

附录A tcpdump程序

tcpdump程序是由Van Jacobson、Craig Leres和Steven McCanne编写的，他们都来自加利福尼亚大学伯克利分校的劳伦斯伯克利实验室。本书中使用的是 2.2.1版（1992年6月）。

tcpdump通过将网络接口卡设置为混杂模式（promiscuous mode）来截获经过网络接口的每一个分组。正常情况下，用于诸如以太网媒体的接口卡只截获送往特定接口地址或广播地址的链路层的帧（2.2节）。

底层的操作系统必须允许将一个接口设置成混杂模式，并且允许一个用户进程截获帧。下列的操作系统可以支持tcpdump，或者可以加入对tcpdump的支持：4.4BSD、BSD/386、SunOS、Ultrix和HP-UX。参考一下随着tcpdump发布的README文件，了解它支持哪些操作系统以及哪些版本。

除了tcpdump还有其他一些选择。在图 10-8中，我们使用了Solaris 2.2的程序snoop来查看一些分组。AIX 3.2.2提供了iptrace程序，该程序也提供了类似的功能。

A.1 BSD 分组过滤器

当前由BSD演变而来的 Unix内核提供了 BSD 分组过滤器 BPF (BSD Packet Filter)，tcpdump用它来截获和过滤来自一个被置为混杂模式的网络接口卡的分组。BPF也可以工作在点对点的链路上，如 SLIP（2.4节），不需要什么特别的处理就可以截获所有通过接口的分组。BPF还可以工作在环回接口上（2.7节）。

BPF有一个很长的历史。1980年卡耐基梅隆大学的Mike Accetta和Rick Rashid创造了Enet分组过滤程序。斯坦福的Jeffrey Mogul将代码移植到BSD，从1983年开始继续开发。从那以后，它演变为DEC的Ultrix分组过滤器、SunOS 4.1下的一个STREAMS NIT模块和BPF。劳伦斯伯克利实验室的Steven McCanne在1990年的夏天实现了BPF。其中很多设计来自于Van Jacobson。[McCanne and Jacobson 1993] 给出了最新版本的细节以及Sun的NIT的一个比较。

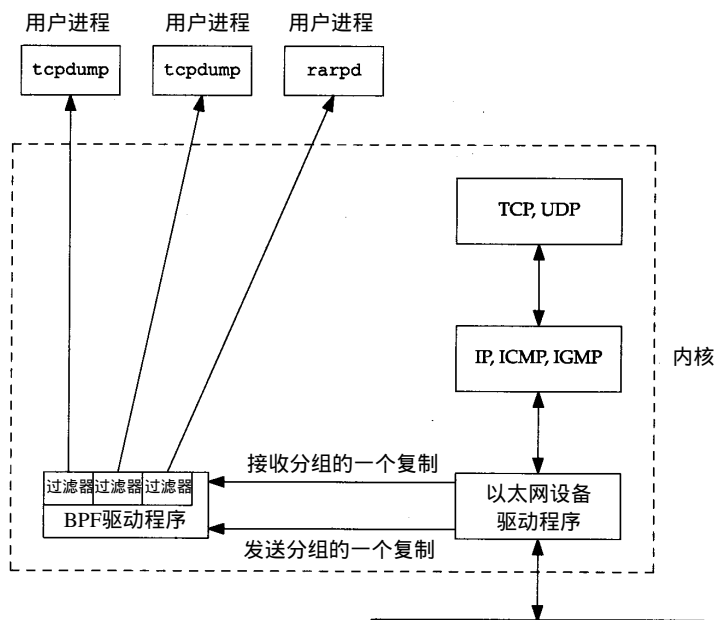
图A-1显示了用于以太网的BPF的特征。

BPF将以太网设备驱动程序设置为混杂模式，然后从驱动程序那里接收每一个收到的分组和传输的分组。这些分组要通过一个用户指明的过滤器，使得只有那些用户进程感兴趣的分组才会传递给用户进程。

多个进程可以同时监视一个接口，每个进程指明了一个自己的过滤器。图 A-1显示了tcpdump的两个实例进程和一个RARP守护进程（5.4节）监视同样的以太网接口。tcpdump的每个实例指明了一个自己的过滤器。tcpdump的过滤器可以由用户在命令行指明，而rarpd总是使用只截获RARP请求的过滤器。

除了指明一个过滤器，BPF的每个用户还指明了一个超时定时器的值。因为网络的数据传输率可以很容易地超过CPU的处理能力，而且一个用户进程从内核中只读小块数据的代价昂

贵, 因此, BPF试图将多个帧装载进一个读缓存, 只有缓存满了或者用户指明的超时到期才将读缓存保存的帧返回。tcpdump将超时定时器置为1秒, 因为它一般从BPF收到很多数据。而RARP守护进程收到的帧很少, 所以rarpd将超时置为0(收到一个帧就返回)。



图A-1 BSD分组过滤器

用户指明的过滤器告诉BPF用户进程对什么帧感兴趣, 过滤器是对一个假想机器的一组指令。这些指令被内核中的BPF过滤器解释。在内核中过滤, 而不在用户进程中, 减少了必须从内核传递到用户进程的数据量。RARP守护进程总是使用绑定在程序里的、同样的过滤程序。另一方面, tcpdump在每次运行时, 让用户在命令行指明一个过滤表达式。tcpdump将用户指明的表达式转换为相应的BPF的指令序列。tcpdump表达式的例子如下:

```
% tcpdump tcp port 25
% tcpdump'icmp[0] != 8 and icmp[0] != 0
```

第一个只打印源端口和目的端口为25的TCP报文段。第二个只打印不是回送请求和回送应答的ICMP报文(也就是非ping的分组)。这个表达式指明了ICMP报文的第一个字节, 图6-2中的type字段, 不等于8或0, 即图6-3中的回送请求和回送应答。正像你所看到的, 设计过滤器需要有底层分组结构的知识。第二个例子中的表达式被放在一对单引号中, 防止Unix外壳程序解释特殊字符。

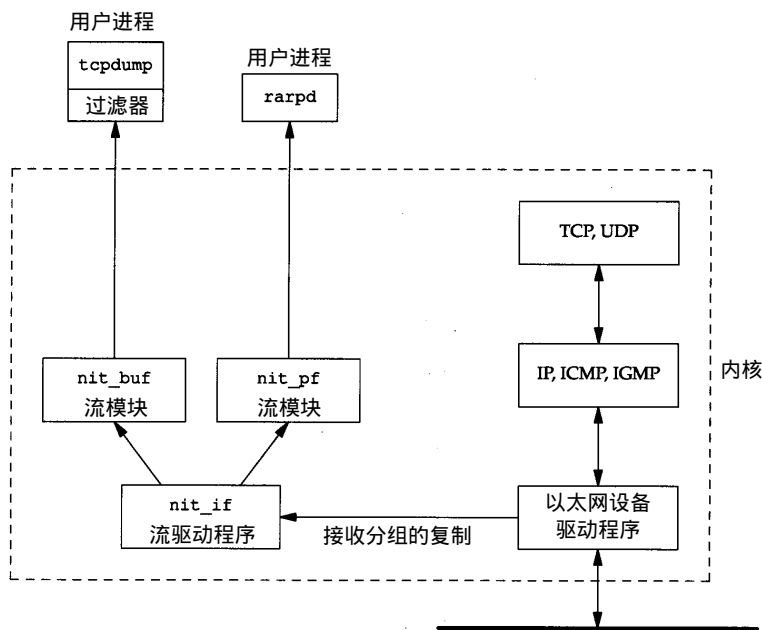
参考tcpdump(1)的手册, 了解用户可以指明的表达式的全部细节。bpf(4)的手册详细描述了BPF使用的假想机器指令。[McCanne and Jacobson 1993] 比较了这个假想机器方法与其他方法的设计与性能。

A.2 SunOS的网络接口分接头

SunOS 4.1.x提供了一个STREAMS伪设备驱动程序(pseudo-device driver), 称为网络接口分接头(Network Interface Tap)或者NIT([Rago 1993] 包含了流设备驱动程序的其他细节。我

们把这种特征叫作“流”)。NIT类似于BSD分组过滤器,但不如后者功能强大和效率高。图A-2显示了使用NIT所用到的流模块。这个图与图A-1之间的一个不同点在于BPF可以截获网络接口收到的和传送的分组,而NIT只能截获接口收到的分组。将tcpdump与NIT结合起来意味着我们只能看见由网络中其他主机发送来的分组——即根本不可能看见我们自己主机发送的分组(尽管BPF可以工作在SunOS 4.1.x上,但它需要对以太网设备驱动程序的源代码进行改变,大多数的用户没有权限访问源代码,因而这是不可能的)。

当设备/dev/nit被打开时,流驱动程序nit_if就会被打开。既然NIT是使用流来构造的,处理模块可以放在nit_if驱动程序之上。tcpdump将模块nit_buf放在STREAM之上。这个模块将多个网络帧聚集在一个读缓存中,允许用户进程指明一个超时的值。这种情况类似于我们在BPF中所描述的。RARP守护进程没有把这个模块放在它的流之上,因为它只处理了一小部分分组。



图A-2 SunOS的网络接口分接头

用户指明的过滤由流模块 nit_pf 处理。在图A-2中,我们注意到这个模块被 RARP 守护进程所用,但没有被 tcpdump 使用。在 SunOS 操作系统中, tcpdump 代之以在用户进程中完成自己的过滤操作。这么做的理由是 nit_pf 使用的假想机器的指令与 BPF 所支持的指令不同(不如 BPF 所支持的功能强大)。这就意味着当用户对 tcpdump 指明了一个过滤表达式时,与 BPF 相比较,使用 NIT 就会有更多的数据在内核与用户进程之间交换。

A.3 SVR4数据链路提供者接口

SVR4支持数据链路提供者接口 DLPI (Data Link Provider Interface),它是OSI数据链路服务定义的一个流实现。SVR4的大多数版本支持第1版的DLPI,SVR4.2同时支持第1版和第2版,Sun的Solaris 2.x支持第2版,但是增强了一些功能。

像tcpdump的网络监视程序必须使用DLPI来直接访问数据链路设备驱动程序。在Solaris 2.x中,分组过滤的流模块被改名为 pfmod,缓存模块被改名为 bufmod。

尽管Solaris 2.x还很新, tcpdump在其上的一个实现有一天也会出现。Sun还提供了一个叫作snoop的程序, 完成的功能类似于tcpdump (snoop代替了SunOS 4.x的程序etherfind)。作者还不知道tcpdump到vanilla SVR4上的任何端口实现。

A.4 tcpdump的输出

tcpdump的输出是“原始的”。在本书中包含它的输出时, 我们对它进行了修改以便阅读。

首先, 它总是输出它正在监听的网络接口的名字。我们把这一行给删去了。

其次, tcpdump输出的时间戳在一个微秒精度的系统中采用如同09:11:22.642008的格式, 在一个10ms时钟精度的系统中则如同09:11:22.64一样(在附录B中, 我们更多地讨论了计算机时钟的精度)。在任何一种情况下, HH:MM:SS的格式都不是我们想要的。我们感兴趣的是每个分组与开始监听的相对时间以及与下面分组的时间差。我们修改了输出以显示这两个时间差。第1个差值在微秒精度的系统中打印到十进制小数点后面6位(对于只有10 ms精度的系统打印到小数点后面2位), 第2个差值打印到十进制小数点后面4位或2位(依赖于时钟精度)。

本书中大多数tcpdump的输出都是在sun主机上收集的, 它提供了微秒精度。一些输出来自于运行0.9.4版BSD/386操作系统的主机bsdi, 它只提供了10 ms的精度(如图5-1所示)。一些输出收集于当bsdi主机运行1.0版BSD/386时, 后者提供了微秒级的精度。

tcpdump总是打印发送主机的名字, 接着一个大于号, 然后是目的主机的名字。这样显示很难追踪两个主机之间的分组流。尽管tcpdump输出仍然显示了数据流的方向, 但我们经常把这条输出删掉, 代替以产生一条时间线(在本书中的第一次出现是在图6-11)。在我们的时间线上, 一个主机在左边, 另一个在右边。这样很容易看出哪一边发送分组, 哪一边接收分组。

我们给tcpdump的每条输出增加了行号, 使得我们可以在书中引用特定的行。还在某些行之间增加了额外的空白, 以区别一些不同的分组交换。

最后, tcpdump的输出可能会超出一页的宽度。我们在太长行的适当地方进行了换行。

作为一个例子, 相应于图4-4的tcpdump的原始输出显示在图A-3中。这里假设了一个80列的终端窗口。

没有显示我们键入的中断键(用于中止tcpdump), 也没有显示接收到的和漏掉的分组个数(漏掉的分组是那些到达得太快, tcpdump来不及处理的分组。因为本文中的例子经常运行在另外一个空闲网络上, 所以漏掉的分组个数总是0)。

```
sun % tcpdump -e
tcpdump: listening on le0
09:11:22.642008 0:0:c0:6f:2d:40 ff:ff:f f:ff:ff:ff arp 60: arp who-has svr4 tell
bsdi
09:11:22 .644182 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60: arp reply svr4 is-at 0:0:
:c0:c2:9b:26
09:11:22.644839 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
S 596459521:596459521(0) win 4096 <mss 1024> [tos 0x10]
09:11:22. 649842 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60: svr4.discard > bsdi.1030:
S 3562228225:3562228225(0) ack 596459522 win 4096 <mss 1024>
09:1 1:22.651623 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
. ack 1 win 4096 [tos 0x10]
```

我们没有显示其他4个分组

键入中断字符来中断显示

```
^?
9 packets received by filter
0 packets dropped by kernel
```

图A-3 图4-4的tcpdump 的输出

A.5 安全性考虑

很明显，截获网络中传输的数据流使我们可以看到很多不应该看到的东西。例如，Telnet和FTP用户输入的口令在网络中传输的内容和用户输入的一样（与口令的加密表示相比，这称为口令的明文表示。在Unix口令文件中，一般是/etc/passwd或/etc/shadow，存储的是加密的表示）。然而，很多时候一个网络管理员需要使用一个类似于tcpdump的工具来分析网络中出现的问题。

我们是把tcpdump作为一个学习的工具，用来查看网络中实际传输的东西。对tcpdump以及其他厂商提供的类似工具的访问权限依赖于具体系统。例如，在SunOS，对NIT设备的访问只限于超级用户。BSD的分组过滤器使用了一种不同的技术：通过对/dev/bpfXX设备的授权来控制访问。一般来说，只有属主才能读写这些设备（属主应该是超级用户），对于同组用户是可读的（经常是系统管理组）。这就是说如果系统管理员不对程序设置用户的ID，一般的用户是不能运行类似于tcpdump的程序的。

A.6 插口排错选项

查看一个TCP连接上发生的事情的另一种方法是使能插口排错选项，当然是在支持这一特征的系统中。这个特征只能工作在TCP上（其他协议都不行），并且需要应用程序支持（当应用程序启动时，使能一个插口排错选项）。

大多数伯克利演变的实现都支持这个特征，包括SunOS、4.4BSD和SVR4。

程序使能了一个插口选项，内核就会保留在那个连接上发生的事情的一个痕迹记录。在这之后，所有记录的信息都可以使用trpt(8)程序打印出来。使能一个插口排错选项不需要特别的许可，但是因为trpt程序访问了内核的内存，所以运行trpt需要特别的权限。

sock程序（附录C）的-D选项支持这个特征，但是输出的信息比相应的tcpdump的输出更难解析和理解。然而，我们在21.4节确实使用它查看了TCP连接上tcpdump不能访问的内核变量。