理解 TCP/IP 网络栈 & 编写网络应用

2015/05/05 • IT技术 • 1.5K 阅读 • TCP, 网络应用

分享到 : 33

impress让你的内容“舞”起来
慕课网2048私人订制
Java实现Base64加密
Android必学-BaseAdapter的使用与优化

本文由 伯乐在线 - 蛋疼的axb 翻译。未经许可，禁止转载！
英文出处：Hyeongyeop Kim。欢迎加入翻译组。

1. 译注

之前在网上看到了这篇描述tcp网络栈原理的文章，感觉不错，决定抽空把这篇文章翻译一下。一来重新温习一下TCP相关知识，二来练练英文。 很久没翻译文章了难免有误，建议有能力的同学还是看一下原文。

2. 概述

我们难以想象没有了TCP/IP之后的网络服务。所有我们开发并在NHN使用的网络服务都基于TCP/IP这个坚实的基础。理解数据如何通过网络传输可以帮助你通过调优、排查或引进新技术之类的手段提升性能。

本文将基于Linux OS和硬件层的数据流和控制流来描述网络栈的整体运行方式。

3. TCP/IP的关键特性

我如何设计一个能在快速传输数据的同时保证数据顺序并且不丢失数据的网络协议？TCP/IP在设计时就基于这个考虑。以下是在了解整体网络栈前需要知道的TCP/IP的主要特性：

> TCP和IP
> 从技术上讲，TCP和IP处于不同的层，应该分别解释它们。但在这里我们把他们看做一个整体。

1. 面向连接首先，传输数据前需要在两个终端之间建立连接（本地和远程）。在这里，“TCP连接标识符(TCP connection identifier)”是两个终端地址的组合，类似本地ip地址，本地端口号，远程ip地址，远程端口号的形式。
2. 双向字节流通过字节流实现双向数据通信。
3. 顺序投递接收者在接收数据时与发送者发送的数据顺序相同。因此，数据需要是有序的，为了表示这个顺序，TCP/IP使用了32位的int数据类型。
4. 通过ACK实现可靠性当发送者向接收者发送数据，但没有收到来自接收方的ACK（acknowledgement，应答）时，发送者的TCP层将重发数据。因此，发送者的TCP层会把接收者还没有应答的数据暂存起来。
5. 流量控制发送者的发送速度与接收者的接收能力相关。接收者会把它能接收的最大字节数（未使用的缓冲区大小，又叫接收窗口，receive window）告知发送者。发送者发送的最大字节数与接收者的接收窗口大小一致。
6. 阻塞控制阻塞窗口是不同于接收窗口的另一个概念，它通过限制网络中的数据流的体积来防止网络阻塞。类似于接收窗口，发送者通过通过一些算法（例如TCP Vegas，Westwood，BIC，CUBIC）来计算发送对应的接收者的阻塞窗口能容纳的最多的数据。和流量控制不同，阻塞控制只在发送方实现。（译注：发送者类似于通过ack时间之类的算法判断当前网络是否阻塞，从而调节发送速度）

4. 数据传输

网络栈有很多层。图一表示了这些层的类型：

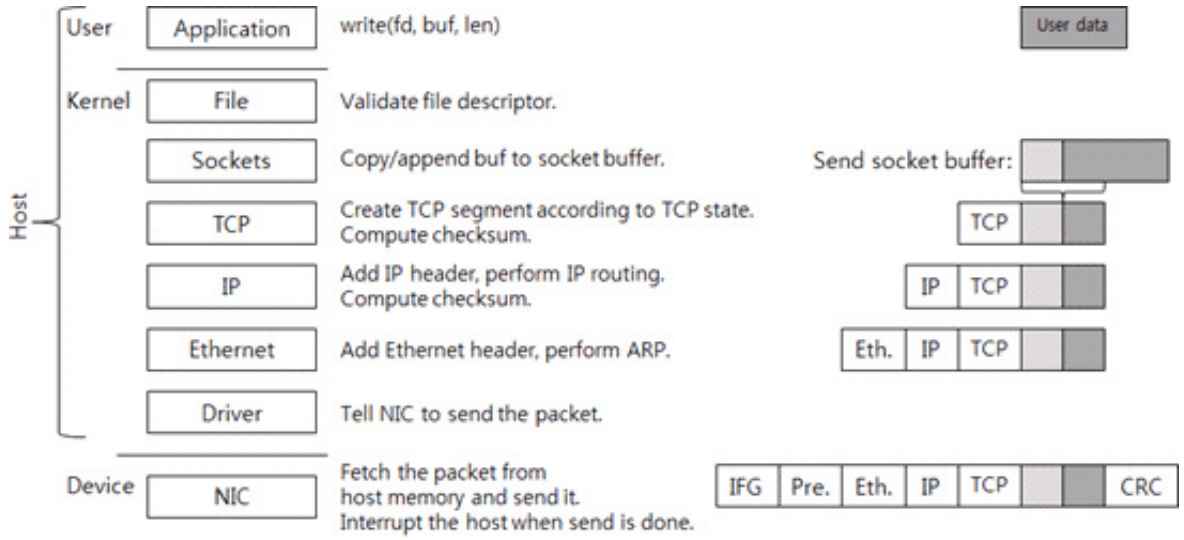

图1：发送数据时网络栈中各层对数据的操作

这些层可以被大致归类到三个区域中：

1. 用户区
2. 内核区

3. 设备区

用户区和内核区的任务由CPU执行。用户区和内核区被叫做“主机（host）”以区别于设备区。在这里，设备是发送和接收数据包的网络接口卡（Network Interface Card,NIC）。它有一个更常用的术语：网卡。

我们来了解一下用户区。首先，应用程序创建要发送的数据（图1中的“User Data”）并且调用write()系统调用来发送数据（在这里假设socket已经被创建了，对应图1中的“fd”）。当系统调用被调用之后上下文切换到内核区。

像Linux或者Unix这类POSIX系列的操作系统通过文件描述符（file descriptor）把socket暴露给应用程序。在这类系统中，socket是文件的一种。文件系统执行简单的检查并调用socket结构中指向的socket函数。

内核中的socket包含两个缓冲区。

1. 一个用于缓冲要发送的数据
2. 一个用于缓冲要接收的数据

当write()系统调用被调用时，用户区的数据被拷贝到内核内存中，并插入到socket的发送缓冲区末尾。这样来保证发送的数据有序。在图1中，浅灰色框表示在socket缓冲区中的数据。之后，TCP被调用了。

socket会关联一个叫做TCP控制块（TCP Control Block）的结构，TCB包含了处理TCP连接所需的数据。包括连接状态（LISTEN，ESTABLISHED，TIME_WAIT），接收窗口，阻塞窗口，顺序号，重发计时器，等等。

如果当前的TCP状态允许数据传输，一个新的TCP分段（TCP segment,或者叫数据包,packet）将被创建。如果由于流量控制或者其它原因不能传输数据，系统调用会在这里结束，之后会返回到用户态。（换句话说，控制权会交回到应用程序代码）

TCP分段有两部分

1. TCP头
2. 携带的数据

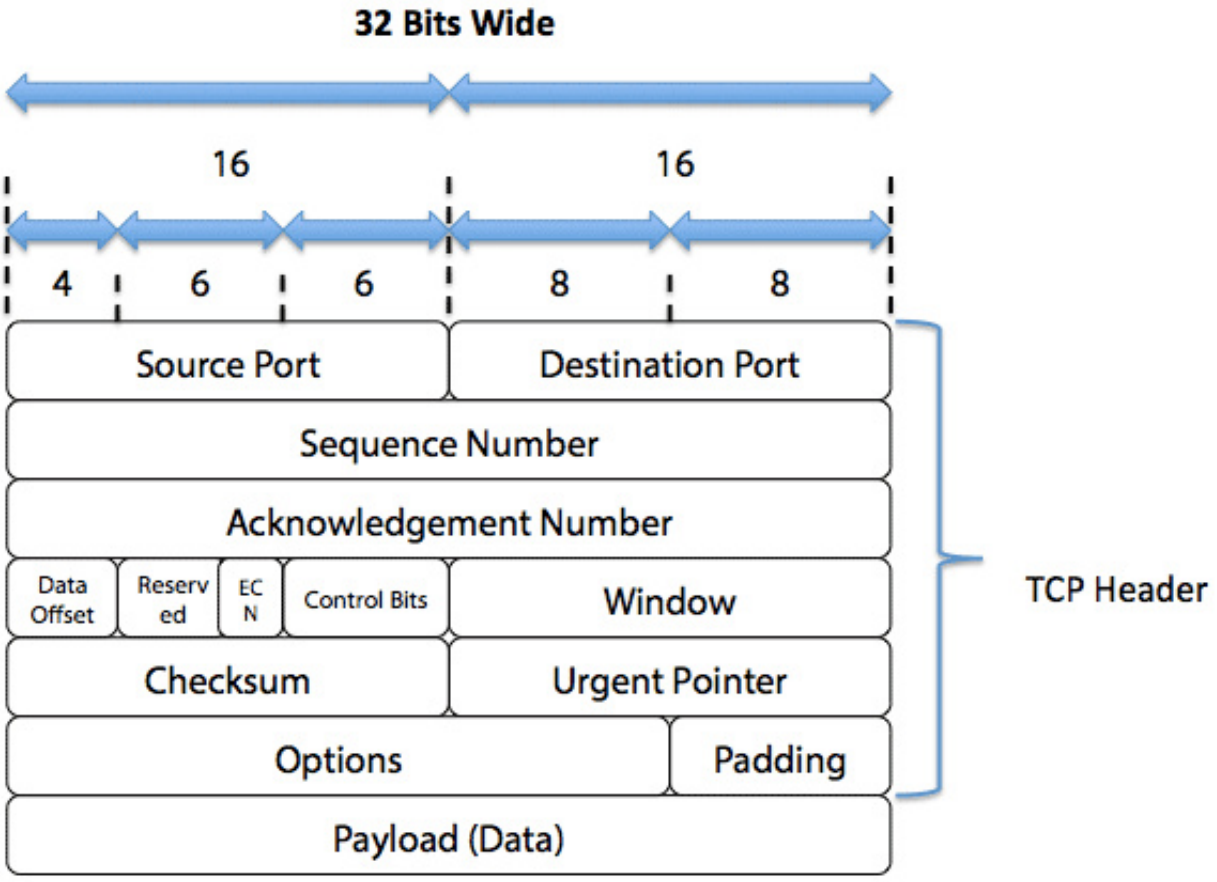

图2：TCP帧的结构

TCP数据包的payload部分会包含在socket发送缓冲区里的没有应答的数据。携带数据的最大长度是接收窗口、阻塞窗口和最大分段长度（maximum segment size,MSS）中的最大值。

之后会计算TCP校验值。在计算时，头信息（ip地址、分段长度和端口号）会包含在内。根据TCP状态可发送一个或多个数据包。

> 事实上，当前的网络栈使用了校验卸载（checksum offload），TCP校验和会由NIC计算，而不是内核。但是，为了解释方便我们还是假设内核计算校验和。

被创建的TCP分段继续走到下面的IP层。IP层向TCP分段中增加了IP头并且执行了IP路由（IP routing）。IP路由是寻找到达目的IP的下一跳IP地址的过程。

在IP层计算并增加了IP头校验和之后，它把数据发送到链路层。链路层通过地址解析协议（Address Resolution Protocol，ARP）搜索下一跳IP地址对应的MAC地址。之后它会向数据包中增加链路头，在增加链路头之后主机要发送的数据包就是完整的了。

在执行IP路由时，会选择一个传输接口（NIC）。接口被用于把数据包传送至下一跳IP。于是，用于发送的NIC驱动程序被调用了。

在这个时候，如果正在执行数据包捕获程序（例如tcpdump或wireshark）的话，内核将把数据包拷贝到这些程序的内存缓冲区中。用相同的方式，接收的数据包直接在驱动被捕获。通常来说，traffic shaper（没懂）函数被实现以在这个层上运行。

驱动程序（与内核）通过请求NIC制造商定义的通讯协议传输数据。

在接收到数据包传输请求之后，NIC把数据包从系统内存中拷贝到它自己的内存中，之后把数据包发送到网络上。在此时，由于要遵守以太网标准（Ethernet standard），NIC会向数据包中增加帧间隙（Inter-Frame Gap，IFG），同步码（preamble）和crc校验和。帧间隙和同步码用于区分数据包的开始（网络术语叫做帧，framing），crc用于保护数据（与TCP或者IP校

验和的目的相同）。NIC会基于以太网的物理速度和流量控制决定数据包开始传输的时间。It is like getting the floor and speaking in a conference room.（没看懂）

当NIC发送了数据包，NIC会在主机的CPU上产生中断。所有的中断会有自己的中断号，操作系统会用这个中断号查找合适的程序去处理中断。驱动程序在启动时会注册一个处理中断的函数。操作系统调用中断处理程序，之后中断处理程序会把已发送的数据包返回给操作系统。

到此为止我们讨论了当应用程序执行了write之后，数据流经内核和设备的过程。但是，除了应用程序直接调用write之外，内核也可以直接调用TCP传输数据包。比如当接收到ACK并且得知接收方的接收窗口增大之后，内核会创建包含socket缓冲区剩余数据的TCP片段并且把数据发送给接收者。

5. 数据接收

现在我们看一下数据是如何被接收的。数据接收是网络栈如何处理流入数据包的过程。图3展现了网络栈如何处理接收的数据包。

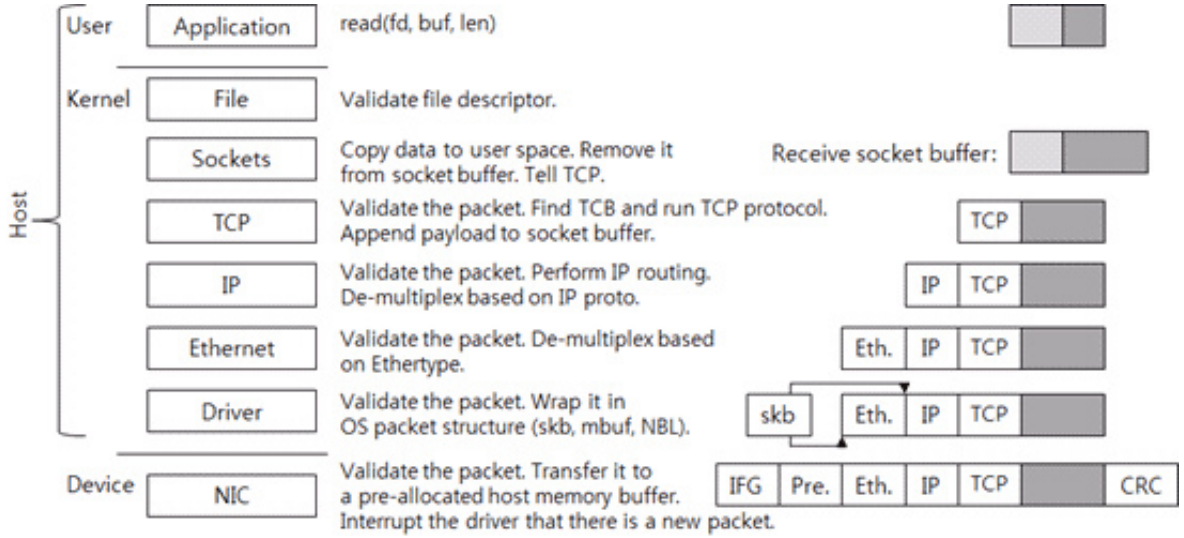

图3：接收数据时网络栈中各层对数据的操作

首先，NIC把数据包写入它自身的内存。通过CRC校验检查数据包是否有效，之后把数据包发送到主机的内存缓冲区。这里说的缓冲区是驱动程序提前向内核申请好的一块内存区域，用于存放接收的数据包。在缓冲区被系统分配之后，驱动会把这部分内存的地址和大小告知NIC。如果主机没有为驱动程序分配缓冲区，那么当NIC接收到数据包时有可能会直接丢弃它。

在把数据包发送到主机缓冲区之后，NIC会向主机发出中断。

之后，驱动程序会判断它是否能处理新的数据包。到目前为止使用的是由制造商定义的网卡驱动的通讯协议。

当驱动应该向上层发送数据包时，数据包必须被放进一个操作系统能够理解和使用的数据结构。例如Linux的sk_buff，或者BSD系列内核的mbuf，或者windows的NET_BUFFER_LIST。驱动会把数据包包装成指定的数据结构，并发送到上一层。

链路层会检查数据包是否有效并且解析出上层的协议（网络协议）。此时它会判断链路头中的以太网类型。（IPv4的以太网类型是0×0800）。它会把链路头删掉并且把数据包发送到IP层。

IP层同样会检查数据包是否有效。或者说，会检查IP头校验和。它会执行IP路由判断，判断是

由本机处理数据包还是把数据包发送到其它系统。如果数据包必须由本地系统处理，IP层会通过IP header中引用的原型值（proto value）解析上层协议（传输协议）。TCP原型值为6。系统会删除IP头，并且把数据包发送到TCP层。

就像之前的几层，TCP层检查数据包是否有效，同时会检查TCP校验和。就像之前提到的，如果当前的网络栈使用了校验卸载，那么TCP校验和会由NIC计算，而不是内核。

之后它会查找数据包对应的TCP控制块（TCB），这时会使用数据包中的<源ip，源端口，目的IP，目的端口>做标识。在查询到对应的连接之后，会执行协议中定义的操作去处理数据包。如果接收到的是新数据，数据会被增加到socket的接收缓冲区。根据tcp连接的状态，此时也可以发送新的TCP包（比如发送ACK包）。此时，TCP/IP接收数据包的处理完成。

socket接收缓冲区的大小就是TCP接收窗口。TCP吞吐量会随着接收窗口变大而增加。过去socket缓冲区大小是应用或操作系统的配置的定值。最新的网络栈使用了一个函数去自动决定接收缓冲区的大小。

当应用程序调用read系统调用时，程序会切换到内核区，并且会把socket接收缓冲区中的数据拷贝到用户区。拷贝后的数据会从socket缓冲区中移除。之后TCP会被调用，TCP会增加接收窗口的大小（因为缓冲区有了新空间）。并且会根据协议状态发送数据包。如果没有数据包传送，系统调用结束。

6. 网络栈开发方向

到此为止描述的网络栈中的函数都是最基础的函数。在1990年代早期的网络栈函数只比上面描述的多一些。但是最近的随着网络栈的实现层次变高，网络栈增加了很多函数和复杂度。

最新的网络栈可以按以下需求分类：

6.1. 操作报文处理过程

它包括类似Netfilter（firewall,NAT）的功能和流量控制。通过在处理流程中增加用户可以控制的代码，可以由用户控制实现不同的功能。

6.2. 协议性能

用于改进特定网络环境下的吞吐量、延迟和可靠性。典型例子是增加的流量控制算法和额外的类似SACK的TCP功能。由于已经超出了范围，在这里就不深入讨论协议改进了。

6.3. 高效的报文处理

高效的报文处理指的是通过降低处理报文的CPU周期、内存用量和处理报文需要访问内存的次数这些手段，来提升每秒可以处理的报文数量。有很多降低系统延迟的尝试，比如协议栈并行处理（stack parallel processing）、报头预处理（header prediction）、零拷贝（zero-copy）、单次拷贝（single-copy）、校验卸载（checksum offload）、TSO（TCP Segment Offload）、LRO（Large Receive Offload）、RSS（Receive Side Scaling）等。

7. 网络栈中的流程控制

现在我们看一下Linux网络栈内部流程的细节。网络栈基于事件驱动的方式运行，在事件产生

时做出相应的反应。因此，没有一个独立的线程去运行网络栈。图1和图3展示了最简单的控制流程图。图4说明了更加准确的控制流程。

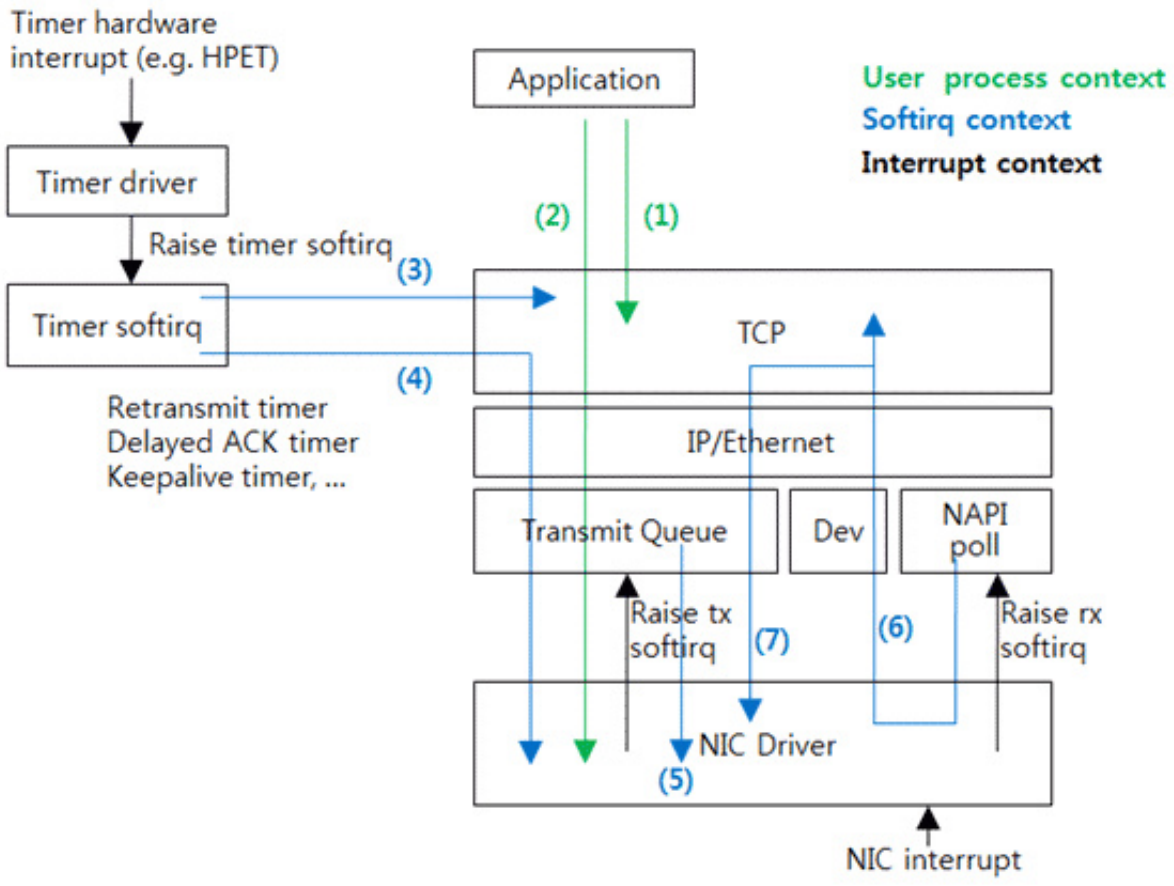

图4：网络栈中的流程控制

在图4中的Flow(1)，应用程序通过系统调用去执行（使用）TCP。例如，调用read系统调用或者wirte系统调用并执行TCP。但是，这一步里没有数据包传输。

Flow(2)跟Flow(1)类似，在执行TCP后需要传输报文。它会创建数据包并且把数据包发送给驱动程序。驱动程序上层有一个队列。数据包首先被放入队列，之后队列的数据结构决定何时把数据包发送给驱动程序。这个是Linux的队列规则（queue discipline,qdisc）。Linux的传输管理函数用来管理队列规则。默认的队列规则是简单的先入先出队列。通过使用其它的队列管理规则，可以执行多种操作，例如人造数据丢包、数据包延迟、通信比率限制，等等。在Flow(1)和Flow(2)中，驱动和应用程序处于相同的线程。

Flow(3)展示了TCP使用的定时器超时的情况。比如，当TIME_WAIT定时器超时后，TCP被调用并删除连接。

类似Flow(3)，Flow（4）是TCP的定时器超时并且需要发送TCP执行结果数据包的情况。比如，当重传计时器超时后，会发送“没有收到ACK”数据包。

Flow（3）和Flow（4）展示了定时器软中断的处理过程。

当NIC驱动收到中断时，它将释放已经传输的数据包。在大多数情况下，驱动的执行过程到这里就结束了。Flow（5）是数据包积累在传输队列里的情况。驱动请求软中断，之后软中断的处理程序把传输队列里堆积的数据包发送给驱动程序。

当NIC驱动程序接收到中断并且发现有新接收的数据包时，它会请求软中断。软中断会调用驱动程序处理接收到的数据包并且把他们传送到上一层。在Linux中，上面描述的处理接收到的数据包的过程叫做New API（NAPI）。它和轮询类似，因为驱动并不直接把数据包传送到上一

层，而是上层直接来取数据。实际代码中叫做NAPI poll 或 poll。

Flow（6）展示了TCP执行完成，Flow（7）展示了需要更多数据包传输的情况。Flow（5、6、7）的NIC中断都是通过软中断执行的。

8. 如何处理中断和接收数据包

中断处理过程是复杂的，但是你需要了解数据包接收和处理流程中的和性能相关的问题。图5展示了一次中断的处理流程。

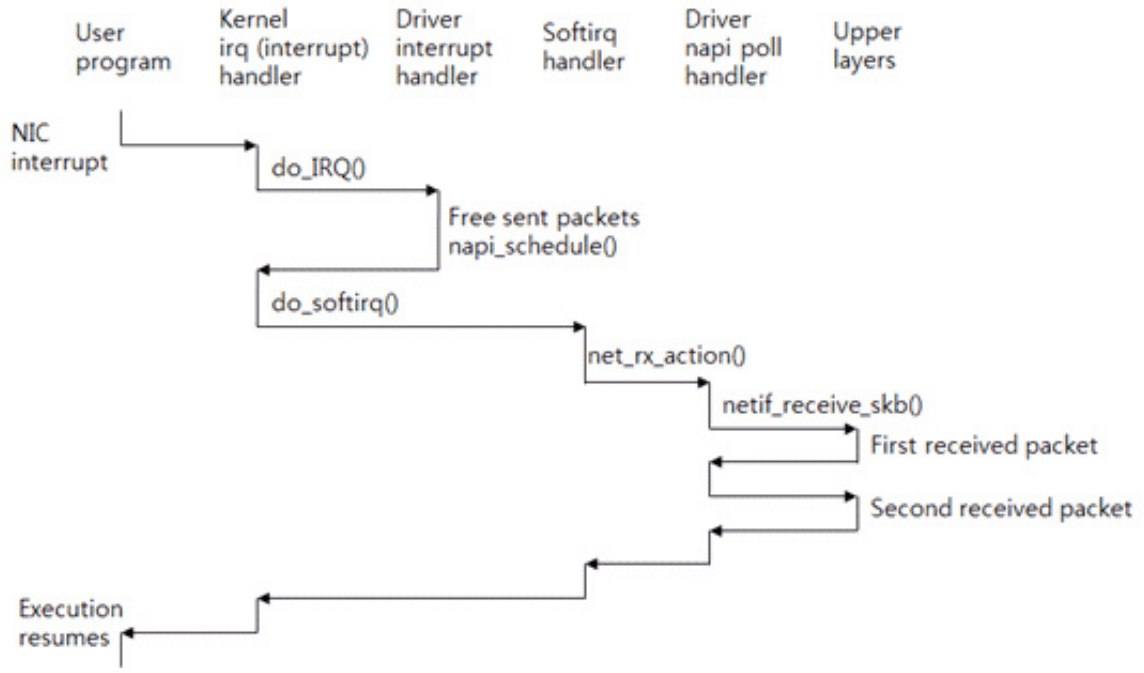

图5：处理中断、软中断和接收数据

假设CPU0正在执行应用程序。在此时，NIC接收到了一个数据包并且在CPU0上产生了中断。CPU0执行了内核中断处理程序(irq)。这个处理程序关联了一个中断号并且内核会调用驱动里对应的中断处理程序。驱动在释放已经传输的数据包之后调用napi_schedule()函数去处理接收到的数据包，这个函数会请求软中断。

在执行了驱动的中断处理程序后，控制权被转移到内核中断处理程序。内核中的处理程序会执行软中断的处理程序。

在中断上下文被执行之后，软中断的上下文会被执行。中断上下文和软中断上下文会通过相同的线程执行，但是它们会使用不同的栈。并且中断上下文会屏蔽硬件中断；而软中断上下文不会屏蔽硬件中断。

处理接收到的数据包的软中断处理程序是net_rx_action()函数。这个函数会调用驱动程序的poll()函数。而poll()函数会调用netif_receive_skb()函数，并把接收到的数据包一个接一个的发送到上层。在软中断处理结束后，应用程序会从停止的位置重新开始执行。（After processing the softirq, the application restarts execution from the stopped point in order to request a system call.没太明白这一句的system call是啥意思）

因此，接收中断请求的CPU会负责处理接收数据包从始至终的整个过程。在Linux、BSD和Windows中，处理过程基本是类似的。

当你检查服务器CPU利用率时，有时你可以检查服务器的很多CPU中是否只有一个CPU在艰难执行软中断。这个现象产生的原因就是上文所解释的数据包接收的处理过程。为了解决这个问题

开发出了多队列NIC(multi-queue NIC)、RSS和RPS。

下面是译者翻译的《理解TCP/IP网络栈&编写网络应用》的下篇，会通过讲解TCP的代码实现帮助大家理解发送、接收数据的流程，也描述了一些网卡、驱动等网络栈底层的原理。

8. 数据结构

以下是一些关键数据结构。我们了解一下这些数据结构再开始查看代码。

8.1. sk_buff_structure

首先，sk_buff结构或skb结构代表一个数据包。图6展现了sk_buff中的一些结构。随着功能变得更强大，它们也变得更复杂了。但是还是有一些任何人都能想到的基本功能。

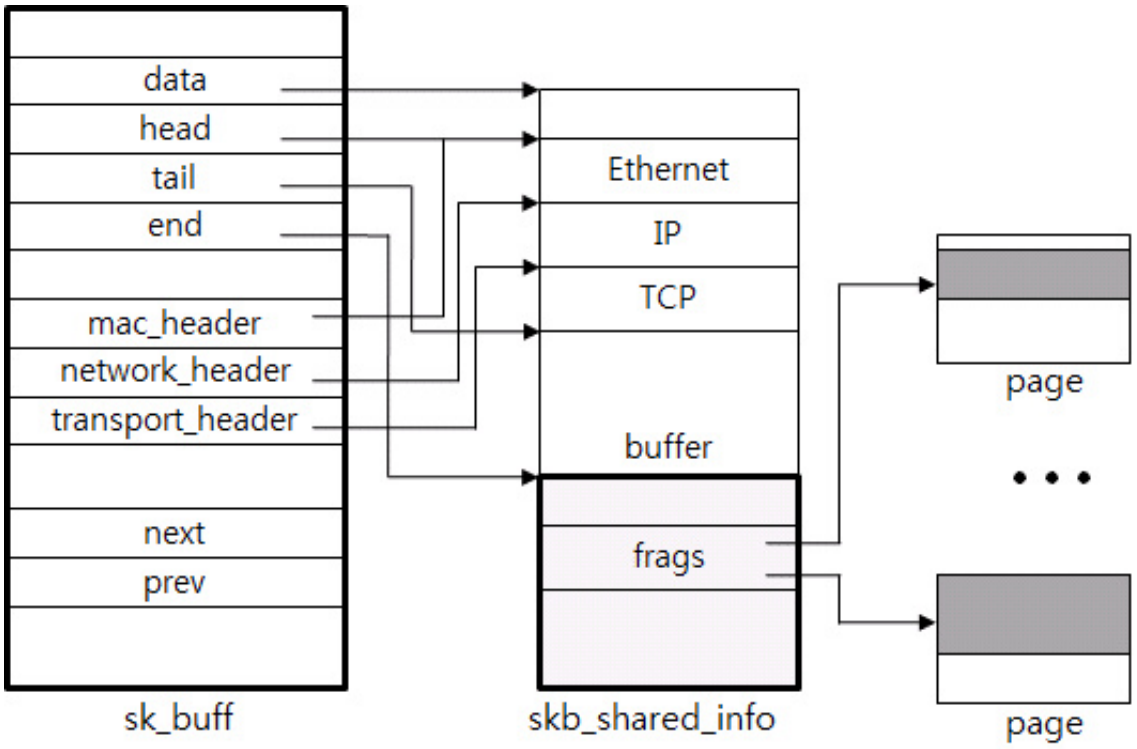

图6：数据包结构

8.1.1. 包含数据和元数据

这个结构直接包含或者通过指针引用了数据包。在图6中，一些数据包（从Ethernet到Buffer部分）使用了指针，一些额外的数据(frags）引用了实际的内存页。

一些必要的信息比如头和内容长度被保存在元数据区。例如，在图6中，mac_header、network_header和transport_header都有相应的指针，指向链路头、IP头和TCP头的起始地址。这种方式让TCP协议处理过程变得简单。

8.1.2. 如何增加或删除头

数据包在网络栈的各层中上升或下降时会增加或删除数据头。为了更有效率的处理而使用了指针。例如，要删除链路头只需要修改head pointer的值。

8.1.3. 如何合并或切分数据包

为了更有效率的执行把数据包增到或从socket缓冲区中删除这类操作而使用了链表，或者叫数据包链。next和prev指针用于这个场景。

8.1.4. 快速分配和释放

无论何时创建数据包都会分配一个数据结构，此时会用到快速分配器。比如，如果数据通过10Gb的以太网传输，每秒会有超过一百万个对象被创建和销毁。

9. TCP控制块（TCP Control Block）

其次，有一个表示TCP连接的数据结构，之前它被抽象的叫做TCP控制块。Linux使用了tcp_sock这个数据结构。在图7中，你可以看到文件、socket和tcp_socket的关系。

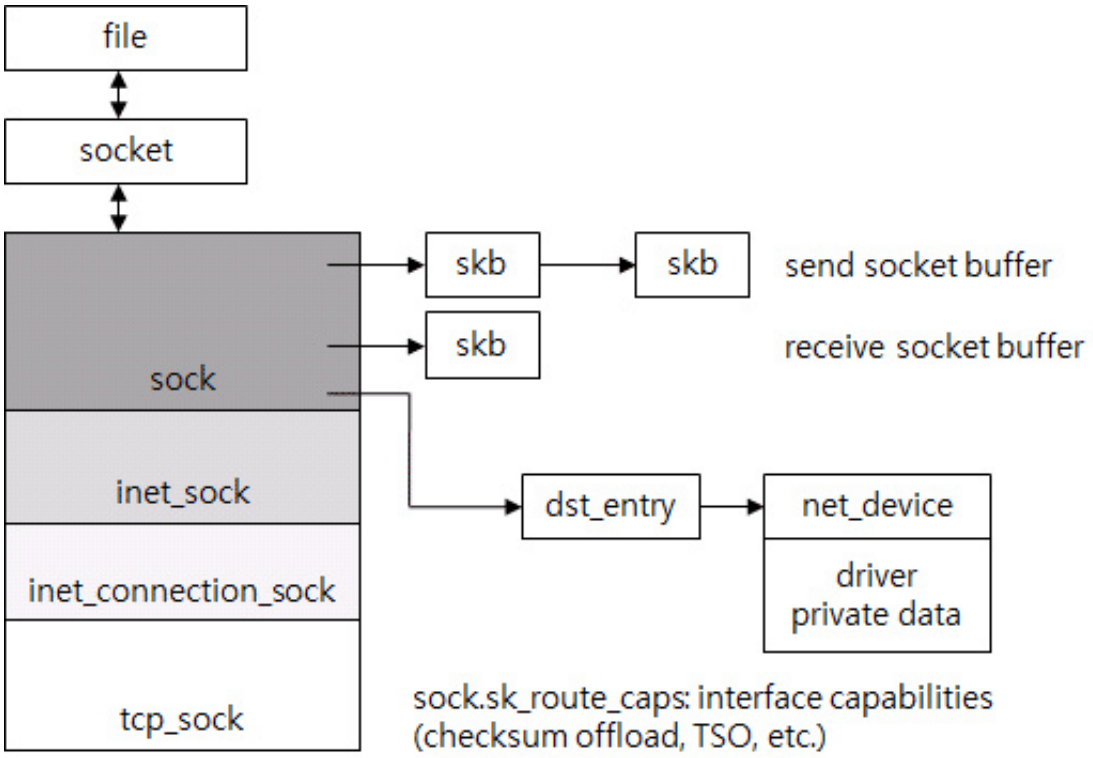

图7：TCP Connection结构

当系统调用发生后，它会找到应用程序在进行系统调用时使用的文件描述符对应的文件。对Unix系的操作系统来说，文件本身和通用文件系统存储的设备都被抽象成了文件。因此，文件结构包含了必要的信息。对于socket来说，使用独立的socket结构保存socket相关的信息，文件结构通过指针来引用socket。socket又引用了tcp_sock。tcp_sock可以分为sock,inet_sock等等,用来支持除了TCP之外的协议，可以认为这是一种多态。

所有TCP协议用到的状态信息都被存在tcp_sock里。例如顺序号、接收窗口、阻塞控制和重发送定时器都保存在tcp_sock中。

socket的发送缓冲区和接收缓冲区由sk_buff链表组成并被包含在tcp_sock中。为防止频繁查找路由，也会在tcp_sock中引用IP路由结果dst_entry。通过dst_entry可以简单的查找到目标的MAC地址之类的ARP的结果。dst_entry是路由表的一部分，而路由表是个很复杂的结构，在这篇文档里就不再讨论了。用来传送数据的网卡(NIC)可以通过dst_entry找到。网卡通过net_device描述。

因此，仅通过查找文件和指针就可以很简单的查找到处理TCP连接的所有的数据结构（从文件到驱动）。这个数据结构的大小就是每个TCP连接占用内存的大小。这个结构占用的内存只有几kb大小（排除了数据包中的数据）。但随着一些功能被加入，内存占用也在逐渐增加。

最后，我们来看一下TCP连接查找表（TCP connection lookup table）。这是一个用来查找接收到的数据包对应tcp连接的哈希表。系统会用数据包的<来源ip，目标ip，来源端口，目标端口>和Jenkins哈希算法去计算哈希值。选择这个哈希函数的原因是为了防止对哈希表的攻击。

10. 追踪代码：如何传输数据

我们将会通过追踪实际的Linux内核源码去检查协议栈中执行的关键任务。在这里，我们将会观察经常使用的两条路径。

首先是应用程序调用了write系统调用时的执行路径。

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, ...)

{

struct file *file;

[...]

file = fget_light(fd, &fput_needed);

[...] ===>

ret = filp->f_op->aio_write(&kiocb, &iov, 1, kiocb.ki_pos);

struct file_operations {

[...]

ssize_t (*aio_read) (struct kiocb *, const struct iovec *, ...)

ssize_t (*aio_write) (struct kiocb *, const struct iovec *, ...)

[...]

};

static const struct file_operations socket_file_ops = {

[...]

.aio_read = sock_aio_read,

.aio_write = sock_aio_write,

[...]

};
```

当应用调用了write系统调用时，内核将在文件层执行write()函数。首先，内核会取出文件描述符对应的文件结构体，之后会调用aio_write，这是一个函数指针。在文件结构体中，你可以看到file_perations结构体指针。这个结构被通称为函数表（function table），其中包含了一些函数的指针，比如aio_read或者aio_write。对于socket来说，实际的表是socket_file_ops，aio_write对应的函数是sock_aio_write。在这里函数表的作用类似于java中的interface，内核使用这种机制进行代码抽象或重构。

```
static ssize_t sock_aio_write(struct kiocb *iocb, const struct iovec *iov, ..)

{

[...]
```

```
struct socket *sock = file->private_data;

[...] ===>

return sock->ops->sendmsg(iocb, sock, msg, size);

struct socket {

[...]

struct file *file;

struct sock *sk;

const struct proto_ops *ops;

};

const struct proto_ops inet_stream_ops = {

.family = PF_INET,

[...]

.connect = inet_stream_connect,

.accept = inet_accept,

.listen = inet_listen, .sendmsg = tcp_sendmsg,

.recvmsg = inet_recvmsg,

[...]

};

struct proto_ops {

[...]

int (*connect) (struct socket *sock, ...)

int (*accept) (struct socket *sock, ...)

int (*listen) (struct socket *sock, int len);

int (*sendmsg) (struct kiocb *iocb, struct socket *sock, ...)

int (*recvmsg) (struct kiocb *iocb, struct socket *sock, ...)

[...]

};
```

sock_aio_write()函数会从文件结构体中取出socket结构体并调用sendmsg，这也是一个函数指针。socket结构体中包含了proto_ops函数表。IPv4的TCP实现中，proto_ops的具体实现是inet_stream_ops，sendmsg的实现是tcp_sendmsg。

```
int tcp_sendmsg(struct kiocb *iocb, struct socket *sock,

struct msghdr *msg, size_t size)

{

struct sock *sk = sock->sk;

struct iovec *iov;

struct tcp_sock *tp = tcp_sk(sk);

struct sk_buff *skb;

```

```
[...]

mss_now = tcp_send_mss(sk, &size_goal, flags);

/* Ok commence sending. */

iovlen = msg->msg_iovlen;

iov = msg->msg_iov;

copied = 0;

[...]

while (--iovlen >= 0) {

int seglen = iov->iov_len;

unsigned char __user *from = iov->iov_base;

iov++;

while (seglen > 0) {

int copy = 0;

int max = size_goal;

[...]

skb = sk_stream_alloc_skb(sk,

select_size(sk, sg),

sk->sk_allocation);

if (!skb)

goto wait_for_memory;

/*

* Check whether we can use HW checksum.

*/

if (sk->sk_route_caps & NETIF_F_ALL_CSUM)

skb->ip_summed = CHECKSUM_PARTIAL;

[...]

skb_entail(sk, skb);

[...]

/* Where to copy to? */

if (skb_tailroom(skb) > 0) {

/* We have some space in skb head. Superb! */

if (copy > skb_tailroom(skb))

copy = skb_tailroom(skb);

if ((err = skb_add_data(skb, from, copy)) != 0)

goto do_fault;

[...]

if (copied)

tcp_push(sk, flags, mss_now, tp->nonagle);
```

```
90
91  [...]
92
93  }
```

tcp_sendmsg（译注：原文写的是tcp_sengmsg，应该是笔误)会从socket中取得tcp_sock（也就是TCP控制块，TCB），并把应用程序请求发送的数据拷贝到socket发送缓冲中（译注：就是根据发送数据创建sk_buff链表）。当把数据拷贝到sk_buff中时，每个sk_buff会包含多少字节数据？在代码创建数据包时，每个sk_buff中会包含MSS字节(通过tcp_send_mss函数获取)，在这里MSS表示每个TCP数据包能携带数据的最大值。通过使用TSO(TCP Segment Offload)和GSO(Generic Segmentation Offload)技术，一个sk_buff可以保存大于MSS的数据。在这篇文章里就不详细解释了。

sk_stream_alloc_skb函数会创建新的sk_buff，之后通过skb_entail把新创建的sk_buff放到send_socket_buffer的末尾。skb_add_data函数会把应用层的数据拷贝到sk_buff的buffer中。通过重复这个过程（创建sk_buff然后把它加入到socket发送缓冲区）完成所有数据的拷贝。因此，大小是MSS的多个sk_buff会在socket发送缓冲区中形成一个链表。最终调用tcp_push把待发送的数据做成数据包，并且发送出去。

```
1   static inline void tcp_push(struct sock *sk, int flags, int mss_now, ...)
2
3   [...] ===>
4
5   static int tcp_write_xmit(struct sock *sk, unsigned int mss_now, ...)
6
7   int nonagle,
8
9   {
10
11  struct tcp_sock *tp = tcp_sk(sk);
12
13  struct sk_buff *skb;
14
15  [...]
16
17  while ((skb = tcp_send_head(sk))) {
18
19  [...]
20
21  cwnd_quota = tcp_cwnd_test(tp, skb);
22
23  if (!cwnd_quota)
24
25  break;
26
27  if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
28
29  break;
30
31  [...]
32
33  if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
34
35  break;
36
37  /* Advance the send_head. This one is sent out.
38
39  * This call will increment packets_out.
40
41  */
42
43  tcp_event_new_data_sent(sk, skb);
44
45  [...]
```

tcp_push函数会在TCP允许的范围内顺序发送尽可能多的sk_buff数据。首先会调用tcp_send_head取得发送缓冲区中第一个sk_buff，然后调用tcp_cwnd_test和tcp_send_wnd_test检查堵塞窗口和接收窗口，判断接收方是否可以接收新数据。之后调用tcp_transmit_skb函数来创建数据包。

```
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb,

int clone_it, gfp_t gfp_mask)

{

const struct inet_connection_sock *icsk = inet_csk(sk);

struct inet_sock *inet;

struct tcp_sock *tp;

[...]

if (likely(clone_it)) {

if (unlikely(skb_cloned(skb)))

skb = pskb_copy(skb, gfp_mask);

else

skb = skb_clone(skb, gfp_mask);

if (unlikely(!skb))

return -ENOBUFS;

}

[...]

skb_push(skb, tcp_header_size);

skb_reset_transport_header(skb);

skb_set_owner_w(skb, sk);

/* Build TCP header and checksum it. */

th = tcp_hdr(skb);

th->source = inet->inet_sport;

th->dest = inet->inet_dport;

th->seq = htonl(tcb->seq);

th->ack_seq = htonl(tp->rcv_nxt);

[...]

icsk->icsk_af_ops->send_check(sk, skb);

[...]

err = icsk->icsk_af_ops->queue_xmit(skb);

if (likely(err <= 0))

return err;

tcp_enter_cwr(sk, 1);

return net_xmit_eval(err);

}
```

tcp_transmit_skb会创建指定sk_buff的拷贝（通过pskb_copy），但它不会拷贝应用层发送的数据，而是拷贝一些元数据。之后会调用skb_push来确保和记录头部字段的值。send_check计算TCP校验和（如果使用校验和卸载checksum offload技术则不会做这一步计算）。最终调用queue_xmit把数据发送到IP层。IPv4中queue_xmit的实现函数是ip_queue_xmit。

```
int ip_queue_xmit(struct sk_buff *skb)

[...]

rt = (struct rtable *)__sk_dst_check(sk, 0);

[...]

/* OK, we know where to send it, allocate and build IP header. */

skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen : 0));

skb_reset_network_header(skb);

iph = ip_hdr(skb);

*((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));

if (ip_dont_fragment(sk, &rt->dst) && !skb->local_df)

iph->frag_off = htons(IP_DF);

else

iph->frag_off = 0;

iph->ttl = ip_select_ttl(inet, &rt->dst);

iph->protocol = sk->sk_protocol;

iph->saddr = rt->rt_src;

iph->daddr = rt->rt_dst;

[...]

res = ip_local_out(skb);

[...] ===>

int __ip_local_out(struct sk_buff *skb)

[...]

ip_send_check(iph);

return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT, skb, NULL,

skb_dst(skb)->dev, dst_output);

[...] ===>

int ip_output(struct sk_buff *skb)

{

struct net_device *dev = skb_dst(skb)->dev;

[...]

skb->dev = dev;

skb->protocol = htons(ETH_P_IP);

return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb, NULL, dev,

ip_finish_output,

[...] ===>

static int ip_finish_output(struct sk_buff *skb)

[...]

if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb))
```

```
76
77  return ip_fragment(skb, ip_finish_output2);
78
79  else
80
81  return ip_finish_output2(skb);
```

ip_queue_xmit函数执行IP层的一些必要的任务。__sk_dst_check检查缓存的路由是否有效。如果没有被缓存的路由项，或者路由无效，它将会执行IP路由选择（IP routing)。之后调用skb_push来计算和记录IP头字段的值。之后，随着函数执行，ip_send_check计算IP头校验和并且调用netfilter功能（译注：这是内核的一个模块）。如果使用ip_finish_output函数会创建IP数据分片，但在使用TCP协议时不会创建分片，因此内核会直接调用ip_finish_output2来增加链路头，并完成数据包的创建。

```
1   int dev_queue_xmit(struct sk_buff *skb)
2
3   [...] ===>
4
5   static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q, ...)
6
7   [...]
8
9   if (...) {
10
11  ....
12
13  } else
14
15  if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&
16
17  qdisc_run_begin(q)) {
18
19  [...]
20
21  if (sch_direct_xmit(skb, q, dev, txq, root_lock)) {
22
23  [...] ===>
24
25  int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q, ...)
26
27  [...]
28
29  HARD_TX_LOCK(dev, txq, smp_processor_id());
30
31  if (!netif_tx_queue_frozen_or_stopped(txq))
32
33  ret = dev_hard_start_xmit(skb, dev, txq);
34
35  HARD_TX_UNLOCK(dev, txq);
36
37  [...]
38
39  }
40
41  int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev, ...)
42
43  [...]
44
45  if (!list_empty(&ptype_all))
46
47  dev_queue_xmit_nit(skb, dev);
48
49  [...]
50
51  rc = ops->ndo_start_xmit(skb, dev);
52
53  [...]
54
55  }
```

最终的数据包会通过dev_queue_xmit函数完成传输。首先，数据包通过排队规则（译注：

qdisc，上篇文章简单介绍过）传递。如果使用了默认的排队规则并且队列是空的，那么会跳过队列而直接调用sch_direct_xmit把数据包发送到驱动。dev_hard_start_xmit会调用实际的驱动程序。在调用驱动之前，设备的发送（译注：TX，transmit）被锁定，防止多个线程同时使用设备。由于内核锁定了设备的发送，驱动发送数据相关的代码就不需要额外的锁了。这里下次要讲（译注：这里是说原作者的下篇文章）的并行开发有很大关系。

ndo_start_xmit函数会调用驱动的代码。在这之前，你会看到ptype_all和dev_queue_xmit_nit。ptype_all是个包含了一些模块的列表（比如数据包捕获）。如果捕获程序正在运行，数据包会被ptype_all拷贝到其它程序中。因此，tcpdump中显示的都是发送给驱动的数据包。当使用了校验和卸载(checksum offload)或TSO(TCP Segment Offload)这些技术时，网卡（NIC）会操作数据包，所以tcpdump得到的数据包和实际发送到网络的数据包有可能不一致。在结束了数据包传输以后，驱动中断处理程序会返回发送了的sk_buff。

11. 追踪代码：如何接收数据

一般来说，接收数据的执行路径是接收一个数据包并把数据加入到socket的接收缓冲区。在执行了驱动中断处理程序之后，首先执行的是napi poll处理程序。

```
static void net_rx_action(struct softirq_action *h)

{

struct softnet_data *sd = &__get_cpu_var(softnet_data);

unsigned long time_limit = jiffies + 2;

int budget = netdev_budget;

void *have;

local_irq_disable();

while (!list_empty(&sd->poll_list)) {

struct napi_struct *n;

[...]

n = list_first_entry(&sd->poll_list, struct napi_struct,

poll_list);

if (test_bit(NAPI_STATE_SCHED, &n->state)) {

work = n->poll(n, weight);

trace_napi_poll(n);

}

[...]

}

int netif_receive_skb(struct sk_buff *skb)

[...] ===>

static int __netif_receive_skb(struct sk_buff *skb)

{

struct packet_type *ptype, *pt_prev;

[...]

```

```
__be16 type;

[...]

list_for_each_entry_rcu(ptype, &ptype_all, list) {

if (!ptype->dev || ptype->dev == skb->dev) {

if (pt_prev)

ret = deliver_skb(skb, pt_prev, orig_dev);

pt_prev = ptype;

}

}

[...]

type = skb->protocol;

list_for_each_entry_rcu(ptype,

&ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {

if (ptype->type == type &&

(ptype->dev == null_or_dev || ptype->dev == skb->dev ||

ptype->dev == orig_dev)) {

if (pt_prev)

ret = deliver_skb(skb, pt_prev, orig_dev);

pt_prev = ptype;

}

}

if (pt_prev) {

ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);

static struct packet_type ip_packet_type __read_mostly = {

.type = cpu_to_be16(ETH_P_IP),

.func = ip_rcv,

[...]

};
```

就像之前说的，net_rx_action函数是用于接收数据的软中断处理函数。首先，请求napi poll的驱动会检索poll_list，并且调用驱动的poll处理程序(poll handler)。驱动会把接收到的数据包包装成sk_buff，之后调用netif_receive_skb。

如果有模块在请求数据包，那么netif_receive_skb会把数据包发送给那个模块。类似于之前讨论过的发送的过程，在这里驱动接收到的数据包会发送给注册到ptype_all列表的那些模块，数据包在这里被捕获。

之后，根据数据包的类型，不同数据包会被传输到相应的上层。链路头中包含了2字节的以太网类型(ethertype)字段，这个字段的值标识了数据包的类型。驱动会把这个值记录到sk_buff中（skb->protocol)。每一种协议有自己的packet_type结构体，并且会把指向结构体的指针放入ptype_base哈希表中。IPv4使用的是ip_packate_type，类型字段中的值是IPv4类型(ETH_P_IP）。于是，对于IPv4类型的数据包会调用ip_recv函数。

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, ...)

{

struct iphdr *iph;

u32 len;

[...]

iph = ip_hdr(skb);

[...]

if (iph->ihl < 5 || iph->version != 4)

goto inhdr_error;

if (!pskb_may_pull(skb, iph->ihl*4))

goto inhdr_error;

iph = ip_hdr(skb);

if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))

goto inhdr_error;

len = ntohs(iph->tot_len);

if (skb->len < len) {

IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INTRUNCATEDPKTS);

goto drop;

} else if (len < (iph->ihl*4))

goto inhdr_error;

[...]

return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,

ip_rcv_finish);

[...] ===>

int ip_local_deliver(struct sk_buff *skb)

[...]

if (ip_hdr(skb)->frag_off & htons(IP_MF | IP_OFFSET)) {

if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))

return 0;

}

return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,

ip_local_deliver_finish);

[...] ===>

static int ip_local_deliver_finish(struct sk_buff *skb)

[...]

__skb_pull(skb, ip_hdrlen(skb));

[...]

int protocol = ip_hdr(skb)->protocol;
```

```
int hash, raw;

const struct net_protocol *ipprot;

[...]

hash = protocol & (MAX_INET_PROTOS - 1);

ipprot = rcu_dereference(inet_protos[hash]);

if (ipprot != NULL) {

[...]

ret = ipprot->handler(skb);

[...] ===>

static const struct net_protocol tcp_protocol = {

.handler = tcp_v4_rcv,

[...]

};
```

ip_rcv函数执行IP层必要的操作。它会解析数据包中的长度和IP头校验和。在经过netfilter代码时会执行ip_local_deliver函数。如果需要的话还会装配IP分片。之后会通过netfilter的代码调用ip_local_deliver_finish函数，这个函数通过调用__skb_pull移除IP头，并且通过IP头的protocol值查找上层协议标记。类似于ptype_base，每个传输层协议会在inet_protos中注册自己的net_protocol结构体。IPv4 TCP使用tcp_protocol，于是会调用tcp_v4_rcv函数继续处理。

当数据包到达TCP层时，数据包的处理流程取决于TCP状态和包类型。在这里，我们将看到TCP连接处于ESTABLISHED状态时处理到达的数据包的过程。在没有出现出现丢包或者乱序等异常的情况下，服务器会频繁的执行这条路径。

```
int tcp_v4_rcv(struct sk_buff *skb)

{

const struct iphdr *iph;

struct tcphdr *th;

struct sock *sk;

[...]

th = tcp_hdr(skb);

if (th->doff < sizeof(struct tcphdr) / 4)

goto bad_packet;

if (!pskb_may_pull(skb, th->doff * 4))

goto discard_it;

[...]

th = tcp_hdr(skb);

iph = ip_hdr(skb);

TCP_SKB_CB(skb)->seq = ntohl(th->seq);

TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +

skb->len - th->doff * 4);

```

```
TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);

TCP_SKB_CB(skb)->when = 0;

TCP_SKB_CB(skb)->flags = iph->tos;

TCP_SKB_CB(skb)->sacked = 0;

sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);

[...]

ret = tcp_v4_do_rcv(sk, skb);
```

首先，tcp_v4_rcv函数验证包的有效性，如果tcp头的大小比数据的偏移大时(th->doff < sizeof(struct tcphdr) /4)则说明包头有错误。（如果没有错误）之后会调用__inet_lookup_skb在存放TCP连接的哈希表里查找数据包所属的连接。从查找到的sock结构体可以找到所有需要的数据结构（比如tcp_sock），也可以取得对应的socket。

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)

[...]

if (sk->sk_state == TCP_ESTABLISHED) {
/* Fast path */

sock_rps_save_rxhash(sk, skb->rxhash);

if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {

[...] ===>

int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,

[...]

/*

* Header prediction.

*/

if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&

TCP_SKB_CB(skb)->seq == tp->rcv_nxt &&

!after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt))) {

[...]

if ((int)skb->truesize > sk->sk_forward_alloc)

goto step5;

NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPHITS);

/* Bulk data transfer: receiver */

__skb_pull(skb, tcp_header_len);

__skb_queue_tail(&sk->sk_receive_queue, skb);

skb_set_owner_r(skb, sk);

tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;

[...]

if (!copied_early || tp->rcv_nxt != tp->rcv_wup)

__tcp_ack_snd_check(sk, 0);

[...]
```

```
55
56  step5:
57
58  if (th->ack && tcp_ack(sk, skb, FLAG_SLOWPATH) < 0)
59
60  goto discard;
61
62  tcp_rcv_rtt_measure_ts(sk, skb);
63
64  /* Process urgent data. */
65
66  tcp_urg(sk, skb, th);
67
68  /* step 7: process the segment text */
69
70  tcp_data_queue(sk, skb);
71
72  tcp_data_snd_check(sk);
73
74  tcp_ack_snd_check(sk);
75
76  return 0;
77
78  [...]
79
    }
```

实际的协议在tcp_v4_do_rcv函数中处理。如果TCP正处于ESTABLISHED状态则会调用tcp_rcv_established（译注：原文写的是tcp_rcv_esablished，应该是笔误）。由于ESTABLISHED是最常用的状态，所以它被单独处理和优化了。tcp_rcv_established首先会执行头部预测（header prediction)的代码。头部预测会快速检测和处理常见情况。在这里常见的情况是没有在发送数据，实际收到的数据包正是应该收到的下一个数据包，例如TCP接收到了它期望收到的那个顺序号。在这种情况下在把数据加入到socket缓冲区、发送ack之后完成处理。

往前翻的话你会看到比较truesize和sk_forward_allow的语句。这个用来检查socket接收缓冲区中是否还有空闲空间来存放刚收到的数据。如果有的话，头部预测为“命中”（预测成功）。之后调用__skb_pull来删除TCP头。再之后调用__skb_queue_tail来把数据包增加到socket接收缓冲区。最终会视情况调用__tcp_ack_snd_check发送ACK。此时处理完成。

如果没有足够的空间，那么会走一条比较慢的路径。tcp_data_queue函数会新分配socket接收缓冲区的空间并把数据增加到缓冲区中。在这种情况下，接受缓冲区的大小在需要时会自动增加。和上一段说的快速路径不同，此时会在可行的情况下会调用tcp_data_snd_check发送一个新的数据回包。最终，如果需要的话会调用tcp_ack_snd_check来发送一个ACK包。

这两条路径下执行的代码量不大，因为这里讨论的都是通常情况。或者说，其它特殊情况的处理会慢的多，比如接收乱序这类情况。

12. 驱动和网卡如何通讯

驱动（driver）和网卡（nic）之间的通讯处于协议栈的底层，大多数人并不关心。但是，为了解决性能问题，网卡会处理越来越多的任务。理解基础的处理方式会帮助你理解额外这些优化技术。

网卡和驱动之间使用异步通讯。首先，驱动请求数据传输时CPU不会等待结果而是会继续处理其它任务，之后网卡发送数据包并通知CPU，驱动程序返回通过事件接收的数据包（这些数据包可以看作是异步发送的返回值）。

和数据包传输类似，数据包的接收也是异步的。首先驱动请求接受接收数据包然后CPU去执行其它任务，之后网卡接收数据包并通知CPU，然后驱动处理接收到的数据包（返回数据）。

因此需要有一个空间来保存请求和响应（request and response）。大多数情况网卡会使用环形队列数据结构（ring structure）。环形队列类似于普通的队列，其中有固定数量的元素，每个元素会保存一个请求或一个相应数据。环形队列中元素是顺序的，“环形”的意思是队列虽然是定长的，但是其中的元素会按顺序重用。

图8展示了数据包传输的过程，你会看到如何使用环形队列。

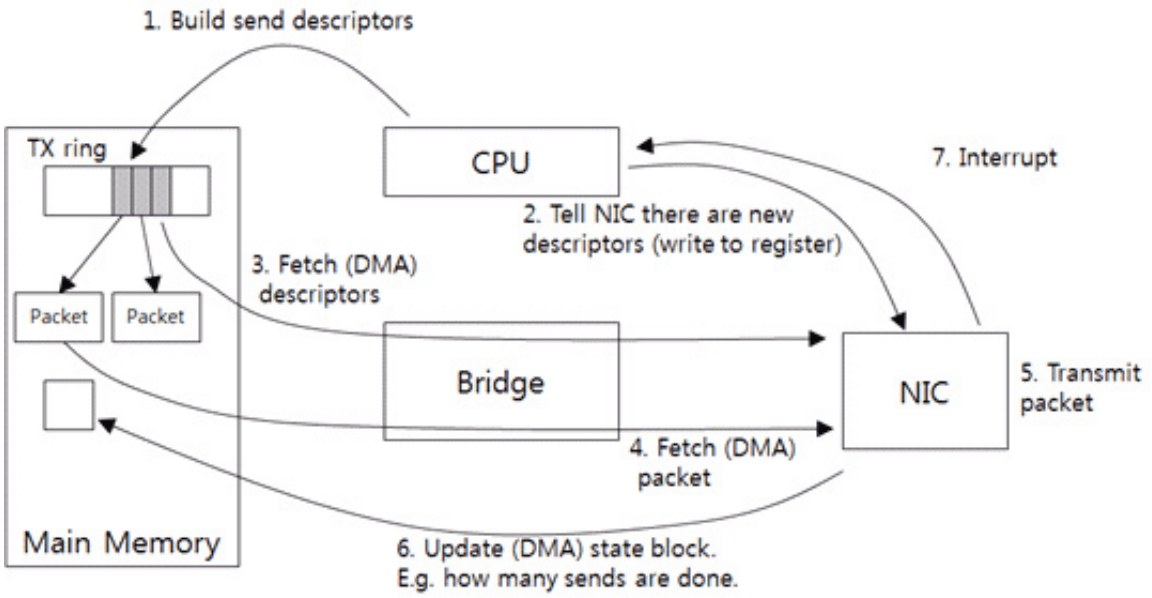

图8：驱动和网卡之间的通讯：如何传输数据包

驱动接收到上层发来的数据包并创建网卡能够识别的描述符。发送描述符（send descriptor）默认会包含数据包的大小和内存地址。这里网卡需要的是内存中的物理地址，驱动需要把数据包的虚拟地址转换成物理地址。之后，驱动会把发送描述符添加到发送环形队列（TX ring）（1），发送环形队列中包含的实际是发送描述符。

之后，驱动会把请求通知给网卡（2）。驱动直接把数据写入指定的网卡内存地址中（译注：这里应该是网卡的寄存器地址，写入的是通知而不是数据包）。这种CPU直接向设备发送数据的传输方式叫做程序化I/O（Programmed I/O，PIO）。

网卡被通知后从主机的发送队列中取得发送描述符（3）。这种设备直接访问内存而不需要调用CPU的内存访问方式叫做直接内存访问（Direct Memory Access，DMA）。

在取得发送描述符后，网卡会得到数据包的地址和大小并且从主机内存中取得实际的数据包（4）。如果有校验和卸载（checksum offload）的话，网卡会在拿到数据后计算数据包的校验和，因此开销不大。

网卡发送数据包（5）之后把发送数据包的数量写入主机内存（6）。之后它会触发一次中断，驱动程序会读取发送数据包的数量并根据数量返回已发送的数据包。

在图9中，你会看到接收数据包的流程。

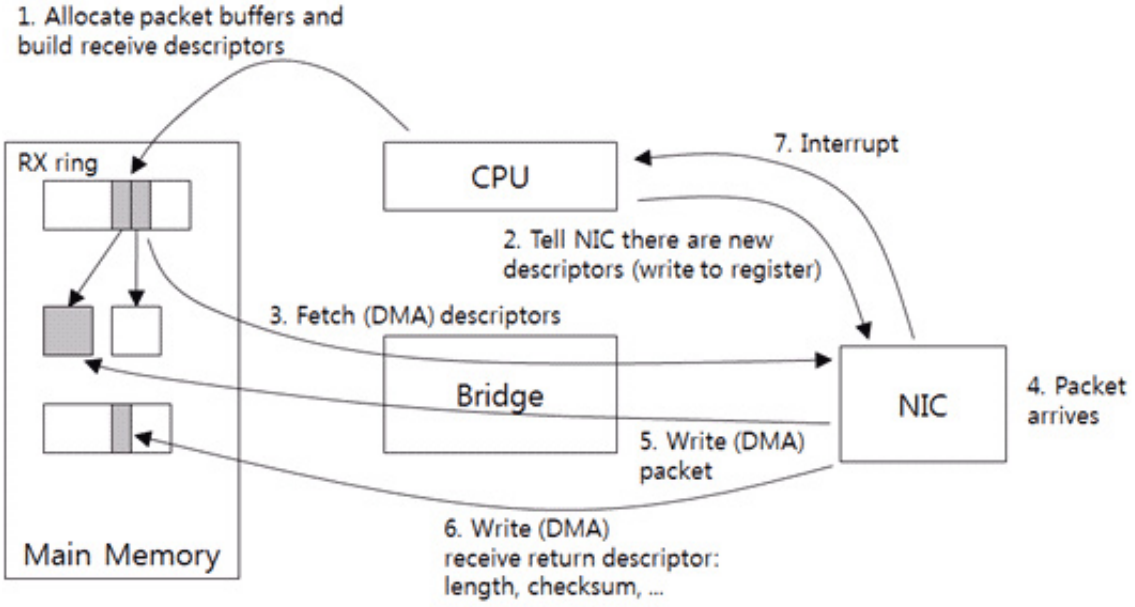

图9：驱动和网卡之间的通讯：如何接收数据包

首先，驱动会在主机内存中分配一块缓冲区用来接收数据包，之后创建接收描述符（receive descriptor）。接收描述符默认会包含缓冲区的大小和地址。类似于发送描述符，接收描述符中保存的也是DMA使用的物理地址。之后，驱动会把接收描述符加入到接收环形队列（RX ring）（1），接收环形队列保存的都是接收数据的请求。

通过PIO，驱动程序通知网卡有一个新的描述符（2），网卡会把新的描述符从接收队列中取出来，然后把缓冲区的大小和地址保存到网卡内存中（3）。

在接收到数据包时，网卡会把数据包发送到主机内存缓冲区中（5）。如果存在校验和卸载函数，那么网卡会在此时计算校验和。接收数据包的实际大小、校验和结果和其他信息会保存在独立的环形队列中（接收返回环形队列，receive return ring）（6）。接收返回队列保存了接收请求的处理结果，或者叫响应（response）。之后网卡会发出一个中断（7）。驱动程序从接收返回队列中获取接收到的数据包的信息，如果必要的话，还会分配新内存缓冲区并重复（1）和（2）。

调优网络栈时，大部分人会说环形队列和中断的设置需要被调整。当发送环形队列很大时，很多次的发送请求可以一次完成；当接收环形队列很大，可以一次性接收多个数据包。更大的环形队列对于大流量数据包接收/发送是很有用的。由于CPU在处理中断时有大量开销，大量大多数情况下，网卡使用一个计时器来减少中断。为了避免对宿主机过多的中断，发送和接收数据包的时候中断会被收集起来并且定期调用（interrupt coalescing，中断聚合）。

13. 网络栈中的缓冲区和流量控制

在网络栈中流量控制在几个阶段被执行。图10展示了传输数据时的一些缓冲区。首先，应用会创建数据并把数据加入到socket发送缓冲区。如果缓冲区中没有剩余空间的话，系统调用会失败或阻塞应用进程。因此，应用程序到内核的发送速率由socket缓冲区大小来限制。

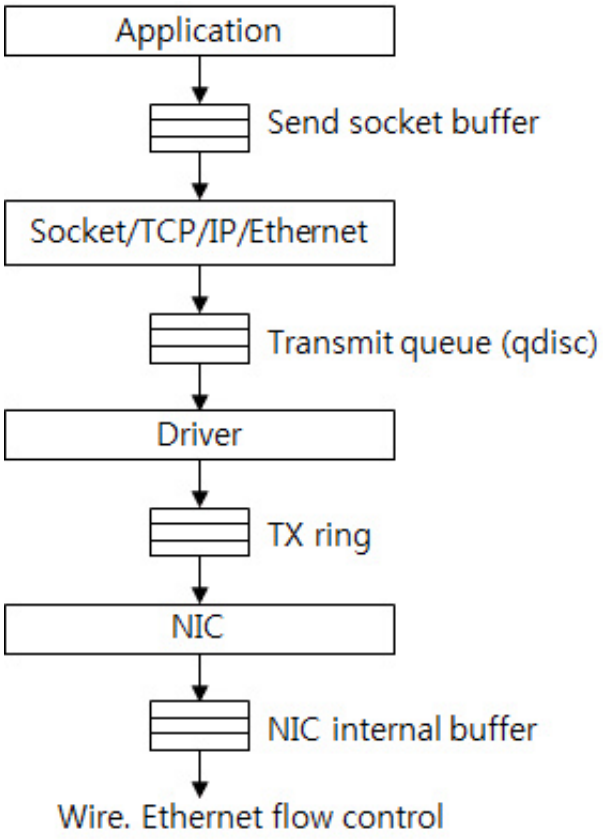

图10：数据发送相关的缓冲

TCP通过传输队列（qdisc）创建并把数据包发送给驱动程序。这是一个典型的先入先出队列，队列最大长度是txqueuelen，可以通过ifconfig命令来查看实际大小。通常来说，大约有几千个数据包。

驱动和网卡之间是传输环形队列（TX ring），它被认为是传输请求队列（transmission request queue）。如果队列中没有剩余空间的话就不会再继续创建传输请求，并且数据包会积累在传输队列中，如果数据包积累的太多，那么新的数据包会被丢弃。

网卡会把要发送的数据包保存在内部缓冲区中。这个队列中的数据包速度受网卡物理速度的影响（例如，1Gb/s的网卡不能承担10Gb/s的性能）。根据以太网流量控制，当网卡的接收缓冲区没有空间时，数据包传输会被停止。

当内核速度大于网卡时，数据包会堆积在网卡的缓冲区中。如果缓冲区中没有空间时会停止处理传输环形队列（TX ring）。越来越多的请求堆积在传输环形队列中，最终队列中空间被耗尽。驱动程序不能再继续创建传输请求数据包会堆积在传输队列（transmit queue）中。压力通过各种缓冲从底向上逐级反馈。

图11展示了接收数据包经过的缓冲区。数据包先被保存在网卡的接收缓冲区中。从流量控制的视角来看，驱动和网卡之间的接收环形缓冲区(RX ring)可以被看作是数据包的缓冲区。驱动程序从环形缓冲区取得数据包并把它们发送到上层。服务器系统的网卡驱动默认会使用NAPI，所以在驱动和上层之间没有缓冲区。因此，可以认为上层直接从接收环形缓冲区中取得数据，数据包的数据部分被保存在socket的接收缓冲区中。应用程序从socket接收缓冲区取得数据。

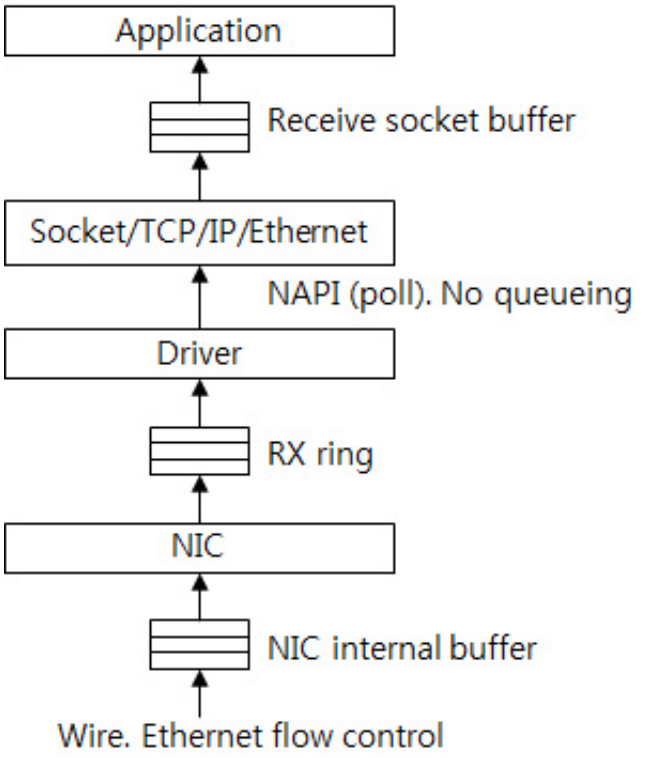

图11：与接收数据包相关的缓冲

不支持NAPI的驱动程序会把数据包保存在积压队列（backlog queue）中。之后，NAPI处理程序取得数据包，因此积压队列可以被认为是在驱动程序和上层之间的缓冲区。

如果内核处理数据包的速度低于网卡的速度，接收循环缓冲区队列(RX ring)会被写满，网卡的缓冲区空间(NIC internal buffer)也会被写满，当使用了以太流量控制（Ethernet flow control）时，接收方网卡会向发送方网卡发送请求来停止传输或丢弃数据包。

因为TCP支持端对端流量控制，所以不会出现由于socket接收队列空间不足而丢包的情况。但是，当使用UDP协议时，因为UDP协议不支持流量控制，如果应用程序处理速度不够的时候会出现socket接收缓冲区空间不足而丢包的情况。

在图10和图11中展示的传输环形队列（TX ring）和接收环形队列（RX ring）的大小可以用ethtool查看。在大多数看重吞吐量的负载情况下，增加环形队列的大小和socket缓冲区大小会有一些帮助。增加大小会减少高速收发数据包时由于缓冲区空间不足而造成的异常。

14. 结论

最初，我计划只解释那些会对你编写网络程序有帮助的东西，执行性能测试和解决问题。即使是我最初的计划，在这篇文档中包含的内容也会很多。我希望这篇文章会帮助你开发网络程序并监控它们的性能。TCP/IP协议本身就十分复杂并有很多异常情况。但是，你不需要理解OS中TCP/IP协议相关的每一行代码来理解性能和分析现象。只需要理解它的上下文对你就会十分有帮助。

随着系统性能和操作系统网络栈实现的持续提升，最近的服务器能承受10-20Gb/s的TCP吞吐量而不出现任何问题。近期也出现了与性能相关的很多的新技术，例如TSO, LRO, RSS, GSO, GRO, UFO, XPS, IOAT, DDIO, TOE等等（译注：这些我就不翻译了……），让我们有些困惑。

在下篇文章中，我会从性能的观点解释关于网络栈，并且讨论这些技术的问题和收益。

By Hyeongyeop Kim, Senior Engineer at Performance Engineering Lab, NHN

Corporation.

15. 译者最后说几句

本来看着文章内容不多，但是实际翻译下来才发现有这么多字……翻译过程中发现有很多地方自己之前的理解也有一些错误，如果各位发现哪里翻译的有问题的话麻烦告知一下，感激不尽~~

对于TCP协议来说，虽然已经看过很多相关的文章，但是真正要完整的描述一次tcp数据包发送和接收的过程实际上还是很难做到的。对于TCP或者内核之类的文章很多，但是我观察了一下，大部分人的问题并不是看的文章不够多，而是看的不够仔细。

与其每天乐此不疲的mark一堆技术文章然后随便扫几眼，还不如安心把一篇文章完整的读透，甚至再退一步，把文章从头到尾通读一遍，那也是极好的。这也是这次翻译这篇文章的初衷，与各位共勉。

1 赞 6 收藏 评论

关于作者：蛋疼的axb

秦迪，新浪微博平台及大数据部技术专家。负责微博平台通讯系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中... 个人主页 · 我的文章 · 2 ·

33

相关文章

- TCP协议疑难杂症全景解析
- TCP数据重传时间细节探秘及数据中心优化
- 从TCP协议的原理来谈谈RST复位攻击
- 互联网协议入门（一）
- 通信协议——HTTP、TCP、UDP
- 互联网协议入门（二）
- 游戏服务器：到底使用UDP还是TCP
- 对TCP/IP网络协议的深入浅出归纳
- 设计模式六大原则（3）：依赖倒置原则
- Linux中的两种文件锁—协同锁与强制锁

可能感兴趣的话题