Packet.dll的程序可以运行在不同版本的Windows平台上，而无需重新编译
  - 第三个模块 Wpcap.dll是不依赖于操作系统的。它提供了更加高层、抽象的函数。

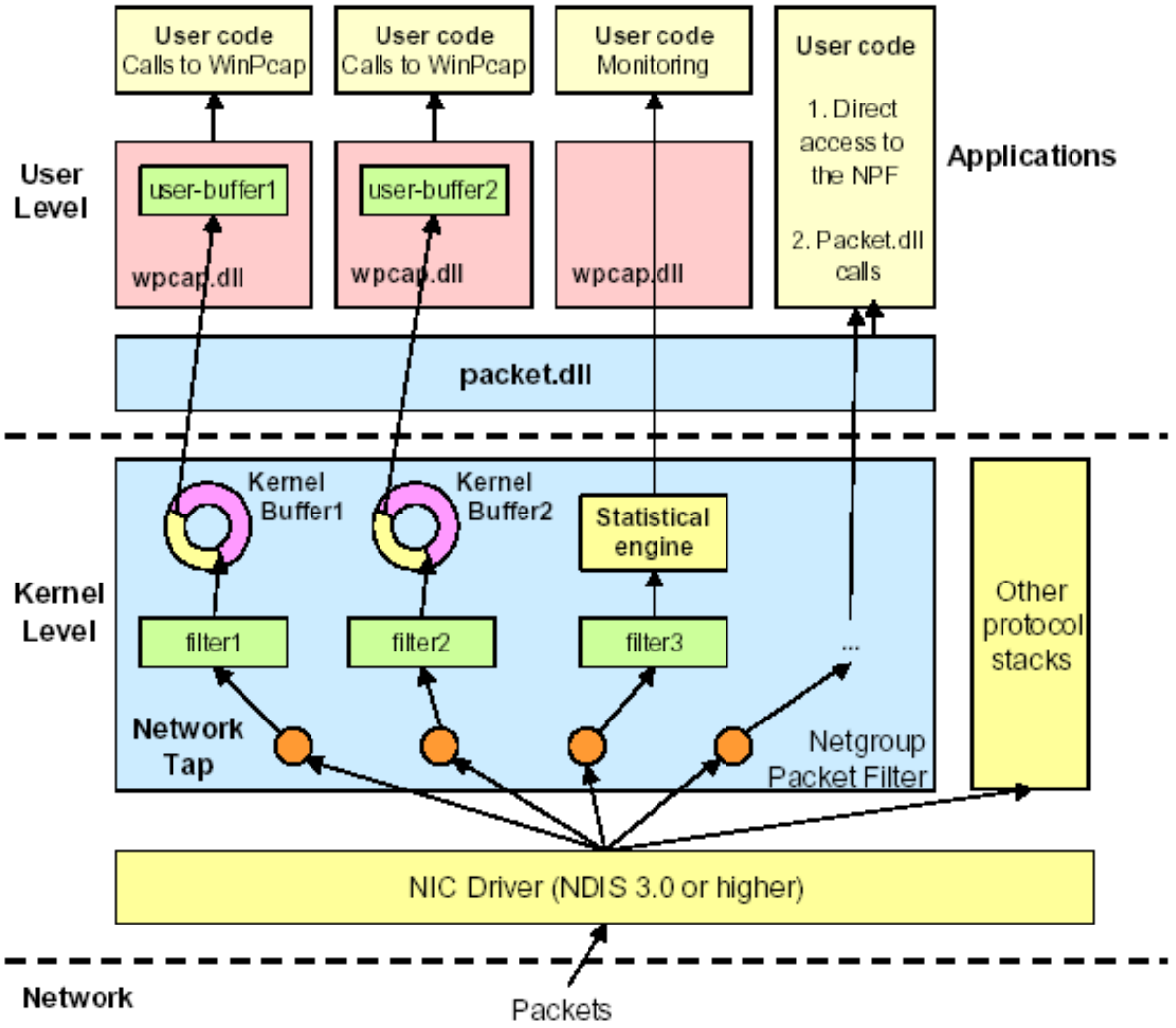


packet.dll直接映射了内核的调用

Wpcap.dll提供了更加友好、功能更加强大的函数调用

Winpcap的主要组成部分

WinPcap和NPF



WinPcap的优势

- 提供了一套标准的抓包接口
  - 与libpcap兼容，可使得原来许多UNIX平台下的网络分析工具快速移植过来
  - 便于开发各种网络分析工具
- 除了与libpcap兼容的功能之外，还有
  - 充分考虑了各种性能和效率的优化，包括对于NPF内核层次上的过滤器支持
  - 支持内核态的统计模式
  - 提供了发送数据包的能力

Winpcap安装

1.下载winpcap安装包 **https://www.winpcap.org/devel.htm**

2.安装后且对文件进行解压

3.在路径WpdPack_4_1_2\Include\pcap\pcap.h文件中加一句：#define WIN32.用来声明在Windows系统中使用

4.在setting中选择Compiler进行编辑器设置

5.在Linker settings和Search directories中添加路径Linker settings中需要添加lib文件，分别为wpcap.lib和Packed.lib.

还有在路径C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Lib下的WS2_32.Lib

Search directories中导入include文件和include下的pcap文件

Winpcap-Npcap

Riverbed Technology | Wireshark

WinPcap

WinPcap | WinDump | NTAR

WinPcap Has Ceased Development. We recommend Npcap.
The WinPcap project has ceased development and WinPcap and WinDump are no longer maintained. WE RECOMMEND USING Npcap INSTEAD.

If you do insist upon using WinPcap, be aware that its installer

- Uses NDIS 5.0, which might not work well with newer versions of Windows.
- Was built with an old version of NSIS and as a result is vulnerable to DLL hijacking.

The last official WinPcap release was 4.1.3
For the list of changes, refer to the changelog.

Version 4.1.3 Installer for Windows

Driver +DLLs

Supported platforms:
- NONE. WinPcap is completely unsupported, and might have compatibility issues with current versions of Windows.

Previously supported platforms:
- Windows NT4/2000
- Windows XP/2003/Vista/2008/Win7/2008R2/Win8 (x86 and x64)

MD5 Checksum: a11a2f0cfe6d0b4c50945989db6360cd
SHA1 Checksum: e2516fcd1573e70334c8f50bee5241cdfdf48a00

Docs | Download | Licensing | Windows 11 | WinPcap

Packet capture library for Windows

Npcap is the Nmap Project's packet capture (and sending) library for Microsoft Windows. It implements the open Pcap API using a custom Windows kernel driver alongside our Windows build of the excellent libpcap library. This allows Windows software to capture raw network traffic (including wireless networks, wired ethernet, localhost traffic, and many VPNs) using a simple, portable API. Npcap allows for sending raw packets as well. Mac and Linux systems already include the Pcap API, so Npcap allows popular software such as Nmap and Wireshark to run on all these platforms (and more) with a single codebase. Npcap began in 2013 as some improvements to the (now discontinued) WinPcap library, but has been largely rewritten since then with hundreds of releases improving Npcap's speed, portability, security, and efficiency. In particular, Npcap now offers:

- **Loopback Packet Capture and Injection**: Npcap is able to sniff loopback packets (transmissions between services on the same machine) by using the Windows Filtering Platform (WFP). After installation, Npcap supplies an interface named `NPF Loopback`, with the description "Adapter for loopback capture". Wireshark users can choose this adapter to capture all loopback traffic the same way as other non-loopback adapters. Packet injection works as well with the pcap_inject() function.
- **Support for all Current Windows Releases**: Npcap supports all versions of Windows and

WinPcap isn't supported on Windows 10

For 14 years, WinPcap was the standard libpcap package for Windows. But when Windows 10 was released without NDIS 5 support, **WinPcap failed to keep up**, leaving users wondering what to do. Fortunately, **the Nmap Project stepped up and created Npcap**, converting the original WinPcap code to the new NDIS 6 API, giving users a fast and completely compatible alternative to WinPcap for Windows 10.

Windows 10 1607 also introduced strict driver-signing requirements that WinPcap can't meet. **Npcap is fully compliant**, with its drivers tested and co-signed by Microsoft.

Npcap runs great on Windows 11. Npcap is under active development and continues to support the latest Windows networking features.

Downloading and Installing Npcap Free Edition

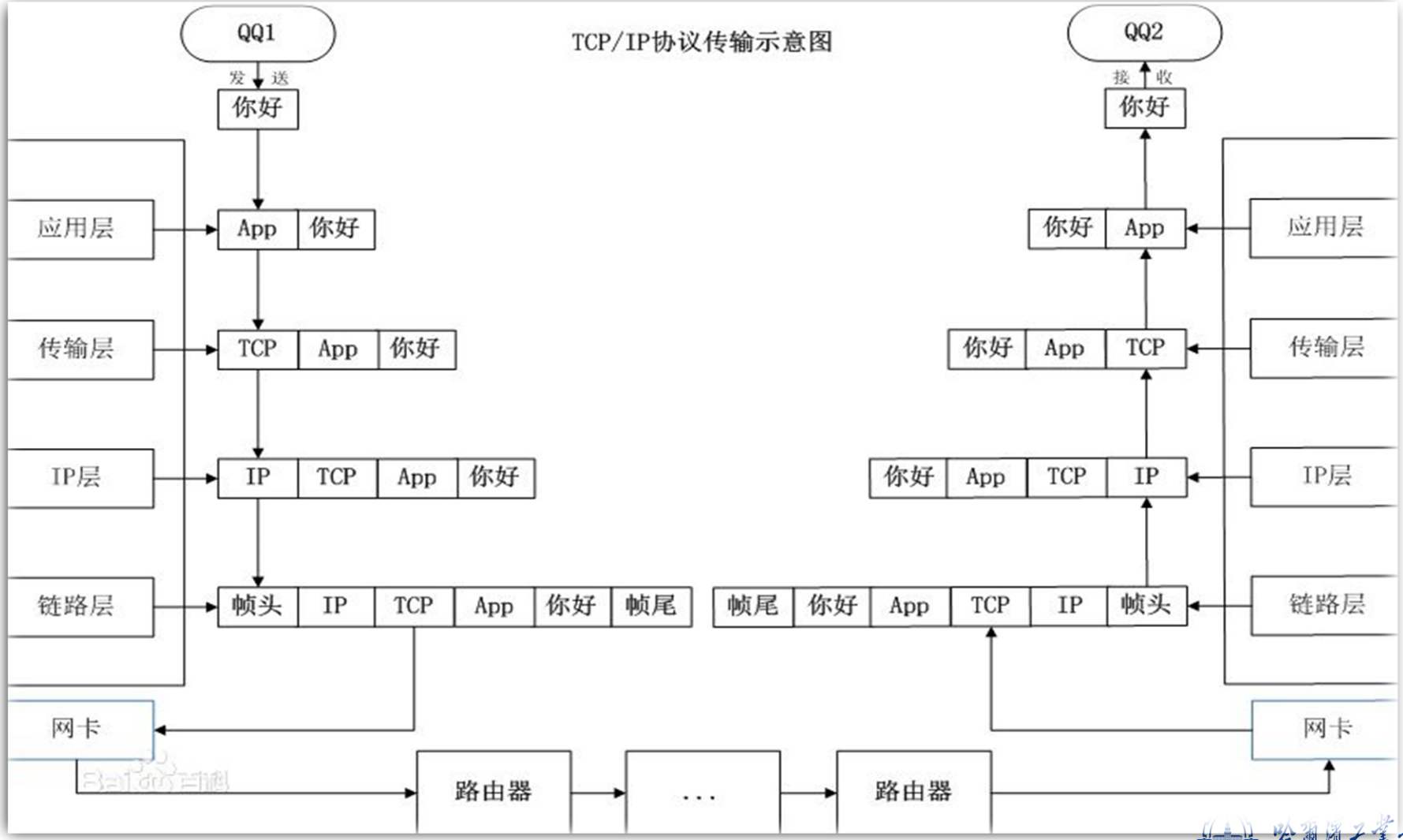
The free version of Npcap may be used (but not externally redistributed) on up to 5 systems (free license details). It may also be used on unlimited systems where it is only used with Nmap, Wireshark, and/or Microsoft Defender for Identity. Simply run the executable installer. The full source code for each release is available, and developers can build their apps against the SDK. The improvements for each release are documented in the Npcap Changelog.

- Npcap 1.79 installer for Windows 7/2008R2, 8/2012, 8.1/2012R2, 10/2016, 2019, 11 (x86, x64, and ARM64).
- Npcap SDK 1.13 (ZIP).
- Npcap 1.79 debug symbols (ZIP).
- Npcap 1.79 source code (ZIP).

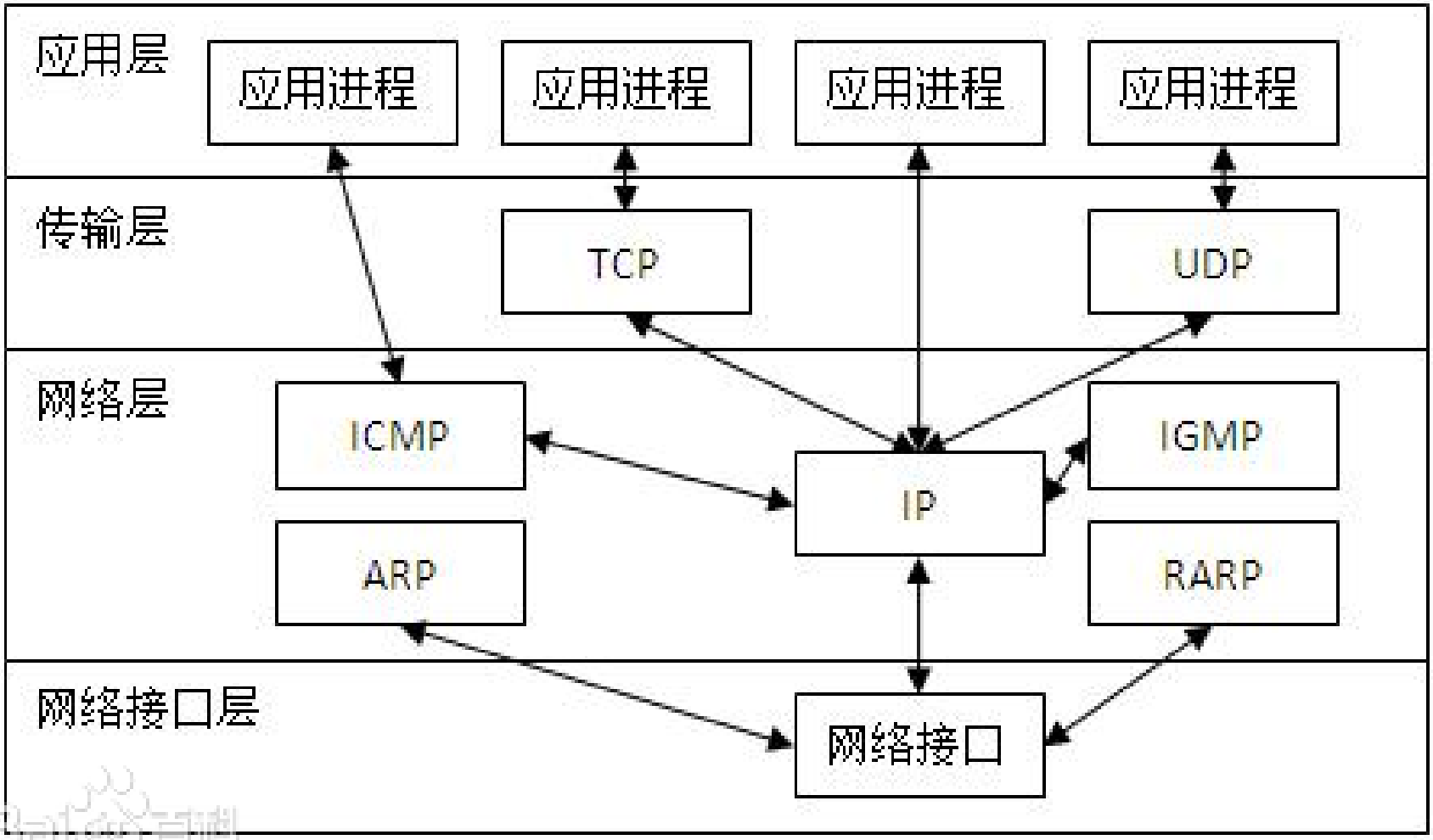
The latest development source is in our Github source repository. Windows XP and earlier are not supported; you can use WinPcap for these versions.

Npcap OEM for Commercial Use and Redistribution

TCP/IP协议

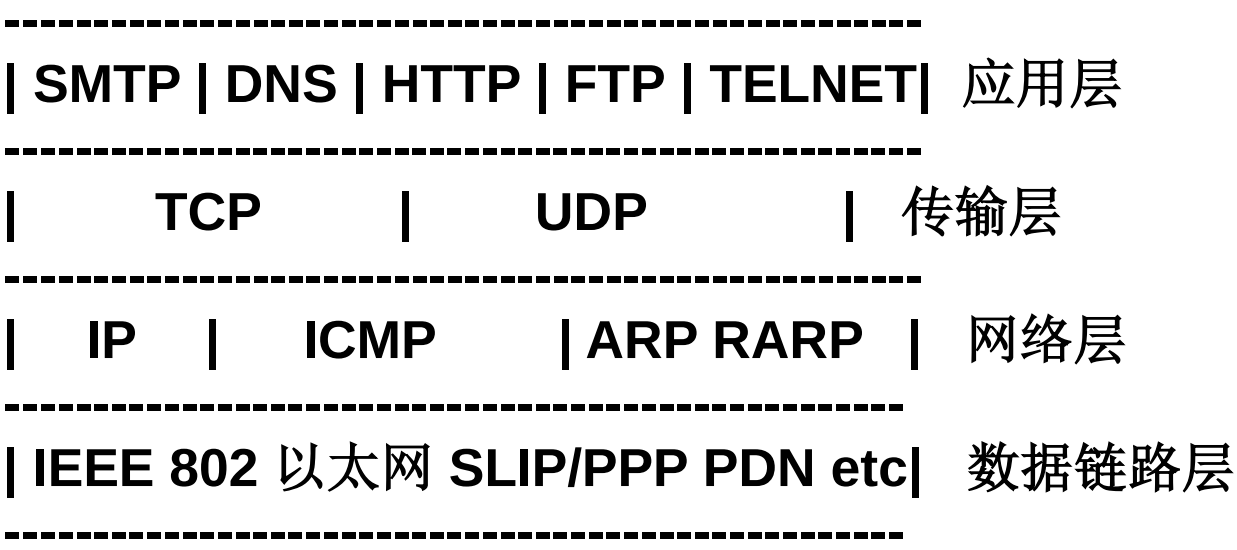

TCP/IP协议传输示意图
QQ1
发送
你好
应用层
App
你好
传输层
TCP
App
你好
IP层
IP
TCP
App
你好
链路层
帧头
IP
TCP
App
你好
帧尾
网卡
路由器
...
路由器
网卡
帧尾
你好
App
TCP
IP
帧头
链路层
IP层
传输层
应用层
你好
App
你好
App
TCP
你好
App
TCP
IP
QQ2
接收
你好

TCP/IP协议

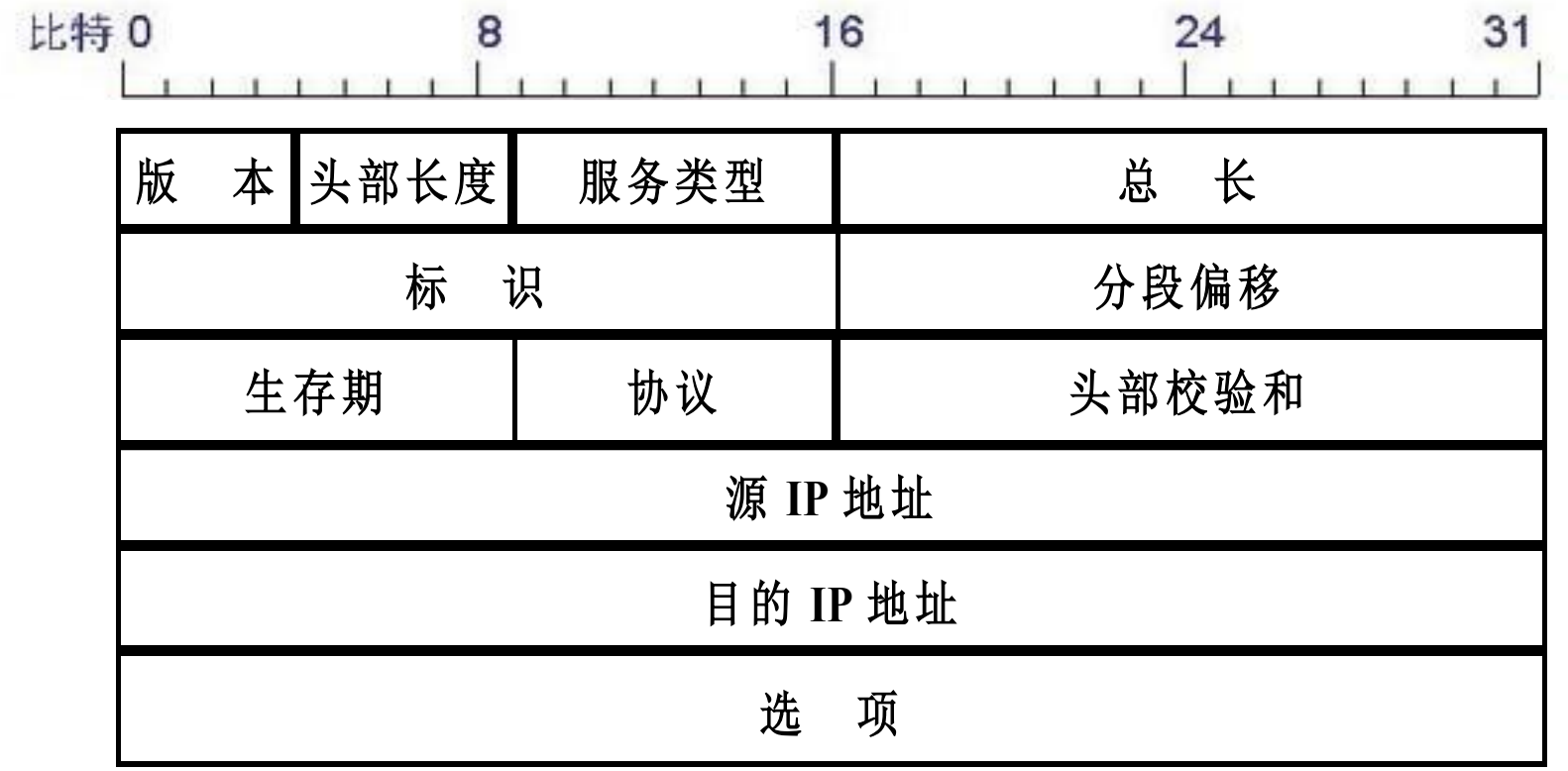

应用层
应用进程
应用进程
应用进程
应用进程
传输层
TCP
UDP
网络层
ICMP
IGMP
IP
ARP
RARP
网络接口层
网络接口

TCP/IP体系结构

- TCP/IP的网络体系结构



IP数据包首部格式


比特 0
8
16
24
31
版 本
头部长度
服务类型
总 长
标 识
分段偏移
生存期
协议
头部校验和
源 IP 地址
目的 IP 地址
选 项

IP数据包首部

```
/* IPv4 header */
struct ip_header
{
    #if defined(WORDS_BIGENDIAN)
        u_int8_t ip_version: 4,   /* 版本 */
        ip_header_length: 4;        /* 首部长度 */
    #else
        u_int8_t ip_header_length: 4, ip_version: 4;
    #endif
    u_int8_t ip_tos;       /* 服务质量 */
    u_int16_t ip_length;     /* 长度 */
    u_int16_t ip_id;      /* 标识 */
    u_int16_t ip_off;     /* 偏移 */
    u_int8_t ip_ttl;     /* 生存时间 */
    u_int8_t ip_protocol;      /* 协议类型 */
    u_int16_t ip_checksum;      /* 校验和 */
    struct in_addr ip_souce_address;     /* 源IP地址 */
    struct in_addr ip_destination_address;     /* 目的IP地址 */
};
```

UDP数据包首部格式

UDP 首部：

UDP报文格式如下图：

16　　31

源端口	目的端口
数据包长度	校验值
数据 DATA	

UDP数据包首部

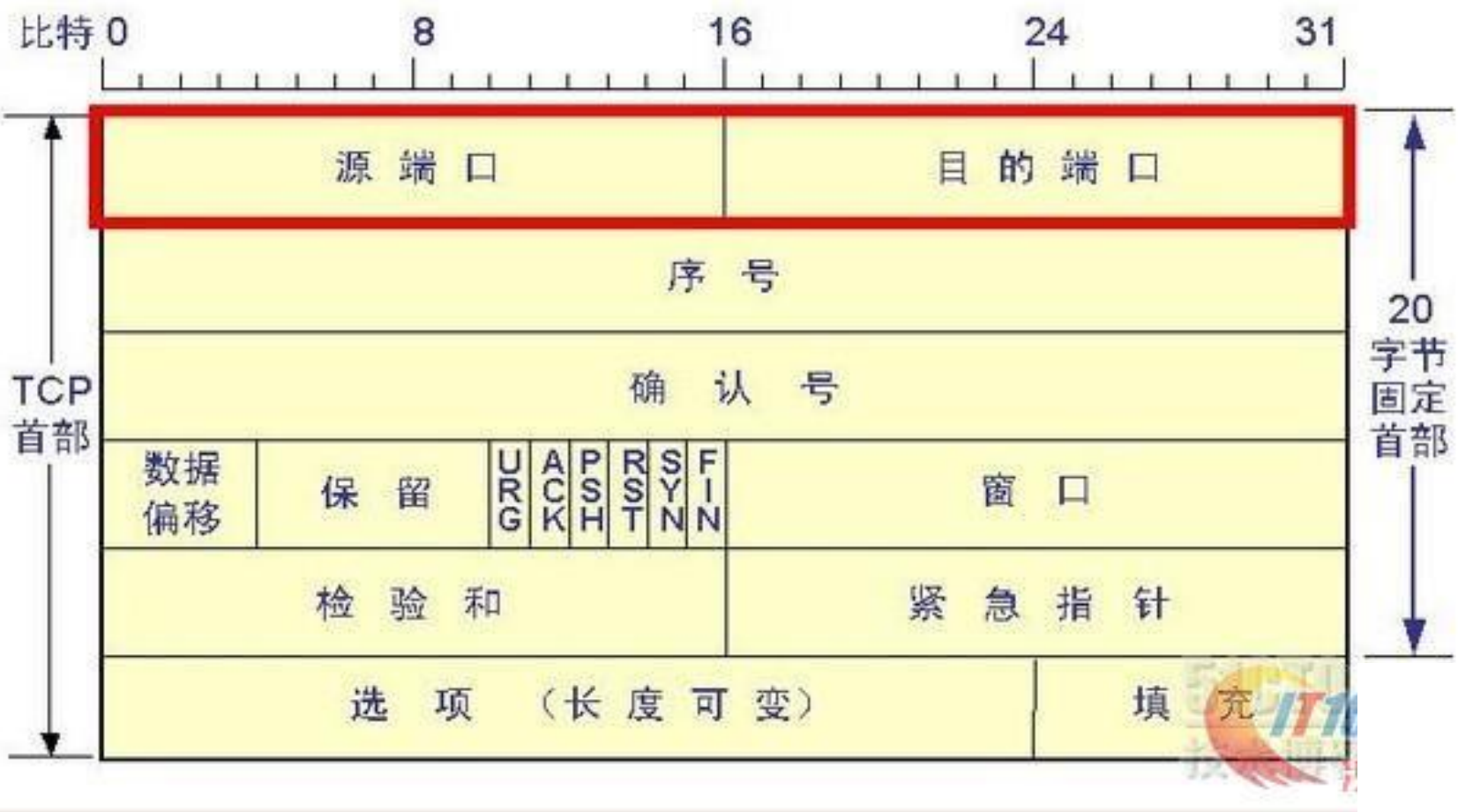
```
/* UDP header */
struct udp_header
{
    u_int16_t udp_source_port;    /* 源端口号 */
    u_int16_t udp_destination_port;    /* 目的端口号 */
    u_int16_t udp_length;    /* 长度 */
    u_int16_t udp_checksum;    /* 校验和 */
};
```

TCP 数据包首部

```
/* UDP header */
struct udp_header
{
    u_int16_t udp_source_port;    /* 源端口号 */
    u_int16_t udp_destination_port;    /* 目的端口号 */
    u_int16_t udp_length;    /* 长度 */
    u_int16_t udp_checksum;    /* 校验和 */
};
```

TDP数据包首部格式

- **TCP首部**



网络捕包程序基本流程

1、把网卡等同于文件进行I/O

- 查找网卡：**Find all devices()**
- 打开网卡：**open ()**

2、从网卡中读取数据

- 监听：**loop()**
- 数据回传给用户变量

3、处理获取的数据

- 转用户程序执行：**Handler()**

4、释放I/O资源

捕包程序示例 - 获取网卡信息

```
#include <pcap.h>// 必须引入的包
#pragma comment( lib, “wpcap.lib” )// 库文件
int main()
{
  pcap_if_t *alldevs;   //定义要获取的设备组
  pcap_if_t *d;        //定义单个设备组
  int i = 0;          //下面的for循环用
  char errbuf[PCAP_ERRBUF_SIZE];  //定义错误信息
  /* 获取本地所有网络适配器的列表 */
  if(pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) == -1)
  {
    fprintf(stderr, “Error in pcap_findalldevs_ex: %s\n”, errbuf);  // 获取失败，打印错误信息
    exit(1);
  }
  /* 循环打印适配器列表 */
  for(d = alldevs; d != NULL; d = d->next)
  {
    printf(“%d. %s”, ++i, d->name);  // 打印适配器的名字
    if (d->description)
      printf(“(%s)\n”, d->description);  // 打印适配器的描述信息
    else
      printf("(No description available)\n");
  }
  if(i == 0)
  {
    printf("\nNo interfaces found! Make sure Winpcap is installed.\n");  // 没有任何适配器列表
    return -1;
  }
  // 释放适配器列表指针的内存
  pcap_freealldevs(alldevs);
  // 防止命令窗口一闪而过
  char c = getchar();
  return 0;
}
```

Winpcap函数介绍

- **int pcap_findalldevs_ex** (char * source, struct pcap_rmtauth * auth, pcap_if_t ** **alldevs**, char * errbuf)
  - 这个函数是’pcap_findalldevs()’的一个超集。
  - pcap_findalldevs()比较老，只允许列出本地机器上的设备。pcap_findalldevs_ex() 除了可以列出本地及其上的设备，还可以列出远程机器上的设备。
  - 参数：sourse 指定需要监控的网络适配器。他有特定的伪语法：
    - file://folder/[列出指定文件夹中的所有文件]
    - rpcap://[列出所有本地的适配器]
    - rpcap://host:port/[列出远程主机上的可用的设备]

注意：port和host参数可以是数字形式也可以是字符形式

- host (字符):例如： host.foo.bar
- host (数字 IPv4): 例如： 10.11.12.13
- host (IPv6型的IPv4数字形式): 例如： [10.11.12.13]
- host (数字 IPv6): 例如： [1:2:3::4]
- port: 也可以是数字 (例如：'80') 或字符 (例如： 'http')

Winpcap函数介绍

- **int pcap_findalldevs_ex** (char * source, struct pcap_rmtauth * auth, pcap_if_t ** **alldevs**, char * errbuf)
  - 参数：auth 结构体定义如下：

```
1  struct pcap_rmtauth
2  {
3      int type;
4      char *username;
5      char *password;
6  };
```

type:简要身份验证所需的类型。

username:用户名

password:密码

  - auth参数可以为NULL.

Winpcap函数介绍

- **int pcap_findalldevs_ex** (char * source, struct pcap_rmtauth * auth, pcap_if_t ** alldevs, char * errbuf)
  - 参数：alldevs 该参数用于存放获取的适配器数据，如果查找失败，值为null。
  - Pcap_if_t 结构体定义：

```
1  struct pcap_if {
2      struct pcap_if *next;
3      char *name;         /* name to hand to "pcap_open_live()" */
4      char *description;  /* textual description of interface, or NULL */
5      struct pcap_addr *addresses;
6      bpf_u_int32 flags;  /* PCAP_IF_ interface flags */
7  };
```

char* name 指向字符串的指针 字符串用来向pcap_open_live()传递设备的名称

char* description 如果不为null，它指向的字符串是对设备的一个简单的描述。

pcap_addr* addresses 指向设备列表中第一个元素的地址；

捕包程序示例 - 获取网卡信息

```
#include <pcap.h>// 必须引入的包
#pragma comment( lib, “wpcap.lib” )// 库文件
int main()
{
  pcap_if_t *alldevs;   //定义要获取的设备组
  pcap_if_t *d;        //定义单个设备组
  int i = 0;          //下面的for循环用
  char errbuf[PCAP_ERRBUF_SIZE];  //定义错误信息
  /* 获取本地所有网络适配器的列表 */
  if(pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) == -1)
  {
    fprintf(stderr, “Error in pcap_findalldevs_ex: %s\n”, errbuf);   // 获取失败，打印错误信息
    exit(1);
  }
  /* 循环打印适配器列表 */
  for(d = alldevs; d != NULL; d = d->next)
  {
    printf(“%d. %s”, ++i, d->name);  // 打印适配器的名字
    if (d->description)
      printf(“(%s)\n”, d->description);  // 打印适配器的描述信息
    else
      printf("(No description available)\n");
  }
  if(i == 0)
  {
    printf("\nNo interfaces found! Make sure Winpcap is installed.\n");  // 没有任何适配器列表
    return -1;
  }
  // 释放适配器列表指针的内存
  pcap_freealldevs(alldevs);
  // 防止命令窗口一闪而过
  char c = getchar();
  return 0;
}
```

捕包程序示例 - 打开网络设备并且开始捕获数据包

```
int main()
{  ....
   int inum;
   pcap_t *adhandle;
   /* 获取本地所有网络设配器的列表 */
   pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) ;
  ....
  /* 选择一个适配器进行捕包 */
  printf("Enter the interface number (1-%d):",i);
  scanf_s("%d", &inum);
  if(inum < 1 || inum > i){
     printf("\nInterface number out of range.\n");
     pcap_freealldevs(alldevs);  //释放适配器
     return -1;
  }
  /* 找到要捕获的适配器指针 */
  for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);
  /* 打开设备 */
  if ( (adhandle= pcap_open(d->name,        // 设备名
                    65536,         // 单包捕获长度限制，65536为最大值
                    PCAP_OPENFLAG_PROMISCUOUS,   // 打开网卡混杂模式
                    1000,          // 超时时间
                    NULL,          // 远程设备验证信息
                    errbuf         // 出错信息 ) ) == NULL)
  {  fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n", d->name);
     pcap_freealldevs(alldevs); /* Free the device list */
     return -1;
  }
  printf("\nlistening on %s...\n", d->description);
  pcap_freealldevs(alldevs); /* 已经打开要捕包的网卡，释放设备列表指针 */
  /* 开始捕包 */
  pcap_loop(adhandle, 0, packet_handler, NULL); //连续捕包，数据包调用回调函数处理
}
```

Winpcap：捕获数据

pcap_t *pcap_open (const char * source, int snaplen, int flags, int read_timeout, struct pcap_rmtauth * auth, char * errbuf)

- 为捕获/发送数据打开一个普通的源;
- 返回值 pcap_t :一个已打开的捕捉实例的描述符。
- 参数 source：包含要打开的源名称的字符串。
- read_timeout：以毫秒单位。read timeout用来设置在遇到一个数据包的时候读操作不必立即返回，而是等待一段时间，让更多的数据包到来后从OS内核一次读多个数据包。并非所有的平台都支持read timeout；在不支持read timeout的平台上将被忽略。 即使没有来自网络的数据包，适配器上的读取将在read_timeout 毫秒之后返回。如果适配器处于统计模式，to_ms还会定义统计报告之间的间隔。
- snaplen：需要保留的数据包的长度。对每一个过滤器接收到的数据包，第一个‘snaplen’字节的内容将被保存到缓冲区，并且传递给用户程序。例如，snaplen等于100，那么仅仅每一个数据包的第一个100字节的内容被保存。简言之就是从每一个包的开头到snaplen的那段内容将被保存。 某些操作系统（如xBSD和Win32）上，数据包驱动程序可以配置为仅捕获任何数据包的初始部分：这会减少要复制到应用程序的数据量，从而提高捕获效率。在这种情况下，可以设置成65536，高于我们可能遇到的最大MTU值。以这种方式，我们确保应用程序将始终收到整个数据包。

Winpcap：捕获数据

pcap_t *pcap_open (const char * source, int snaplen, int flags, int read_timeout, struct pcap_rmtauth * auth, char * errbuf)

- flags：保存一些由于抓包需要的标志。Winpcap定义了三种标志：

最重要的是**PCAP_OPENFLAG_PROMISCUOUS**，表示这个网络设备以混杂模式打开。 在正常操作中，适配器仅捕获来自网络的分发给它的分组; 因此，其他主机交换的数据包被忽略。当适配器处于混杂模式时，它将捕获收到的所有数据包。这意味着在共享介质（如非交换式以太网）上，WinPcap将能够捕获其他主机的数据包。混杂模式是大多数捕获应用程序的默认模式。

- auth：一个指向’**struct pcap_rmtauth**’的指针，保存当一个用户登录到某个远程机器上时的必要信息。假如不是远程抓包，该指针被设置为**NULL**。
- errbuf：一个指向用户申请的缓冲区的指针，存放当该函数出错时的错误信息。

Winpcap：捕获数据

- 捕获数据用到的两个函数

  int pcap_dispatch (pcap_t *p, int cnt, pcap_handler callback, u_char *user)

  int pcap_loop (pcap_t *p, int cnt, pcap_handler callback, u_char *user)

  - 参数含义：
    - p就是我们打开的某个网络设备的描述符。 **pcap_open返回的值。**
    - cnt是count，表示这个循环会总共处理多少个数据包。比如30，就是处理30个数据包。0或负数表示没有限定，一直捕获下去。
    - Callback 回调函数是一个函数指针，用来具体操作如何对数据包进行处理。
    - user是用户信息。一般为NULL。
  - 二者不同之处在于处理实时数据时超时loop不返回，阻塞，而pcap_dispatch会返回。返回处理的数据数量
  - 两个函数都有一个回调参数packet_handler，指向一个将接收数据包的函数。该函数由libpcap为来自网络的每个新数据包调用。请注意，使用pcap_loop（）可能存在缺点，主要涉及到数据包捕获驱动程序调用处理程序的事实; 因此用户应用程序没有直接的控制权。另一种方法（并且具有更多可读程序）是使用pcap_next_ex（）函数。

  int pcap_next_ex (pcap_t* p, struct pcap_pkthdr** pkt_header, const u_char* pkt_data)

  - 从interface或离线记录文件获取一个报文
  - 用指向头和下一个被捕获的数据包的指针为pkt_header和pkt_data参数赋值。

捕包程序示例 - 打开网络设备并且开始捕获数据包

```
int main()
{  ....
   ....
   pcap_loop(adhandle, 0, packet_handler, NULL); //连续捕包，数据包调用回调函数处理
}

 /* 回调函数，每捕获一个满足要求的报文就调用 */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
   struct tm ltime;
   char timestr[16];
   time_t local_tv_sec;

   /*
    * unused variables
    */
   (VOID)(param);
   (VOID)(pkt_data);

   /* convert the timestamp to readable format */
   local_tv_sec = header->ts.tv_sec;
   localtime_s(&ltime, &local_tv_sec);
   strftime( timestr, sizeof timestr, "%H:%M:%S", &ltime);

   printf("%s,%.6d len:%d\n", timestr, header->ts.tv_usec, header->len);

}
```

Winpcap：捕获数据

hypedef void (* pcap_handler)(u_char* user, const struct pcap_pkthdr* pkt_header,const u_char* pkt_data)

- 接收数据包的回调函数原型。当用户程序使用**cap_dispatch()**或者**pcap_loop()**，数据包以这种回调的方法传给应用程序。
- 用户参数是用户自己定义的包含捕获会话状态的参数，它必须跟**pcap_dispatch()**和**pcap_loop()**的参数相一致。**pkt_header**是与抓包驱动有关的头。**pkt_data**指向包里的数据，包括协议头。

一次抓包返回的数据说明

```
struct pcap_pkthdr {
        struct timeval ts;
        bpf_u_int32 caplen;
        bpf_u_int32 len;
}
```

ts：时间戳

cpalen：当前返回数据的长度

len：总长度

Winpcap：设置filter

- 设置过滤器用到的函数：

通常我们只对特定网络通信感兴趣。比如我们只打算监听Telnet服务（port 23）以捕获用户名和口令信息。获知对FTP（port 21）或DNS（UDP port 53）数据流感兴趣。可以通过pcap_compile()和pcap_setfilter来设置数据流过滤规则（filter）

int pcap_compile (pcap_t *p, struct **bpf_program *fp,** char *str, int optimize, bpf_u_int32 netmask)

把字符串str编译成一个过滤器程序

- **int pcap_setfilter** (pcap_t *p, **struct bpf_program *fp**)

设置一个过滤器

过滤规则举例：（和tcpdump的过滤参数一致）

- 只想查目标机器端口是21或80的网络包，其他端口的我不关注：dst port 21 or dst port 80
- 主机172.16.0.11 和主机210.45.123.249或 210.45.123.248的通信，使用命令(注意括号的使用)：host 172.16.0.11 and (210.45.123.249 or210.45.123.248)
- 使用ftp端口和ftp数据端口的网络包 port ftp or ftp-data
  这里 ftp、ftp-data到底对应哪个端口？ linux系统下 /etc/services这个文件里面，就存储着所有知名服务和传输层端口的对应关系。如果你直接把/etc/services里的ftp对应的端口值从21改为了3333，那么就会去抓端口含有3333的网络包了。
- 如果想要获取主机172.16.0.11除了和主机210.45.123.249之外所有主机通信的ip包，使用命令：host 172.16.0.11 and ! 210.45.123.249
- 抓172.16.0.11的80端口和110和25以外的其他端口的包：host 172.16.0.11 and! port 80 and ! port 25 and ! port 110

Libpcap/Winpcap程序编写

Libpcap工作机制- 程序框架

/* 第一步：查找可以捕获数据包的设备 */

- **int pcap_findalldevs_ex (char * source, struct pcap_rmtauth * auth, pcap_if_t ** alldevs, char * errbuf)**

/* 第二步：创建捕获句柄，准备进行捕获 */

- **pcap_t *pcap_open (const char * source, int snaplen, int flags, int read_timeout, struct pcap_rmtauth * auth, char * errbuf)**

重要： flags **混杂模式**

/* 第三步：如果用户设置了过滤条件，则编译和安装过滤代码 */

- **int pcap_compile (pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)**
- **int pcap_setfilter (pcap_t *p, struct bpf_program *fp)**

/* 第四步：循环捕获数据包-传递给回调函数 */

- **int pcap_dispatch (pcap_t *p, int cnt, pcap_handler callback, u_char *user)**
- **int pcap_loop (pcap_t *p, int cnt, pcap_handler callback, u_char *user)**

重要：cnt的值是循环捕多少个包，如果是**0** 或者**负数**表示不停的循环捕包

/* 第五步：编写回调函数，按需求逻辑分析处理数据包*/

- **typedef void (* pcap_handler)(u_char* user, const struct pcap_pkthdr* pkt_header,const u_char* pkt_data)**

60

Winpcap：捕获数据

处理离线的存储文件（**offline dump file**）

Winpcap提供了一些函数把网络通信保存到文件并且可以读取这些文件的内容。**dump**文件的格式跟**libpcap**是一样的。它以二进制形式保存了被捕获的数据包的数据并且其他网络工具（包括**WinDump**，**Ethereal**，**Snort**）也以此为标准。

处理**dump**文件的程序结构跟前面的程序结构大致是一样的

pcap_dumper_t* pcap_dump_open (pcap_t*p, const char* fname)

打开一个保存数据包的文件。

void pcap_dump (u_char* user, const struct pcap_pkthdr* h, const u_char* sp)

把数据包保存到硬盘。

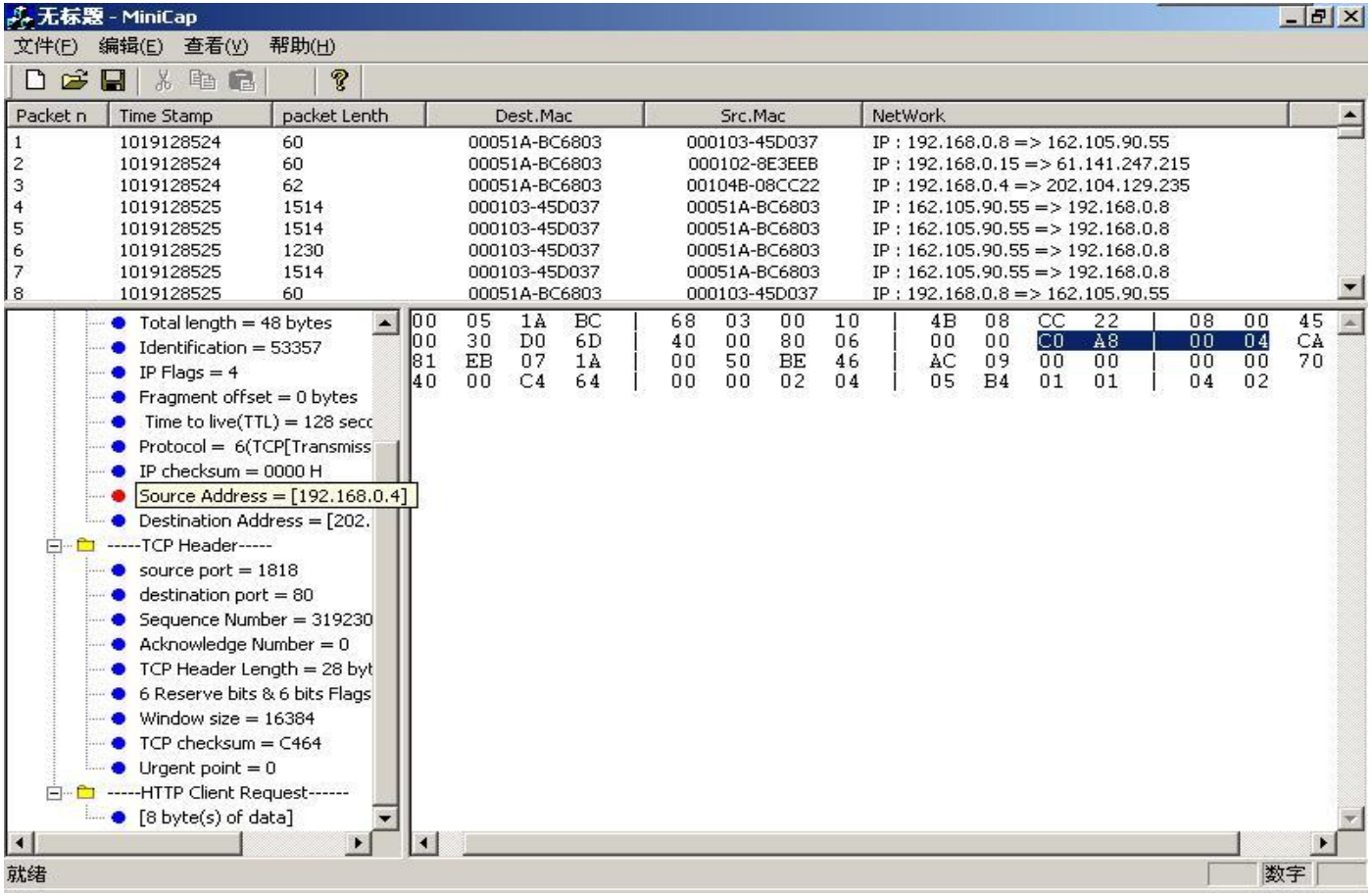
int pcap_live_dump (pcap_t* p, char* filename, int maxsize, int maxpacks)

保存数据包到文件。

pcap_t* pcap_open_offline (const char* fname, char* errbuf)

为读数据包打开一个**tcpdump/libpcap**格式的文件。

用WinPcap开发自己的sniffer



Winpcap函数介绍

int pcap_sendpacket (pcap_t* p, u_char* buf, int size)

发送一个原始数据包（**raw packet**）到网络上。

p是用来发送数据包的那个接口，

buf包含着要发送的数据包的数据（包括各种各样的协议头），

size是**buf**所指的缓冲区的尺寸，也就是要发送的数据包的大小。**MAC**循环冗余码校验不必被包含，因为它很容易被计算出来并被网络接口驱动添加。如果数据包被成功发送，返回**0**；否则，返回－**1**。

发送队列（Send queues）

pcap_sendpacket()提供了一个简单快捷的发送单个数据包的方法，发送队列（send queues）提供了一个高级的，强大的，优化的发送一组数据包的机制。发送队列是一个用来保存将要发送到网络上的的众多数据包的容器。它有一个大小，描述了它所能容纳的最大字节数。

通过指定发送队列的大小，pcap_sendqueue_alloc()函数创建一个发送队列。一旦发送队列被创建好，pcap_ sendqueue_queue()可以把一个数据包添加到发送队列里。然后，pcap_ sendqueue_transmit()提交到网络上. 最后，用pcap_sendqueue_ destroy() 释放资源。

函数pcap_sendqueue_alloc()的参数必须与pcap_ next_ex()和pcap_handler()的相同，因此，从一个文件捕获或读取数据包的时候，如何进行pcap_sendqueue_alloc()的参数传递是一个问题。

pcap_send_queue* pcap_sendqueue_alloc(u_int memsize)

为一个发送队列分配空间，即创建一个用来存储一组原始数据包（**raw packet**）的缓冲区，这些数据包将用**pcap_sendqueue_transmit()**提交到网络上。

- **memsize**是队列容纳的字节数，因此它决定了队列所能容纳的最大数据量。
- 使用**pcap_sendqueue_queue()**可以在发送队列中插入数据包。

int pcap_sendqueue_queue(pcap_send_queue* queue，const struct pcap_pkthdr* pkt_header, const u_char* pkt_data)

添加一个数据包到发送队列中。

- **queue**指向发送队列的尾部；
- **pkt_header**指向一个**pcap_pkthdr**结构体，该结构体包含时间戳和数据包的长度；
- **pkt_data**指向存放数据包数据部分的缓冲区。

- 为了提交一个发送队列，**Winpcap**提供了**pcap_sendqueue_transmit()**函数。

u_int pcap_sendqueue_transmit(pcap_t* p, pcap_send_queue* queue, int sync)

该函数将队列里的内容提交到线路上。

- p是一个指向适配器的指针，数据包将在这个适配器上被发送；
- queue指向pcap_ send_queue结构体，它包含着要发送的所有数据包；
- sync决定了发送操作是否被同步：如果它是非0（non-zero），发送数据包关系到时间戳，否则，他们将以最快的速度发送（即不考虑时间戳）。
- 返回值是发送的字节数。如果它小于size参数，将发生一个错误。该错误可能是由于驱动/适配器（driver/adapter）问题或发送队列的不一致/伪造（inconsistent/bogus）引起。

单包发送队列发送的区别：

1 使用该函数的效率比使用**pcap_sendpacket()**发送一系列数据包的效率高，因为数据包在核心态（**kernel-level**）被缓冲，所以降低了上下文的交换次数。因此，使用**pcap_sendqueue_transmit()**更好。

2 当**sync**被设置为**TRUE**时，随着一个高精度的时间戳，数据包将在内核被同步。这就要求**CPU**的数量是不可忽略的，通常允许以一个微秒级的精度发送数据包（这依赖于机器性能计数器的准确度）。然而，用**pcap_sendpacket()**发送数据包不能达到这样一个精确度。

如果第三个参数非0，发送将被同步（synchronized），即相关的时间戳将被注意。这个操作要求注意CPU的数量，因为使用“繁忙 等待（busy wait）”循环，同步发生在内核驱动。

当不再需要一个队列时，可以用pcap_sendqueue_destroy()来删除之，这将释放与该发送队列相关的所有缓冲区。

Winpcap：举例3

简单的发送一个数据包的过程.打开适配器后，**pcap_sendpacket()**被用来发送一个手工的数据包。

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
#include <remote-ext.h>
void main(int argc, char** argv){
 pcap_t* fp;
 char errbuf[PCAP_ERRBUF_SIZE];
 u_char packet[100];
 int i;

 /* Open the output device */
 if ((fp = pcap_open(argv[1],                  /* name of the device */
     100,                                      /* portion of the packet to capture (only the first 100 bytes)*/
     PCAP_OPENFLAG_PROMISCUOUS,                /* promiscuous mode */
     1000,                                     /* read timeout */
     NULL,                                     /* authentication on the remote machine */
     errbuf                                    /* error buffer */
     )) == NULL)
 {
  fprintf(stderr, "\nUnable to open the adapter. %s is not supported by Winpcap\n", argv[1]);
  return;
 }

/* Supposing to be on ethernet, set mac destinat to 1:1:1:1:1:1 */
 packet[0] = 1;     packet[1] = 1;   packet[2] = 1;   packet[3] = 1;   packet[4] = 1;   packet[5] = 1;

 /* set mac source to 2:2:2:2:2:2 */
 packet[6] = 2;   packet[7] = 2;  packet[8] = 2;  packet[9] = 2;  packet[10] = 2;  packet[11] = 2;
 /* Fill the rest of the packet */
 for (i = 12; i < 100; ++ i)
 {  packet[i] = i % 256;}
 /* Send down the packet */
 if (pcap_sendpacket(fp, packet, 100 /* size */) != 0)
 {
  fprintf(stderr, "\nError sending the packet: \n", pcap_geterr(fp));
  return;
 }
 return;
}
```

（2） libnet

- libnet库的最新版本为1.0.0，
  - 它一共约7600行C源代码，33个源程序文件，12个C头文件，50余个自定义函数。
  - 提供的接口函数包含15种数据包生成器和两种数据包发送器(IP层和数据链路层)。
  - 目前只支持IPv4，不支持IPv6。

- libnet提供的接口函数按其作用可分为四类：
  - 内存管理(分配和释放)函数
  - 地址解析函数
  - 数据包构造函数
  - 数据包发送函数

例如：构造并发送一个TCP包为例：

初始化（ libnet_init( ) ）

↓

构造TCP选项（ libnet_build_tcp_options( ) ）

↓

构造TCP头（ libnet_build_tcp ( ) ）

↓

构造IP头（ libnet_build_ipv4 ( ) ）

↓

构造以太网头（ libnet_build_ethernet( ) ）

↓

发送数据包（ libnet_write( ) ）

↓

销毁（ libnet_destroy( ) ）

主要数据结构

```
Struct libnet_stats
{
   u_int64_t packets_sent;       /* 发送的数据包 */
   u_int64_t packets_errors;     /* 出错的数据包 */
   u_int64_t packets_written;    /* 已经写的字节数目 */
}

Typedef int32_t libnet_ptag_t;
```

```
Struct libnet_protocol_block
{
    u_int8_t *buf;          /* 协议的缓冲区 */
    u_int8_t b_len;         /* 缓冲区的长度 */
    u_int8_t h_len;         /* 首部长度 */
    u_int8_t ip_offset;     /* IP首部的偏移量 */
    u_int8_t copied;        /* 拷贝的字节数目 */
    u_int8_t type;          /* 协议块的类型*/
    u_int8_t flags;         /* 控制标记*/
    libnet_ptag_t ptag;     /* 协议快标记*/
    struct libnet_protocol_block *next;   /* 下一个pblock*/
    struct libnet_protocol_block *prev;   /* 前一个pblock */
}
```

```
Struct libnet_context
{
   int fd;                          /* 数据包设备的文件描述符 */
   int injection_type;              /* 类型 */
   libnet_pblock_t *protocol_block; /* 指向第一个协议块结点 */
   libnet_pblock_t *pblock_end;     /* 指向最后一个协议块结点 */
   u_int32_t n_pblocks;             /* 协议块的数目 */
   int link_type;                   /* 链路层类型 */
   int link_offset;                 /* 链路首部偏移*/
   int aligner;                     /* 用来排列数据包 */
```

```
    char *device;                   /* 设备名称 */
    struct libnet_stats stats;      /* 统计分析 */
    libnet_ptag_t ptag_state;       /* pblock标记状态 */
    char label[LIBNET_LABEL_SIZE];  /* 队列文本标记 */
    char err_buf[LIBNET_ERRBUF_SIZE]; /* 错误信息 */
    u_int32_t total_size;           /* 整体大小 */
}
typedef struct libnet_context libnet_t;
```

```
typedef struct _libnet_context_queue libnet_cq_t;
struct _libnet_context_queue
{
   libnet_t *context;
   libnet_cq_t *next;
   libnet_cq_t *prev;
};
struct _libnet_context_queue_descriptor
{
  u_int32_t node;
  u_int32_t cq_lock;
  libnet_cq_t *current;
};
typedef struct _libnet_context_queue_descriptor
  libnet_cqt_t;
```

主要函数

- 内存管理函数

  核心函数主要包括Libnet初始化和销毁函数以及一些必须的设置函数。

  - **Libnet初始化函数主要完成了对Libnet的初始化功能，其中包括内存分配、网络接口设置和Libnet的类型设置等等。**
  - **内存分配主要实现的是在开始构造数据包之前分配足够的内存空间，在不需要内存的时候在销毁内存空间。**

主要函数

(1) libnet_t *libnet_init(int injection_type, char *device, char *err_buf)

函数返回值：如果成功，返回一个libnet句柄；如果失 败，返回NULL。

例如： l = libnet_init(LIBNET_RAW4,NULL,err)

(2) Void libnet_destroy(libnet_t *l)

函数返回值无，参数l是一个Libnet句柄指针。

主要函数

- 地址解析函数

主要对IP地址进行操作的一些函数

（1） u_int32_t libnet_name2addr4(libnet_t *l, char *host_name, u_int8_t use_name)

函数返回值：返回网络字节顺序的IPv4地址。

参数描述：参数l表示Libnet句柄；参数host_name表示主机的名字；参数use_name就进行域名解析。

数据包构造函数

数据包构造函数完成了所有的协议数据包构造的功能。由于Libnet可以构造很多协议格式的数据包，每种协议的数据包部分都用某一个函数来实现。

例如：

libnet_ptag_t libnet_build_ipv4(u_int16_t len, u_int8_t tos, u_int16_t id, u_int16_t flag, u_int8_t ttl, u_int8_t prot, u_int16_t sum,u_int32_t src, u_int32_t dst, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

此函数构造了一个IPv4协议头。

一个完整的数据包通常由几部分协议构造函数来构造，其构造的顺序不能颠倒，必须按照从高到低的顺序进行。先构造应用层协议数据，再构造传输层协议数据，再构造网络层协议数据，最后构造链路层协议数据。

例如：要构造一个DNS数据包，必须按照下面的调用顺利：

libnet_build_dnsv4( )

libnet_build_udp( )

libnet_build_ipv4( )

libnet_build_ethernet( )

数据包发送函数

发送数据包的函数只有一个，所以构造的任何协议数据包都使用同一个数据包发送函数。

int libnet_write(libnet_t *l);

函数返回值：返回一个整型数值。

参数描述：参数l表示libnet句柄。

此函数发送一个数据包，此数据包由libnet句柄l来指示。

- 高级处理函数
- Libnet句柄队列操作函数
- 辅助函数

一个完整的Libnet程序主要有一下几个步骤构成：

步骤一：对内存进行初始化；

步骤二：对网络进行初始化；

步骤三：构造各种网络数据包；

步骤四：对数据包进行合法性检验；

步骤五：发送网络数据包；

例如：构造并发送一个TCP包为例：

初始化（ libnet_init( ) ）

↓

构造TCP选项（ libnet_build_tcp_options( ) ）

↓

构造TCP头（ libnet_build_tcp ( ) ）

↓

构造IP头（ libnet_build_ipv4 ( ) ）

↓

构造以太网头（ libnet_build_ethernet( ) ）

↓

发送数据包（ libnet_write( ) ）

↓

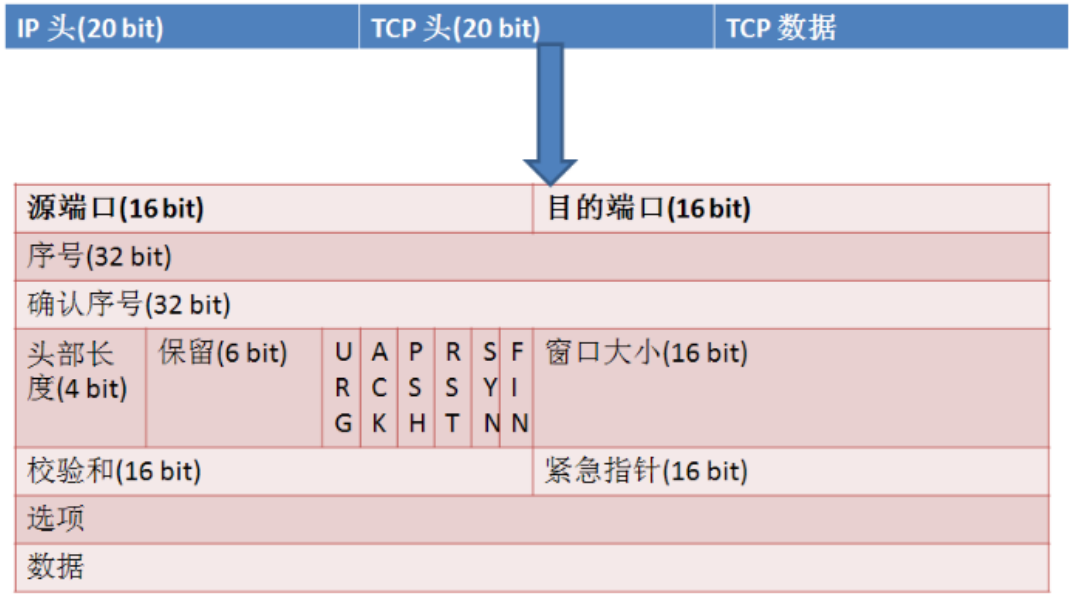
销毁（ libnet_destroy( ) ）

如何发阻断包

• TCP RST 原理

TCP是在IP网络层之上的传输层协议，用于提供port到port面向连接的可靠的字节流传输。

- port到port：IP层只管数据包从一个IP到另一个IP的传输，IP层之上的TCP层加上端口后，就是面向进程了，每个port都可以对应到用户进程。
- 可靠：TCP会负责维护实际上子虚乌有的连接概念，包括收包后的确认包、丢包后的重发等来保证可靠性。由于带宽和不同机器处理能力的不同，TCP要能控制流量。
- 字节流：TCP会把应用进程传来的字节流数据切割成许多个数据包，在网络上发送。IP包是会失去顺序或者产生重复的，TCP协议要能还原到字节流本来面目

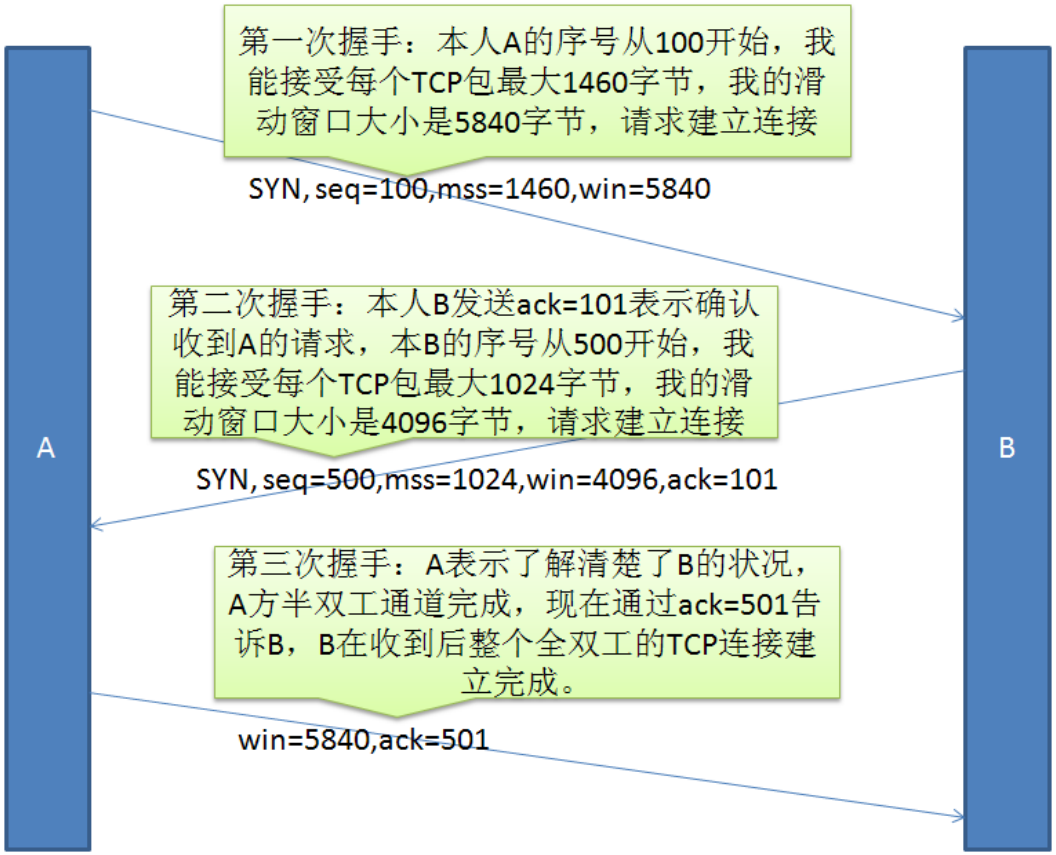


标志位共有六个，其中RST位就在TCP异常时出现

如何发阻断包

- **TCP RST 原理**

A向B建立TCP连接--三次握手



SYN标志位、序号、滑动窗口大小。

建立连接的请求中，标志位SYN都要置为1，在这种请求中会告知MSS段大小，就是本机希望接收TCP包的最大大小。

发送的数据TCP包都有一个序号。它是这么得来的：最初发送SYN时，有一个初始序号，根据RFC的定义，各个操作系统的实现都是与系统时间相关的。之后，序号的值会不断的增加，比如原来的序号是100，如果这个TCP包的数据有10个字节，那么下次的TCP包序号会变成110。

滑动窗口用于加速传输，比如发了一个seq=100的包，理应收到这个包的确认ack=101后再继续发下一个包，但有了滑动窗口，只要新包的seq与没有得到确认的最小seq之差小于滑动窗口大小，就可以继续发。

如何发阻断包

- **TCP RST 原理**

滑动窗口

滑动窗口毫无疑问是用来加速数据传输的。TCP要保证“可靠”，就需要对一个数据包进行ack确认表示接收端收到。有了滑动窗口，接收端就可以等收到许多包后只发一个ack包，确认之前已经收到过的多个数据包。有了滑动窗口，发送端在发送完一个数据包后不用等待它的ack，在滑动窗口大小内可以继续发送其他数据包

TCP的各种实现中，在滑动窗口之外的seq会被扔掉

如何发阻断包

TCP RST 原理

RST 和 FIN 都能用来中断连接

正常关闭连接状态变迁图

FIN标志位用来表示正常关闭连接。

图的左边是主动关闭连接方，右边是被动关闭连接方，用netstat命令可以看到标出的连接状态。

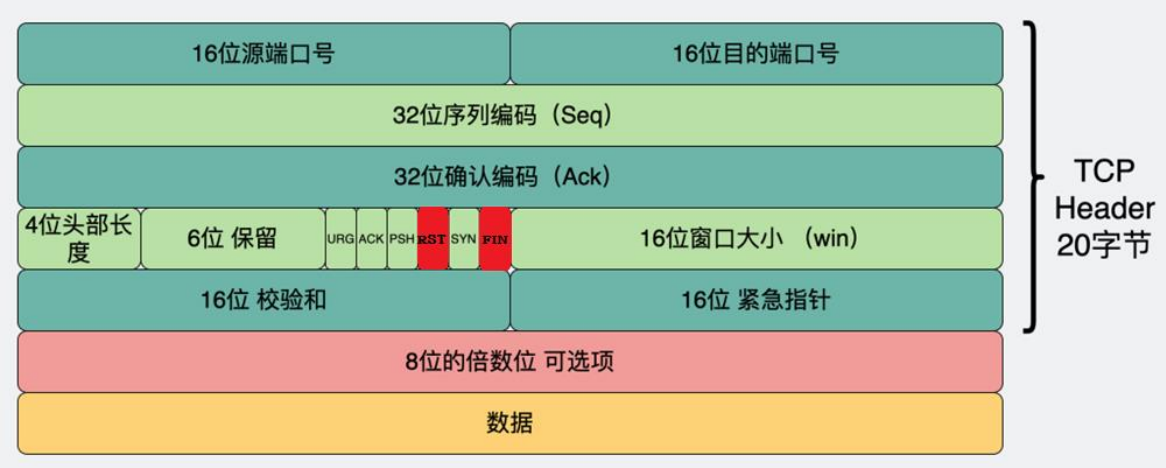
FIN是正常关闭，它会根据缓冲区的顺序来发的，就是说缓冲区FIN之前的包都发出去后再发FIN包，这与RST不同。

如何发阻断包

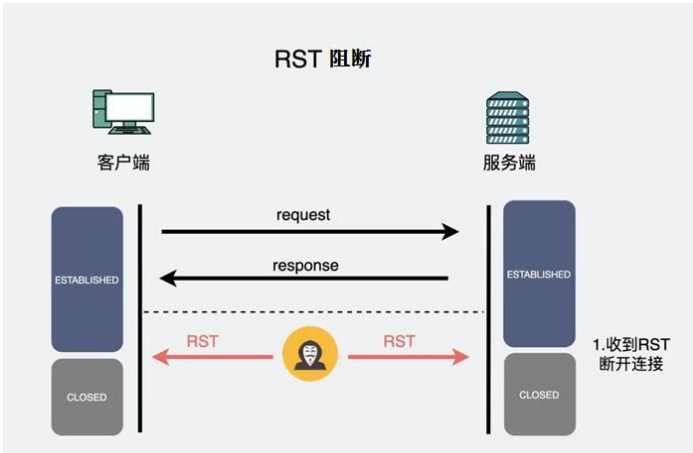
TCP 连接阻断

RST标志位

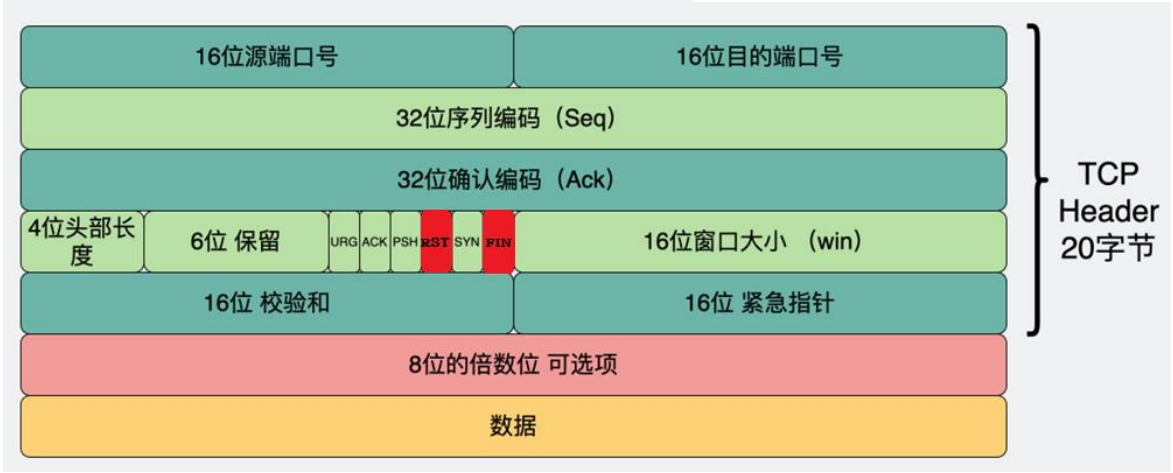
- RST表示复位，用来异常的关闭连接，在TCP的设计中它是不可或缺的。发送RST包关闭连接时，不必等缓冲区的包都发出去（不同于FIN包），直接就丢弃缓存区的包发送RST包。而接收端收到RST包后，也不必发送ACK包来确认。
- TCP处理程序会在自己认为的异常时刻发送RST包。
- 如A向B发起连接，但B之上并未监听相应的端口，这时B操作系统上的TCP处理程序会发RST包。
- 又如，AB正常建立连接了，正在通讯时，A向B发送了FIN包要求关连接，B发送ACK后，网断了，A通过若干原因放弃了这个连接（例如进程重启）。网通了后，B又开始发数据包，A收到后表示压力很大，不知道这连接哪来的，就发了个RST包强制把连接关了，B收到后会出现connect reset by peer错误



TCP 连接阻断

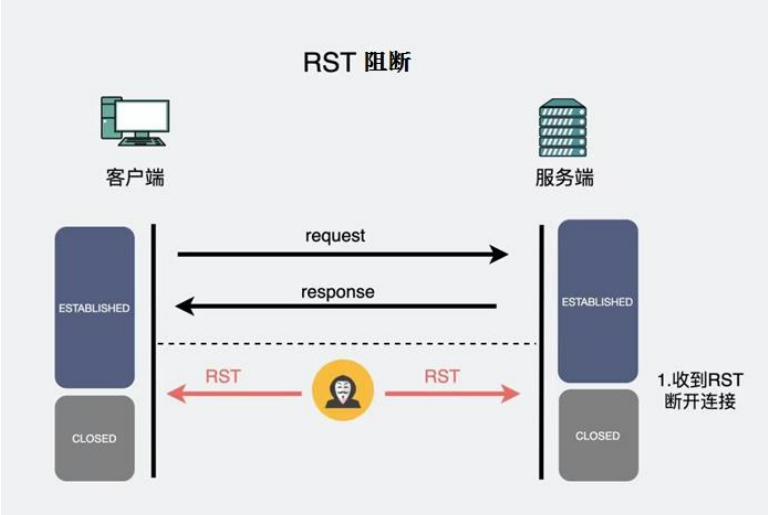


客户端A和服务器B之间建立了TCP连接，此时中间人C在旁路伪造了一个TCP包发给B，使B异常的断开了与A之间的TCP连接，就是RST/FIN阻断。



构造RST报文和FIN报文的方法就是将TCP头部中，对应的标志位RST或者FIN标志位置1

TCP 连接阻断



伪造什么样的TCP包可以达成目的呢？我们至顶向下的看。

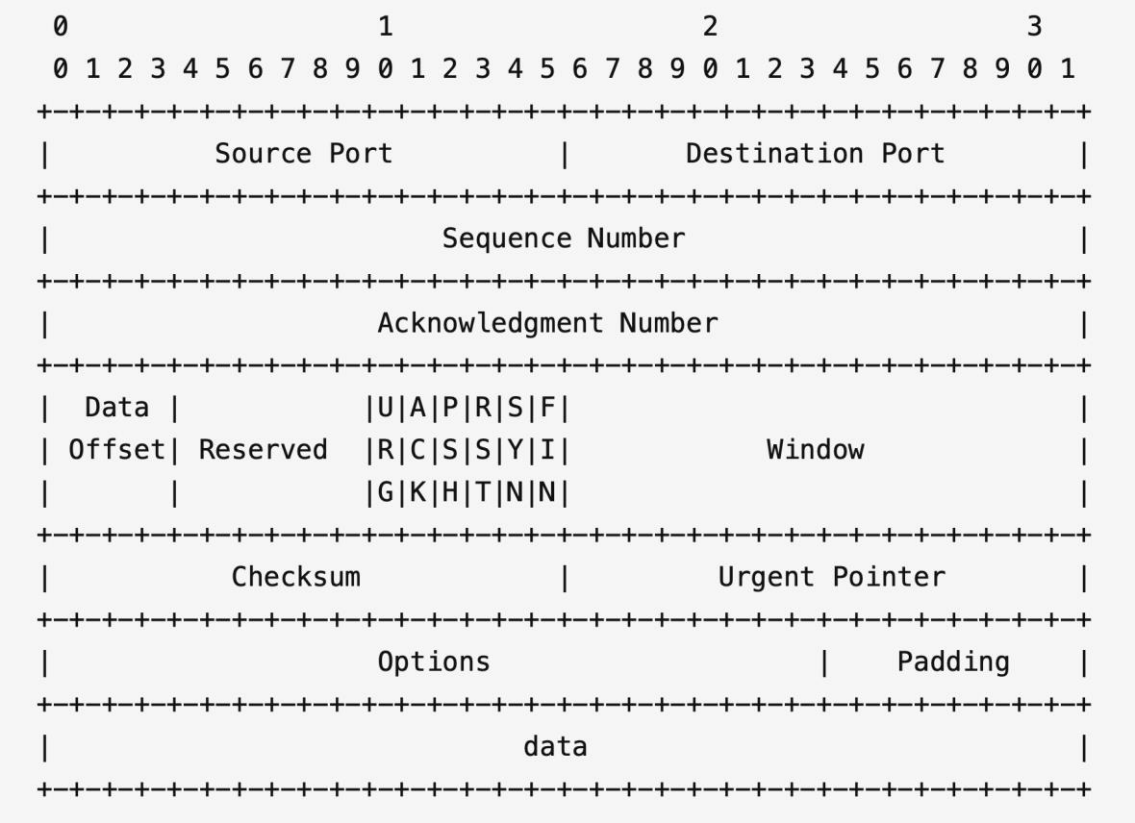
假定C伪装成A发过去的包，这个包如果是RST包的话，毫无疑问，B将会丢弃与A的缓冲区上所有数据，强制关掉连接。

如果发过去的包是SYN包，那么，B会表示A已经发疯了（与OS的实现有关），正常连接时又来建新连接，B主动向A发个RST包，并在自己这端强制关掉连接。

这两种方式都能够达到复位攻击的效果

TCP RST阻断TCP 连接

TCP协议中的ACK和SEQ

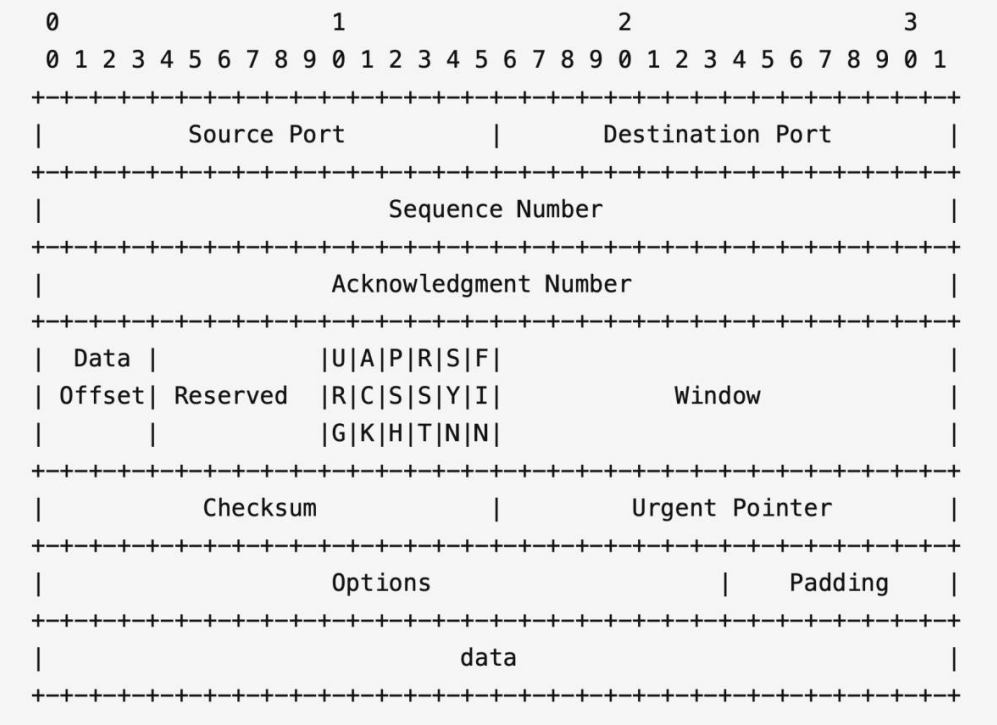


seq（Sequence Number）：32bits，表示这个tcp包的序列号。tcp协议拼凑接收到的数据包时，根据seq来确定顺序，并且能够确定是否有数据包丢失。

ack（Acknowledgment Number）：32bits，表示这个包的确认号。首先意味着已经收到对方了多少字节数据，其次告诉对方接下来的包的seq要从ack确定的数值继续接力。

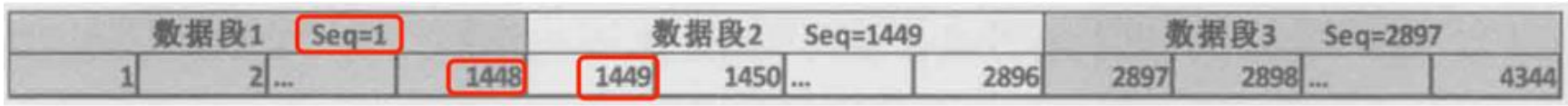
TCP RST阻断TCP 连接

TCP协议中的ACK和SEQ



seq（Sequence Number）：32bits，表示这个tcp包的序列号。tcp协议拼凑接收到的数据包时，根据seq来确定顺序，并且能够确定是否有数据包丢失。

ack（Acknowledgment Number）：32bits，表示这个包的确认号。首先意味着已经收到对方了多少字节数据，其次告诉对方接下来的包的seq要从ack确定的数值继续接力。



复习 – TCP RST阻断TCP 连接

一次网页浏览的http请求抓包

No.	Time	Source	Destination	Protocol	Length	Info
1	11:26:26.139723	192.168.10.40	61.135.185.32	TCP	78	65118 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=1518395
2	11:26:26.144922	61.135.185.32	192.168.10.40	TCP	86	80 → 65118 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1452 WS=32 SAC
3	11:26:26.144975	192.168.10.40	61.135.185.32	TCP	54	65118 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0
4	11:26:26.145023	192.168.10.40	61.135.185.32	HTTP	131	GET / HTTP/1.1
5	11:26:26.161828	61.135.185.32	192.168.10.40	TCP	68	80 → 65118 [ACK] Seq=1 Ack=78 Win=29312 Len=0
6	11:26:26.164639	61.135.185.32	192.168.10.40	TCP	1502	80 → 65118 [PSH, ACK] Seq=1 Ack=78 Win=29312 Len=1440 [TCP segment
7	11:26:26.164677	192.168.10.40	61.135.185.32	TCP	54	65118 → 80 [ACK] Seq=78 Ack=1441 Win=260672 Len=0
8	11:26:26.167381	61.135.185.32	192.168.10.40	HTTP	1403	HTTP/1.1 200 OK (text/html)
9	11:26:26.167414	192.168.10.40	61.135.185.32	TCP	54	65118 → 80 [ACK] Seq=78 Ack=2782 Win=260800 Len=0
10	11:26:26.167639	192.168.10.40	61.135.185.32	TCP	54	65118 → 80 [FIN, ACK] Seq=78 Ack=2782 Win=262144 Len=0
11	11:26:26.173619	61.135.185.32	192.168.10.40	TCP	60	80 → 65118 [ACK] Seq=2782 Ack=79 Win=29312 Len=0

三次握手

No.	Time	Source	Destination	Protocol	Length	Info
1	11:26:26.139723	192.168.10.40	61.135.185.32	TCP	78	65118 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=1518395
2	11:26:26.144922	61.135.185.32	192.168.10.40	TCP	86	80 → 65118 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1452 WS=32 SAC
3	11:26:26.144975	192.168.10.40	61.135.185.32	TCP	54	65118 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0

（客户端）1号包：我能和你建立连接吗？ seq=0 没有ack len=0

（服务端）2号包：我收到了，能进行连接。 seq=0 ack=1（收到了seq=0，下一个seq=1） len=0

（客户端）3号包：好的，那我们就连接吧。 seq=1 ack=1 len=0

TCP RST阻断TCP 连接

一次网页浏览的http请求抓包

数据传输

```
    4 11:26:26.145023 192.168.10.40   61.135.185.32    HTTP    131 GET / HTTP/1.1
    5 11:26:26.161828 61.135.185.32   192.168.10.40    TCP      68 80 → 65118 [ACK] Seq=1 Ack=78 Win=29312 Len=0
    6 11:26:26.164639 61.135.185.32   192.168.10.40    TCP    1502 80 → 65118 [PSH, ACK] Seq=1 Ack=78 Win=29312 Len=1440 [TCP segm
    7 11:26:26.164677 192.168.10.40   61.135.185.32    TCP      54 65118 → 80 [ACK] Seq=78 Ack=1441 Win=260672 Len=0

▸ Frame 4: 131 bytes on wire (1048 bits), 131 bytes captured (1048 bits) on interface en0, id 0
▸ Ethernet II, Src: Apple_69:31:66 (a4:83:e7:69:31:66), Dst: Ubiquiti_4c:90:17 (18:e8:29:4c:90:17)
▸ Internet Protocol Version 4, Src: 192.168.10.40, Dst: 61.135.185.32
▾ Transmission Control Protocol, Src Port: 65118, Dst Port: 80, Seq: 1, Ack: 1, Len: 77
    Source Port: 65118
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 77]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 2588606373
    [Next Sequence Number: 78    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3909216787
    0101 .... = Header Length: 20 bytes (5)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 4096
    [Calculated window size: 262144]
    [Window size scaling factor: 64]
    Checksum: 0x3e60 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ [SEQ/ACK analysis]
  ▸ [Timestamps]
    TCP payload (77 bytes)
```

客户端 **4**号包：我要首页信息：客户端发送**http**请求，需要**tcp**进行控制，然后交给**ip**层，然后由网卡发出

seq=1 上次没有数据传输，seq不变； ack=1 服务器也还没有数据，下一个seq=1； len=77 传输的字节数

服务器 **5**号包：好的，收到请求：

seq=1 如4号包要求； ack=78 4号包的seq+len； len=0

服务端**6**号包：给你数据：

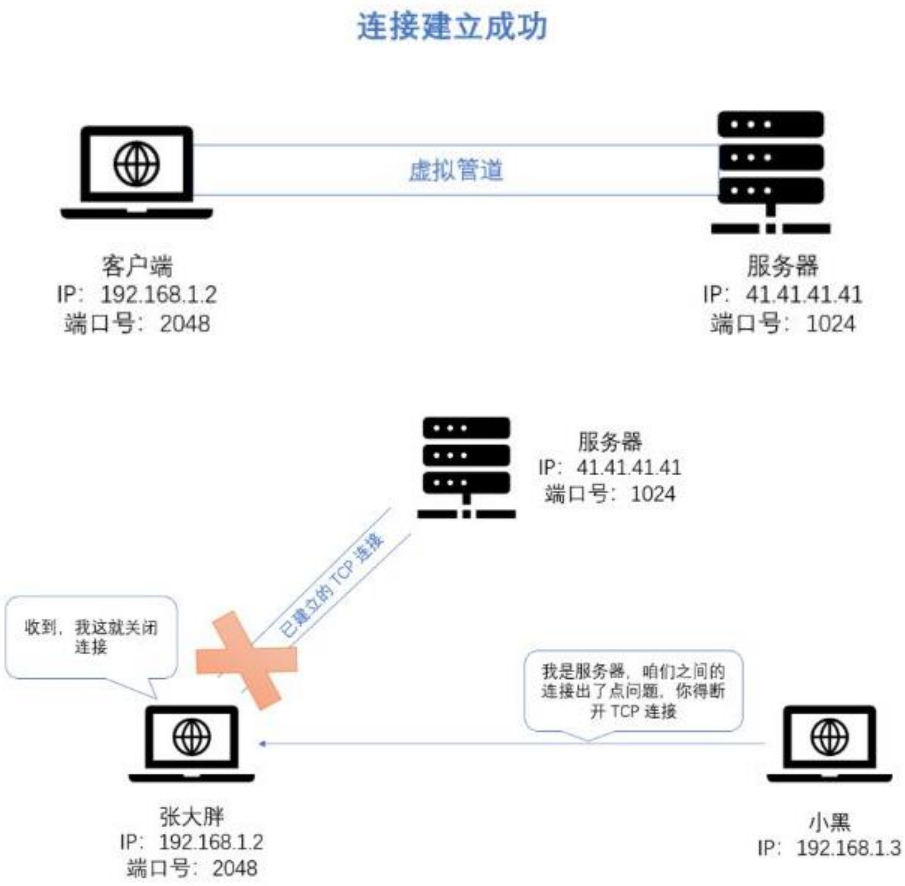
seq=1 ； ack=； len=1440

客户端**7**号包：收到数据：

seq=78 ； ack=1441 6号包的 seq+len； len=0

TCP RST阻断TCP 连接

RST阻断



伪造 RST 包，**最关键的两个因素是源端口和序列号，一个 TCP 链接是由「客户端端口，客户端 IP，服务端端口，服务端 IP」这四个信息惟一肯定的**，首先须要知道这四个信息

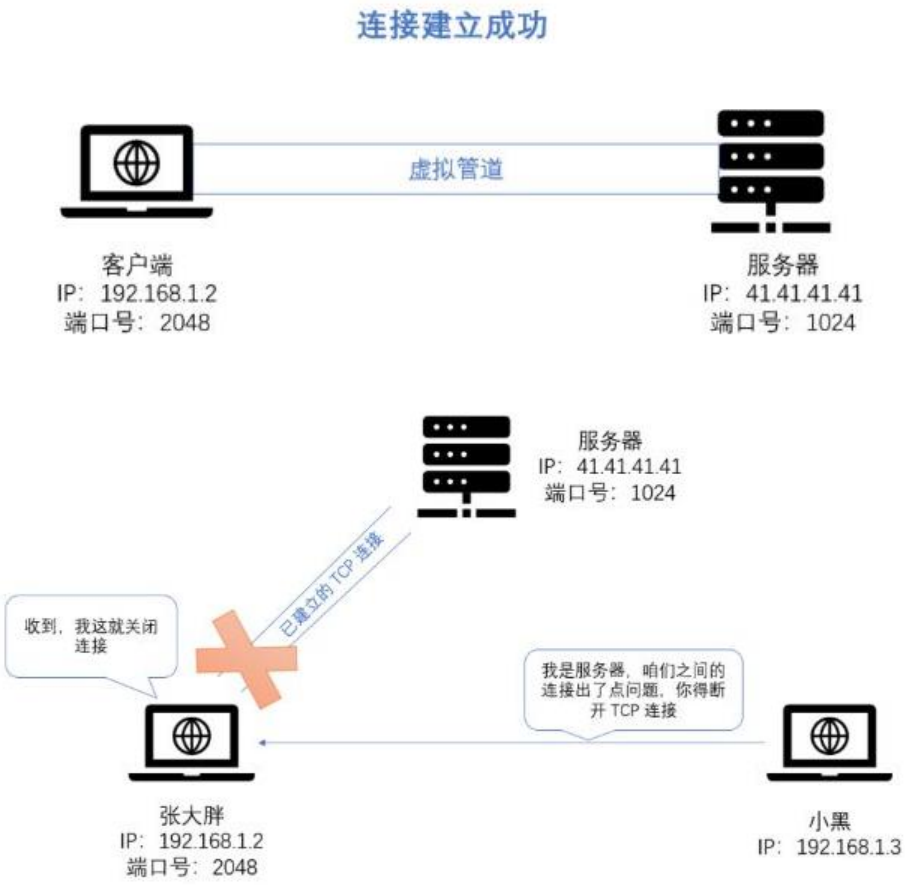
序列号简单的计算方式：

- 假设C向S请求数据，seq=seq1，ack=ack1 len=len1
- 旁路中间节点伪造返回给客户端的seq2和ack2需要满足这种关系：seq2 = ack1 ack2 = seq1+len1
- 阻断S向C方向原理相同
- 需要根据收到的报文计算seq和ack

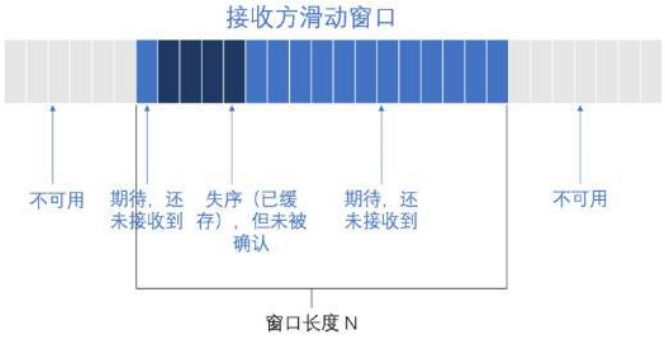
RST 攻击并不能阻止全部的 TCP 链接，好比说在浏览一些简单网页的时候，阻断率就不高，由于等到 RST 包到达客户机电脑的时候，页面内容就已经传输结束了

TCP RST阻断TCP 连接

RST阻断



序列号的值不在A之前向B发送时B的滑动窗口内，B是会主动丢弃的。



多猜几个seq，防止比真正客户机发的慢（参见给大家的参考代码）

```
    ...
    seq = ntohl(tcp->th_ack);
    win = ntohs(tcp->th_win);

    for (i = 0; i < 10; i++) {
        seq += (i * win);
        ...
        libnet_write(l);
    }
    ...
```

TCP 连接阻断

■ 通常构造阻断报文的过程是：

初始化 ⇒ 构造IP头部 ⇒ 计算IP校验和 ⇒ 构造TCP头部 ⇒ 计算TCP校验和 ⇒ 发送阻断报文

■ 构造阻断程序其他需要注意的问题：

- 为了增加成功率，发送双向RST包
- 可以参考libnids源码中nids_killtcp()函数，由于直接基于RAW_SOCKET发送，效率略高，但函数有待继续封装复用
- 构造包的过程中需要注意网络字节序的转换
- 构造封堵包最重要的是序列号的计算，算法可以适当调整

■ 构造报文的TCP头部中 Seq 的计算方法：

- 假设客户端C向服务器S请求数据，seq是seq1，ack是ack1
- 旁路中间节点伪造返回给客户端的seq2和ack2需要满足这种关系：seq2 = ack1 ack2 = seq1+datalen(伪造报文的长度，不包括ip头和tcp头)
- 阻断S向C方向原理相同
- 需要根据收到的报文计算seq和ack

序列号的值不在A之前向B发送时B的滑动窗口内，B是会主动丢弃的，精准点计算，按介绍的方法可以确保不错问题。

THE END