什么是”零拷贝”

为了更好的理解问题的解决法，我们首先需要理解问题本身。首先我们以一个网络服务守护进程为例，考虑它在将存储在文件中的信息通过网络传送给客户这样的简单过程中，所涉及的操作。下面是其中的部分简单代阿：

read(file, tmp_buf, len);

write(socket, tmp_buf, len);

看起来不能更简单了。你也许认为执行这两个系统调用并未产生多少开销。实际上，这简直错的一塌糊涂。在执行这两个系统调用的过程中，目标数据至少被复制了4次，同时发生了同样多次数的用户/内核空间的切换(实际上该过程远比此处描述的要复杂，但是我希望以简单的方式描述之，以更好的理解本文的主题)。

为了更好的理解这两句代码所涉及的操作，请看图1。图的上半部展示了上下文切换，而下半部展示了复制操作。

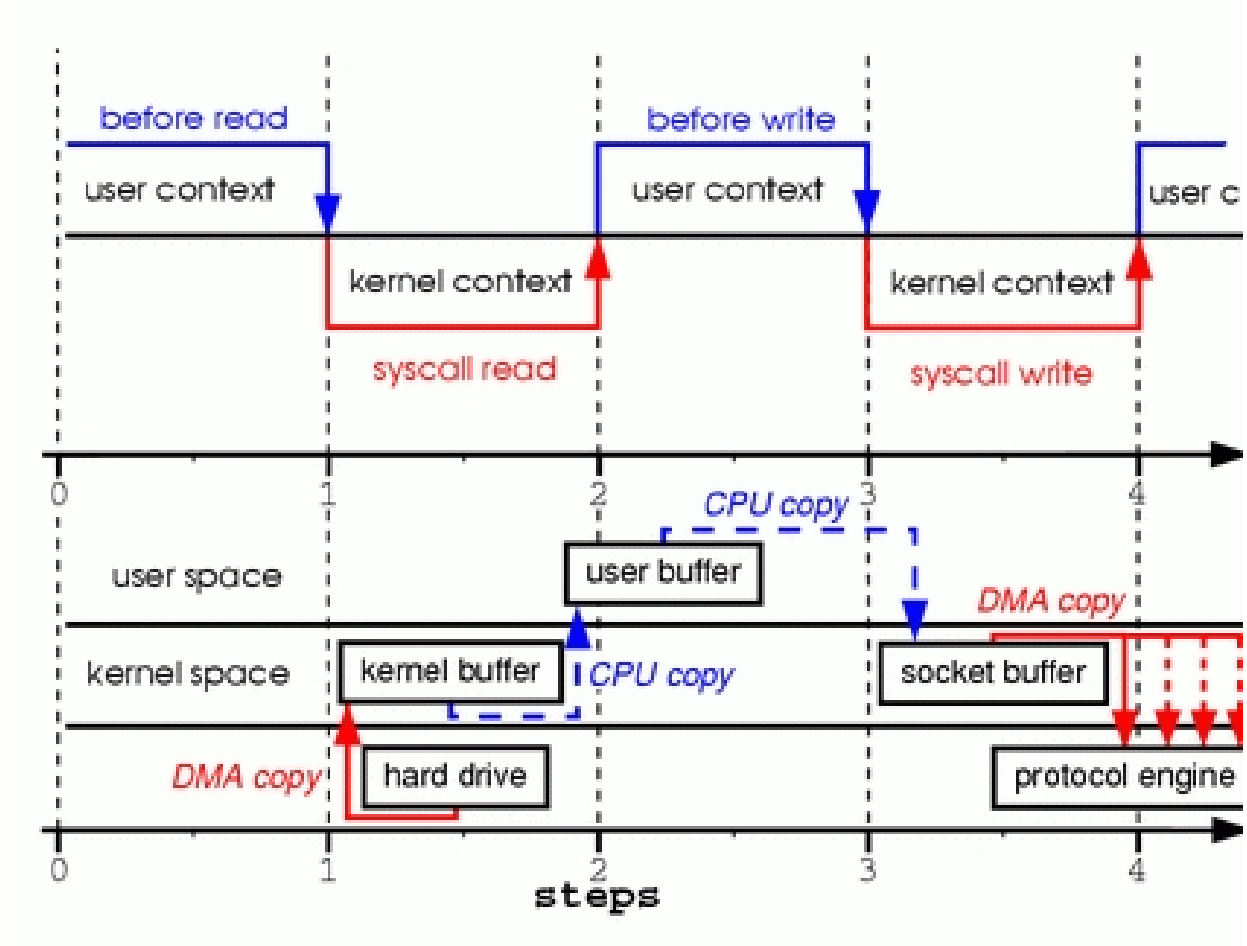

图1. Copying in Two Sample System Calls

步骤一：系统调用read导致了从用户空间到内核空间的上下文切换。DMA模块从磁盘中读取文件内容，并将其存储在内核空间的缓冲区内，完成了第1次复制。

步骤二：数据从内核空间缓冲区复制到用户空间缓冲区，之后系统调用read返回，这导致了从内核空间向用户空间的上下文切换。此时，需要的数据已存放在指定的用户空间缓冲区内(参数tmp_buf)，程序可以继续下面的操作。

步骤三：系统调用write导致从用户空间到内核空间的上下文切换。数据从用户空间缓冲区被再次复制到内核空间缓冲区，完成了第3次复制。不过，这次数据存放在内核空间中与使用的socket相关的特定缓冲区中，而不是步骤一中的缓冲区。

步骤四：系统调用返回，导致了第4次上下文切换。第4次复制在DMA模块将数据从内核空间缓冲区传递至协议引擎的时候发生，这与我们的代码的执行是独立且异步发生的。你可能会疑惑： “为何要说是独立、异步？难道不是在write系统调用返回前数据已经被传送了？write系统调用的返回，并不意味着传输成功——它甚至无法保证传输的开始。调用的返回，只是表明以太网驱动程序在其传输队列中有空位，并已经接受我们的数据用于传输。可能有众多的数据排在我们的数据之前。除非驱动程序或硬件采用优先级队列的方法，各组数据是依照FIFO的次序被传输的(图1中叉状的DMA copy表明这最后一次复制可以被延后)。

正如你所看到的，上面的过程中存在很多的数据冗余。某些冗余可以被消除，以减少开销、提高性能。作为一名驱动程序开发人员，我的工作围绕着拥有先进特性的硬件展开。某些硬件支持完全绕开内存，将数据直接传送给其他设备的特性。这一特性消除了系统内存中的数据副本，因此是一种很好的选择，但并不是所有的硬件都支持。此外，来自于硬盘的数据必须重新打包(地址连续)才能用于

网络传输，这也引入了某些复杂性。为了减少开销，我们可以从消除内核缓冲区与用户缓冲区之间的复制入手。

消除复制的一种方法是将read系统调用，改为mmap系统调用，例如：

```
tmp_buf = mmap(file, len);
write(socket, tmp_buf, len);
```

为了更好的理解这其中设计的操作，请看图2。上下文切换部分与图1保持一致。

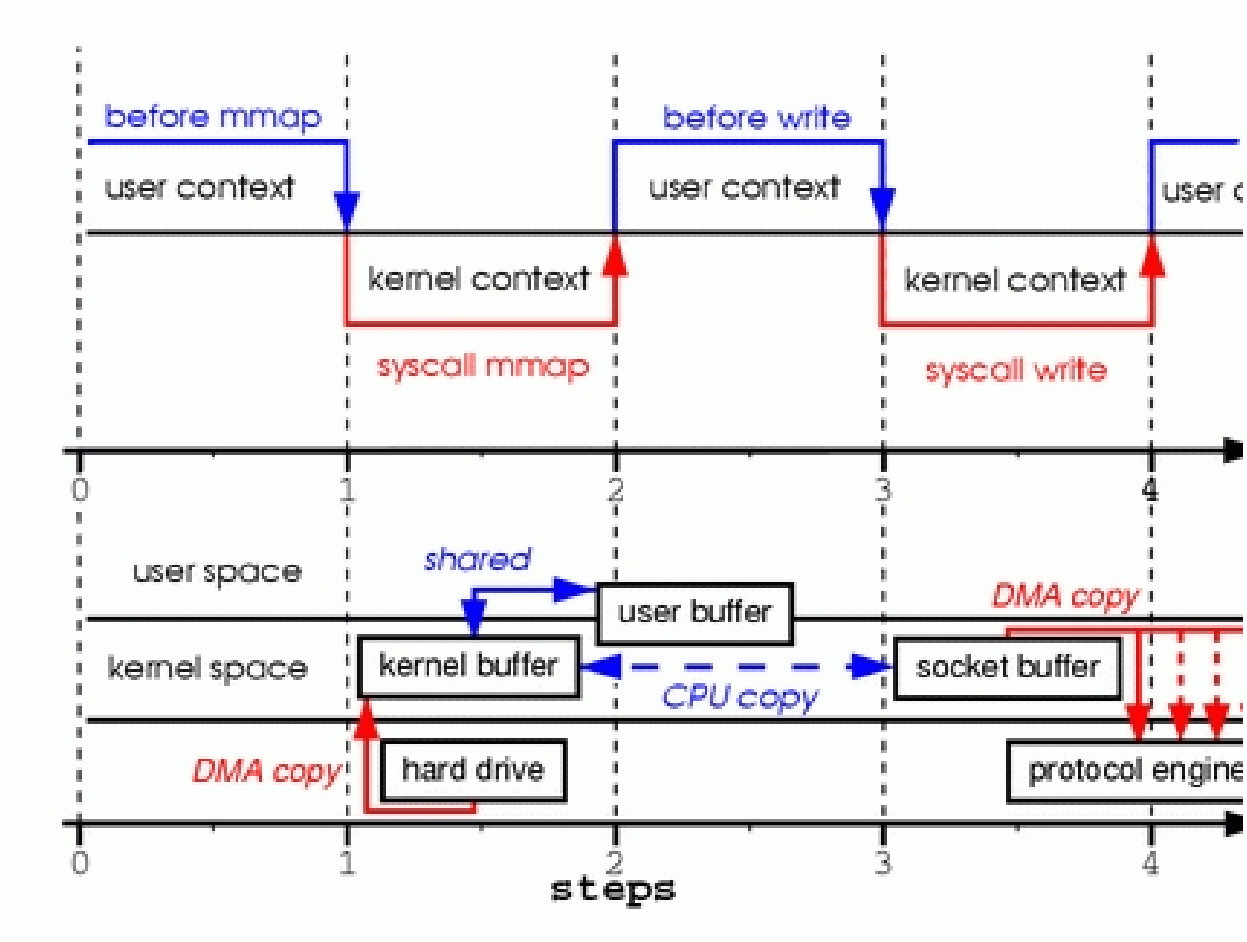

图2. Calling mmap

步骤一：mmap系统调用导致文件的内容通过DMA模块被复制到内核缓冲区中，该缓冲区之后与用户进程共享，这样就内核缓冲区与用户缓冲区之间的复制就不会发生。

步骤二：write系统调用导致内核将数据从内核缓冲区复制到与socket相关联的内核缓冲区中。

步骤三：DMA模块将数据由socket的缓冲区传递给协议引擎时，第3次复制发生。

通过调用mmap而不是read，我们已经将内核需要执行的复制操作减半。当有大量数据要进行传输是，这将有相当良好的效果。然而，性能的改进需要付出代价的;是用mmap与write这种组合方法，存在着一些隐藏的陷阱。例如，考虑一下在内存中对文件进行映射后调用write，与此同时另外一个进程将同一文件截断的情形。此时write系统调用会被进程接收到的SIGBUS信号中断，因为当前进程访问了非法内存地址。对SIGBUS信号的默认处理是杀死当前进程并生成dump core文件——而这对于网络服务器程序而言不是最期望的操作。

有两种方式可用于解决该问题：

第一种方式是为SIGBUS信号设置信号处理程序，并在处理程序中简单的执行return语句。在这样处理方式下，write系统调用返回被信号中断前已写的字节数，并将errno全局变量设置为成功。必须指出，这并不是个好的解决方式——治标不治本。由于收到SIGBUS信号意味着进程发生了严重错误，我不鼓励采取这种解决方式。

第二种方式应用了文件租借（在Microsoft Windows系统中被称为“机会锁”）。这才是解劝前面问题的正确方式。通过对文件描述符执行租借，可以同内核就某个特定文件达成租约。从内核可以获得读/写租约。当另外一个进程试图将你正在传输的文件截断时，内核会向你的进程发送实时信号——RT_SIGNAL_LEASE。该信号通知你的进程，内核即将终止在该文件上你曾获得的租约。这样，在write调用访问非法内存地址、并被随后接收到的SIGBUS信号杀死之前，write系统调用就被RT_SIGNAL_LEASE信号中断了。write的返回值

是在被中断前已写的字节数，全局变量errno设置为成功。下面是一段展示如何从内核获得租约的示例代码。

```
if(fcntl(fd, F_SETSIG, RT_SIGNAL_LEASE) == -1) {

perror("kernel lease set signal");

return -1;

}

/* l_type can be F_RDLCK F_WRLCK */

if(fcntl(fd, F_SETLEASE, l_type)){

perror("kernel lease set type");

return -1;

}
```

Sendfile

sendfile系统调用在内核版本2.1中被引入，目的是简化通过网络在两个本地文件之间进行的数据传输过程。sendfile系统调用的引入，不仅减少了数据复制，还减少了上下文切换的次数。使用方法如下：

```
sendfile(socket, file, len);
```

为了更好的理解所涉及的操作，请看图3

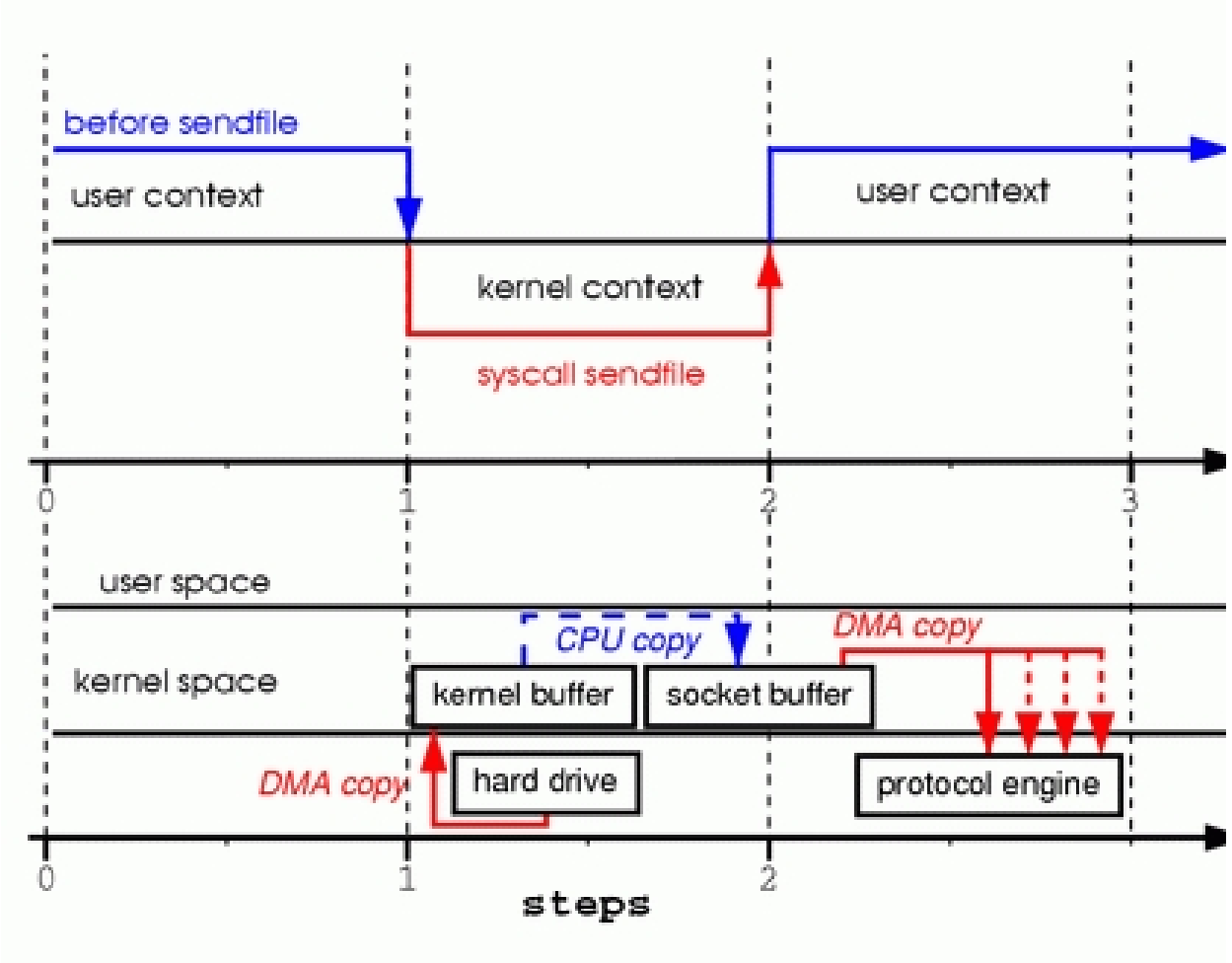

图3. Replacing Read and Write with Sendfile

步骤一：sendfile系统调用导致文件内容通过DMA模块被复制到某个内核缓冲区，之后再被复制到与socket相关联的缓冲区内。

步骤二：当DMA模块将位于socket相关联缓冲区中的数据传递给协议引擎时，执行第3次复制。

你可能会在想，我们在调用sendfile发送数据的期间，如果另外一个进程将文件截断的话，会发生什么事情？如果进程没有为SIGBUS注册任何信号处理函数的话，sendfile系统调用返回被信号中断前已发送的字节数，并将全局变量errno置为成功。

然而，如果在调用sendfile之前，从内核获得了文件租约，那么类似的，在sendfile调用返回前会收到RT_SIGNAL_LEASE。

到此为止，我们已经能够避免内核进行多次复制，然而我们还存在一分多余的副本。这份副本也可以消除吗？当然，在硬件提供的一些帮助下是可以的。为了消除内核产生的素有数据冗余，需要网络适配器支持聚合操作特性。该特性意味着待发送的数据不要求存放在地址连续的内存空间中；相反，可以是分散在各个内存位置。在内核版本2.4中，socket缓冲区描述符结构发生了改动，以适应聚合操作的要求——这就是Linux中所谓的"零拷贝 "。这种方式不仅减少了多个上下文切换，而且消除了数据冗余。从用户层应用程序的角度来开，没有发生任何改动，所有代码仍然是类似下面的形式：

sendfile(socket, file, len);

为了更好的理解所涉及的操作，请看图4

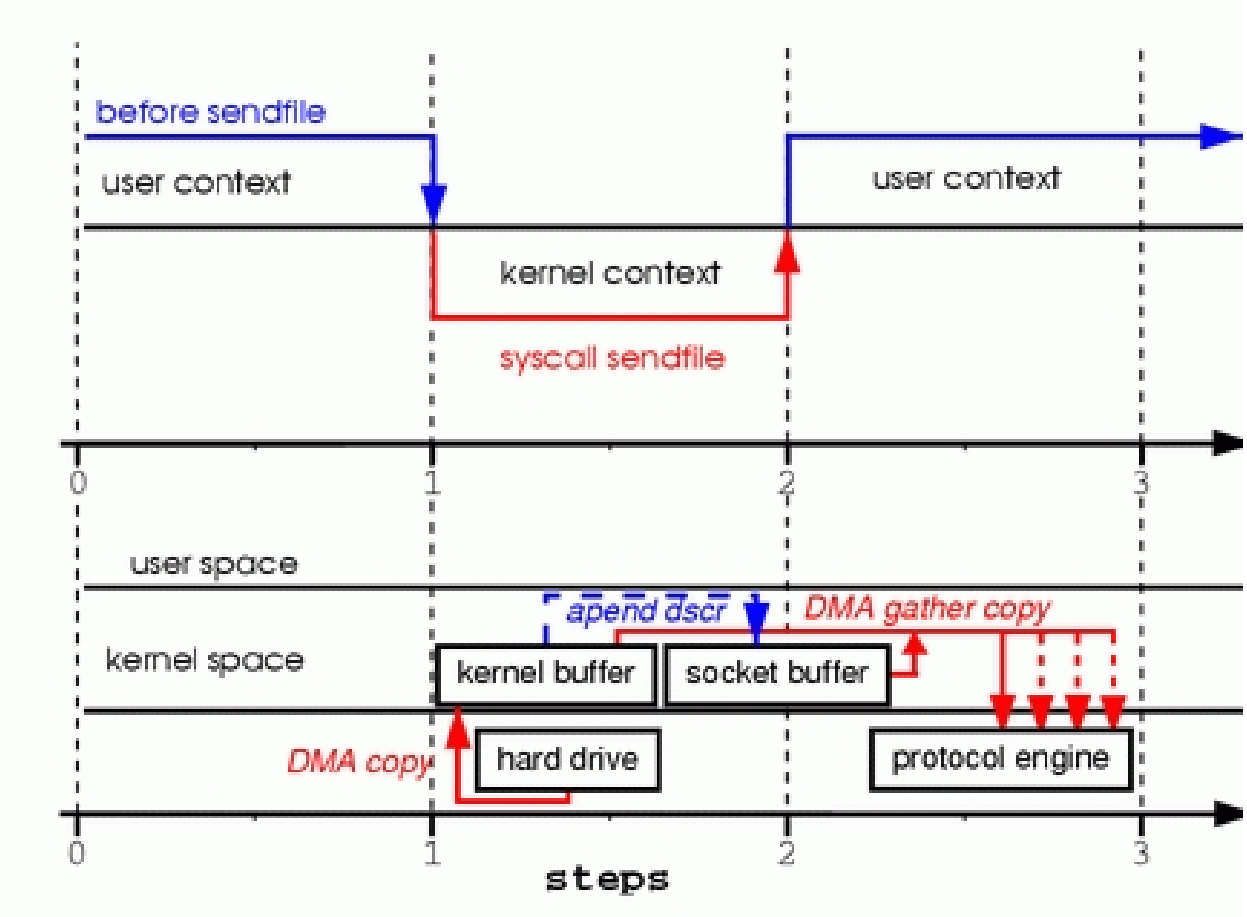

Figure 4. Hardware that supports gather can assemble data from multiple memory locations, eliminating another copy.

步骤一：sendfile系统调用导致文件内容通过DMA模块被复制到内核缓冲区中。

步骤二：数据并未被复制到socket关联的缓冲区内。取而代之的是，只有记录数据位置和长度的描述符被加入到socket缓冲区中。DMA模块将数据直接从内核缓冲区传递给协议引擎，从而消除了遗留的最后一次复制。

由于数据实际上仍然由磁盘复制到内存，再由内存复制到发送设备，有人可能会声称这并不是真正的"零拷贝"。然而，从操作系统的角度来看，这就是"零拷贝"，因为内核空间内不存在冗余数据。应用"零拷贝"特性，出了避免复制之外，还能获得其他性能优势，例如更少的上下文切换，更少的CPU cache污染以及没有CPU必要计算校验和。

现在我们明白了什么是"零拷贝"，让我们将理论付诸实践，编写一些代码。你可以从www.xalien.org/articles/source/sfl-src.tgz处下载完整的源码。执行"tar -zxvf sfl-src.tgz"将源码解压。运行make命令，编译源码，并创建随机数据文件data.bin

从头文件开始介绍代码：

```
/* sfl.c sendfile example program
Dragan Stancevic <
header name function / variable
----------------------------------------------*/
#include /* printf, perror */
#include /* open */
#include /* close */
#include /* errno */
#include /* memset */
```

```
#include /* socket */
#include /* sockaddr_in */
#include /* sendfile */
#include /* inet_addr */
#define BUFF_SIZE (10*1024) /* size of the tmp buffer */
```

除了基本socket操作所需要的 和头文件外，我们还需要包含sendfile系统调用的原型定义，这可以在头文件中找到。

服务器标志：

```
/* are we sending or receiving */
if(argv[1][0] == 's') is_server++;
/* open descriptors */
sd = socket(PF_INET, SOCK_STREAM, 0);
if(is_server) fd = open("data.bin", O_RDONLY);
```

该程序既能以服务端/发送方，也能以客户端/接收方的身份运行。我们需要检查命令行参数中的一项，然后相应的设置is_server标志。程序中大开了一个地址族为PF_INET的流套接字；作为服务端运行时需要向客户发送数据，因此要打开某个数据文件。由于程序中是用sendfile系统调用来发送数据，因此不需要读取文件内容并存储在程序的缓冲区内。

接下来是服务器地址:

```
/* clear the memory */
memset(&sa, 0, sizeof(struct sockaddr_in));
/* initialize structure */
sa.sin_family = PF_INET;
sa.sin_port = htons(1033);
```

```
sa.sin_addr.s_addr = inet_addr(argv[2]);
```

将服务端地址结构清零后设置协议族、端口和IP地址。服务端的IP地址作为命令行参数传递给程序。端口号硬编码为1033，选择该端口是因为它在要求root权限的端口范围之上。

下面是服务端的分支代码：

```
if(is_server){
int client; /* new client socket */
printf("Server binding to [%s]\n", argv[2]);
if(bind(sd, (struct sockaddr *)&sa,sizeof(sa)) < 0){
perror("bind");
exit(errno);
}
```

作为服务端，需要为socket描述符分配一个地址，这是通过系统调用bind完成的，它将服务器地址(sa)分配给socket描述符(sd).

```
if(listen(sd,1) < 0){
perror("listen");
exit(errno);
}
```

由于使用流套接字，必须对内核声明接受外来连接请求的意愿，并设置连接队列的尺寸。此处将队列长度设为1，但是通常会将该值设的高一些，用于接受已建立的连接。在老版本的内核中，该队列被用于阻止SYN flood攻击。由于listen系统调用之改为设定已建立连接的数量，该特性已被listen调用遗弃。内核参数tcp_max_syn_backlog承担了保护系统不受SYN flood攻击的功能。

```
if((client = accept(sd, NULL, NULL)) < 0){
perror("accept");
exit(errno);
}
```

accept系统调用从待处理的已连接队列中选取第一个连接请求，为之建立一个新的socket。accept调用的返回值是新建立连接的描述符；新的socket可以用于read、write和poll/select系统调用。

```
if((cnt = sendfile(client,fd,&off, BUFF_SIZE)) < 0){
perror("sendfile");
exit(errno);
}

printf("Server sent %d bytes.\n", cnt);
close(client);
```

在客户socket描述符上已经建立好连接，因此可以开始将数据传输至远端系统——这时通过调用sendfile系统调用来完成。该调用在Linux中的原型为如下形式：

```
extern ssize_t
sendfile (int __out_fd, int __in_fd, off_t *offset, size_t __count) __THROW;
```

前两个参数为文件描述符，第三个参数表示sendfile开始传输数据的偏移量。第四个参数是打算传输的字节数。为了sendfile可以使用"零拷贝“特性，网卡需要支持聚合操作，此外还应具备校验和计算能力。如果你的NIC不具备这些特性，仍可以是用sendfile来发送数据，区别是内核在传输前会将所有缓冲区的内容合并。

移植性问题

sendfile系统调用的问题之一，总体上来看，是缺少标准化的实现，这与open系统调用类些。sendfile在Linux、Solaris或HP-UX中的实现有很大的不同。这给希望在网络传输代码中利用"零拷贝"的开发者带来了问题。

这些实现差异中的一点在于Linux提供的sendfile，是定义为用于两个文件描述符之间和文件到socket之间的传输接口。另一方面，HP-UX和Solaris中，sendfile只能用于文件到socket的传输。

第二点差异，是Linux没有实现向量化传输。Solaris和HP-UX 中的sendfile系统调用包含额外的参数，用于消除为待传输数据添加头部的开销。

展望

Linux中“零拷贝”的实现还远未结束，并很可能在不久的未来发生变化。更多的功能将会被添加，例如，现在的sendfile不支持向量化传输，而诸如Samba和Apache这样的服务器不得不是用TCP_COKR标志来执行多个sendfile调用。该标志告知系统还有数据要在下一个sendfile调用中到达。TCP_CORK和TCP_NODELAY不兼容，后者在我们希望为数据添加头部时使用。这也正是一个完美的例子，用于说明支持向量化的sendfile将在那些情况下，消除目前实现所强制产生的多个sendfile调用和延迟。

当前sendfile一个相当令人不愉快的限制是它无法用户传输大于2GB的文件。如此尺寸大小的文件，在今天并非十分罕见，不得不复制数据是十分令人失望的。由于这种情况下sendfile和mmap都是不可用的，在未来内核版本中提供sendfile64，将会提供很大的帮助。

结论

尽管有一些缺点，"零拷贝"sendfile是一个很有用的特性。我希望读者认为本文提供了足够的信息以开始在程序中使用sendfile。如果你对这个主题有更深层次

的兴趣，敬请期待我的第二篇文章——"Zero Copy II: Kernel Perspective"，在其中将更深一步的讲述"零拷贝"的内核内部实现。