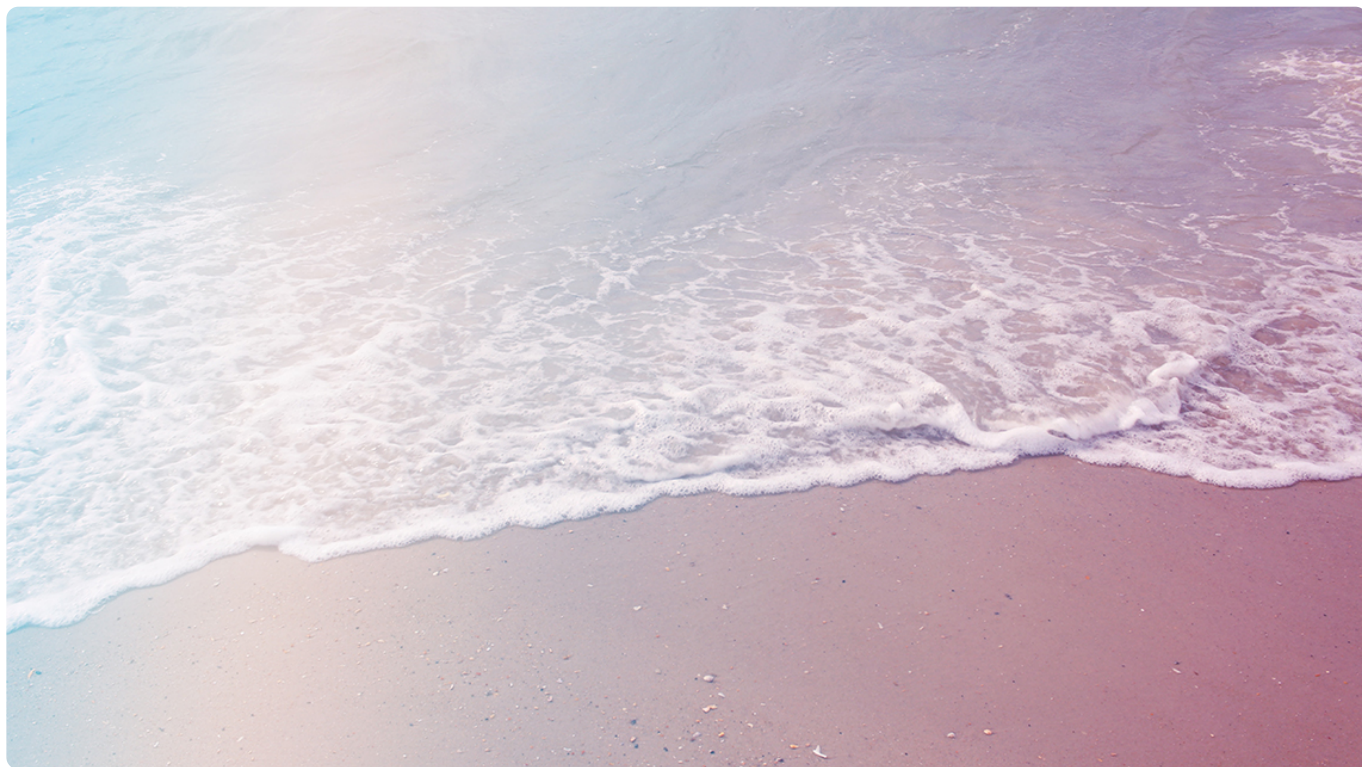


17 | TCP并不总是“可靠”的？

2019-09-09 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 12:12 大小 11.18M



你好，我是盛延敏，这里是网络编程实战第 17 讲，欢迎回来。

在前面一讲中，我们讲到如何理解 TCP 数据流的本质，进而引出了报文格式和解析。在这一讲里，我们讨论通过如何增强读写操作，以处理各种“不可靠”的场景。

TCP 是可靠的？

你可能会认为，TCP 是一种可靠的协议，这种可靠体现在端到端的通信上。这似乎给我们带来了一种错觉，从发送端来看，应用程序通过调用 `send` 函数发送的数据流总能可靠地到达接收端；而从接收端来看，总是可以把对端发送的数据流完整无损地传递给应用程序来处理。

事实上，如果我们对 TCP 传输环节进行详细的分析，你就会沮丧地发现，上述论断是不正确的。

前面我们已经了解，发送端通过调用 send 函数之后，数据流并没有马上通过网络传输出去，而是存储在套接字的发送缓冲区中，由网络协议栈决定何时发送、如何发送。当对应的数据发送给接收端，接收端回应 ACK，存储在发送缓冲区的这部分数据就可以删除了，但是，发送端并无法获取对应数据流的 ACK 情况，也就是说，发送端没有办法判断对端的接收方是否已经接收发送的数据流，如果需要知道这部分信息，就必须在应用层自己添加处理逻辑，例如显式的报文确认机制。

从接收端来说，也没有办法保证 ACK 过的数据部分可以被应用程序处理，因为数据需要接收端程序从接收缓冲区中拷贝，可能出现的状况是，已经 ACK 的数据保存在接收端缓冲区中，接收端处理程序突然崩溃了，这部分数据就没有办法被应用程序继续处理。

你有没有发现，TCP 协议实现并没有提供给上层应用程序过多的异常处理细节，或者说，TCP 协议反映链路异常的能力偏弱，这其实是有原因的。要知道，TCP 诞生之初，就是为了美国国防部服务的，考虑到军事作战的实际需要，TCP 不希望暴露更多的异常细节，而是能够以无人值守、自我恢复的方式运作。

TCP 连接建立之后，能感知 TCP 链路的方式是有限的，一种是以 read 为核心的读操作，另一种是以 write 为核心的写操作。接下来，我们就看下如何通过读写操作来感知异常情况，以及对应的处理方式。

故障模式总结

在实际情景中，我们会碰到各种异常的情况。在这里我把这几种异常情况归结为两大类：



第一类，是对端无 FIN 包发送出来的情况；第二类是对端有 FIN 包发送出来。而这两大类情况又可以根据应用程序的场景细分，接下来我们详细讨论。

网络中断造成的对端无 FIN 包

很多原因都会造成网络中断，在这种情况下，TCP 程序并不能及时感知到异常信息。除非网络中的其他设备，如路由器发出一条 ICMP 报文，说明目的网络或主机不可达，这个时候通过 read 或 write 调用就会返回 Unreachable 的错误。

可惜大多数时候并不是如此，在没有 ICMP 报文的情况下，TCP 程序并不能理解感应到连接异常。如果程序是阻塞在 read 调用上，那么很不幸，程序无法从异常中恢复。这显然是非常不合理的，不过，我们可以通过给 read 操作设置超时来解决，在接下来的第 18 讲中，我会讲到具体的方法。

如果程序先调用了 write 操作发送了一段数据流，接下来阻塞在 read 调用上，结果会非常不同。Linux 系统的 TCP 协议栈会不断尝试将发送缓冲区的数据发送出去，大概在重传 12 次、合计时间约为 9 分钟之后，协议栈会标识该连接异常，这时，阻塞的 read 调用会返回一条 TIMEOUT 的错误信息。如果此时程序还执着地往这条连接写数据，写操作会立即失败，返回一个 SIGPIPE 信号给应用程序。

系统崩溃造成的对端无 FIN 包

当系统突然崩溃，如断电时，网络连接上来不及发出任何东西。这里和通过系统调用杀死应用程序非常不同的是，没有任何 FIN 包被发送出来。

这种情况和网络中断造成的结果非常类似，在没有 ICMP 报文的情况下，TCP 程序只能通过 read 和 write 调用得到网络连接异常的信息，超时错误是一个常见的结果。

不过还有一种情况需要考虑，那就是系统在崩溃之后又重启，当重传的 TCP 分组到达重启后的系统，由于系统中没有该 TCP 分组对应的连接数据，系统会返回一个 RST 重置分节，TCP 程序通过 read 或 write 调用可以分别对 RST 进行错误处理。

如果是阻塞的 read 调用，会立即返回一个错误，错误信息为连接重置 (Connection Reset) 。


如果是一次 write 操作，也会立即失败，应用程序会被返回一个 SIGPIPE 信号。

对端有 FIN 包发出

对端如果有 FIN 包发出，可能的场景是对端调用了 close 或 shutdown 显式地关闭了连接，也可能是对端应用程序崩溃，操作系统内核代为清理所发出的。从应用程序角度上看，无法区分是哪种情形。

阻塞的 read 操作在完成正常接收的数据读取之后，FIN 包会通过返回一个 EOF 来完成通知，此时，read 调用返回值为 0。这里强调一点，收到 FIN 包之后 read 操作不会立即返回。你可以这样理解，收到 FIN 包相当于往接收缓冲区里放置了一个 EOF 符号，之前已经在接收缓冲区的有效数据不会受到影响。


为了展示这些特性，我分别编写了服务器端和客户端程序。

 复制代码

```
1 // 服务端程序
2 int main(int argc, char **argv) {
3     int connfd;
4     char buf[1024];
5
6     connfd = tcp_server(SERV_PORT);
7
8     for (;;) {
9         int n = read(connfd, buf, 1024);
10        if (n < 0) {
11            error(1, errno, "error read");
12        } else if (n == 0) {
13            error(1, 0, "client closed \n");
14        }
15
16        sleep(5);
17
18        int write_nc = send(connfd, buf, n, 0);
19        printf("send bytes: %zu \n", write_nc);
20        if (write_nc < 0) {
21            error(1, errno, "error write");
22        }
23    }
24
25    exit(0);
26 }
```

服务端程序是一个简单的应答程序，在收到数据流之后回显给客户端，在此之前，休眠 5 秒，以便完成后面的实验验证。

客户端程序从标准输入读入，将读入的字符串传输给服务器端：

 复制代码

```
1 // 客户端程序
2 int main(int argc, char **argv) {
3     if (argc != 2) {
4         error(1, 0, "usage: reliable_client01 <IPaddress>");
5     }
6
7     int socket_fd = tcp_client(argv[1], SERV_PORT);
8     char buf[128];
9     int len;
10    int rc;
11
12    while (fgets(buf, sizeof(buf), stdin) != NULL) {
13        len = strlen(buf);
14        rc = send(socket_fd, buf, len, 0);
15        if (rc < 0)
16            error(1, errno, "write failed");
17        rc = read(socket_fd, buf, sizeof(buf));
18        if (rc < 0)
19            error(1, errno, "read failed");
20        else if (rc == 0)
21            error(1, 0, "peer connection closed\n");
22        else
23            fputs(buf, stdout);
24    }
25    exit(0);
26 }
```

read 直接感知 FIN 包

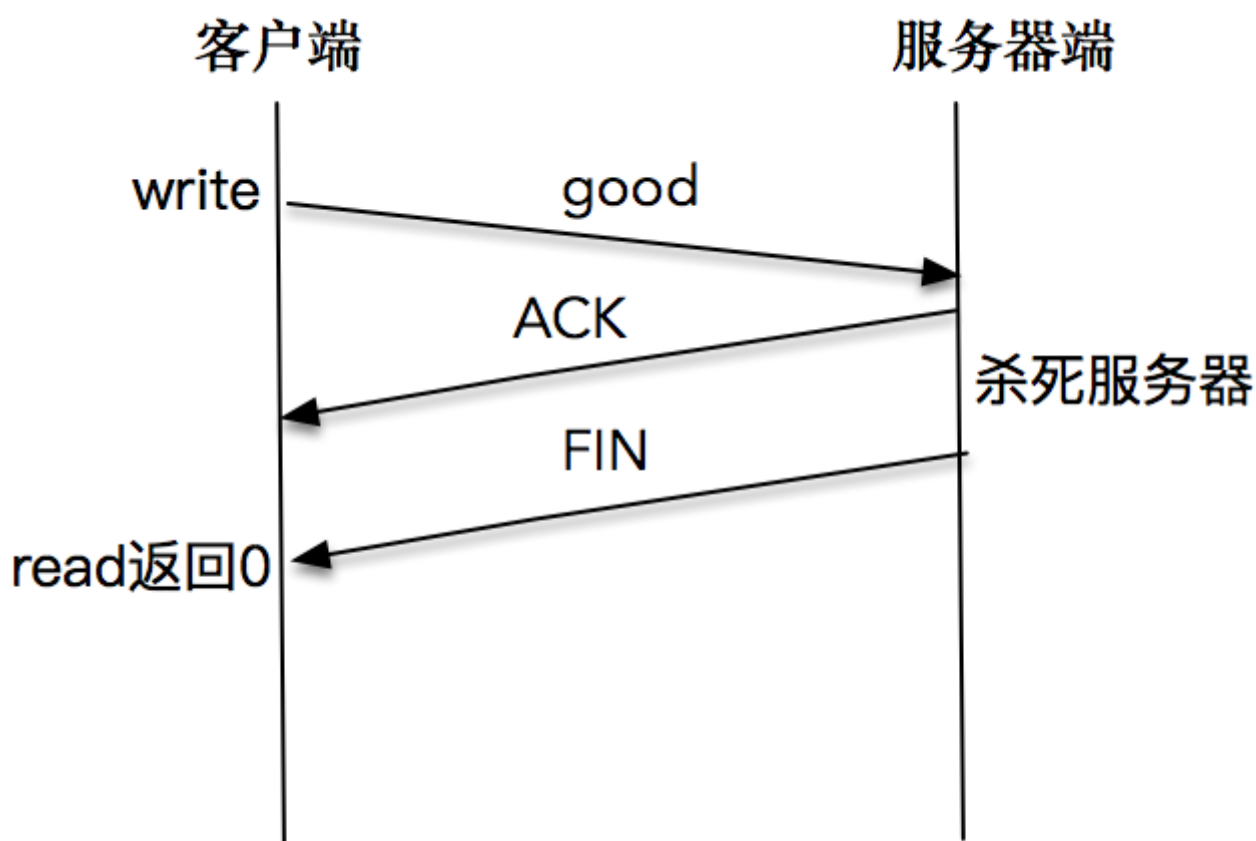
我们依次启动服务器端和客户端程序，在客户端输入 good 字符之后，迅速结束掉服务器端程序，这里需要赶在服务器端从睡眠中苏醒之前杀死服务器程序。

屏幕上打印出：peer connection closed。客户端程序正常退出。

 复制代码

```
1 $ ./reliable_client01 127.0.0.1
2 $ good
```

这说明客户端程序通过 read 调用，感知到了服务端发送的 FIN 包，于是正常退出了客户端程序。




注意如果我们的速度不够快，导致服务器端从睡眠中苏醒，并成功将报文发送出来后，客户端会正常显示，此时我们停留，等待标准输入。如果不继续通过 read 或 write 操作对套接字进行读写，是无法感知服务器端已经关闭套接字这个事实的。

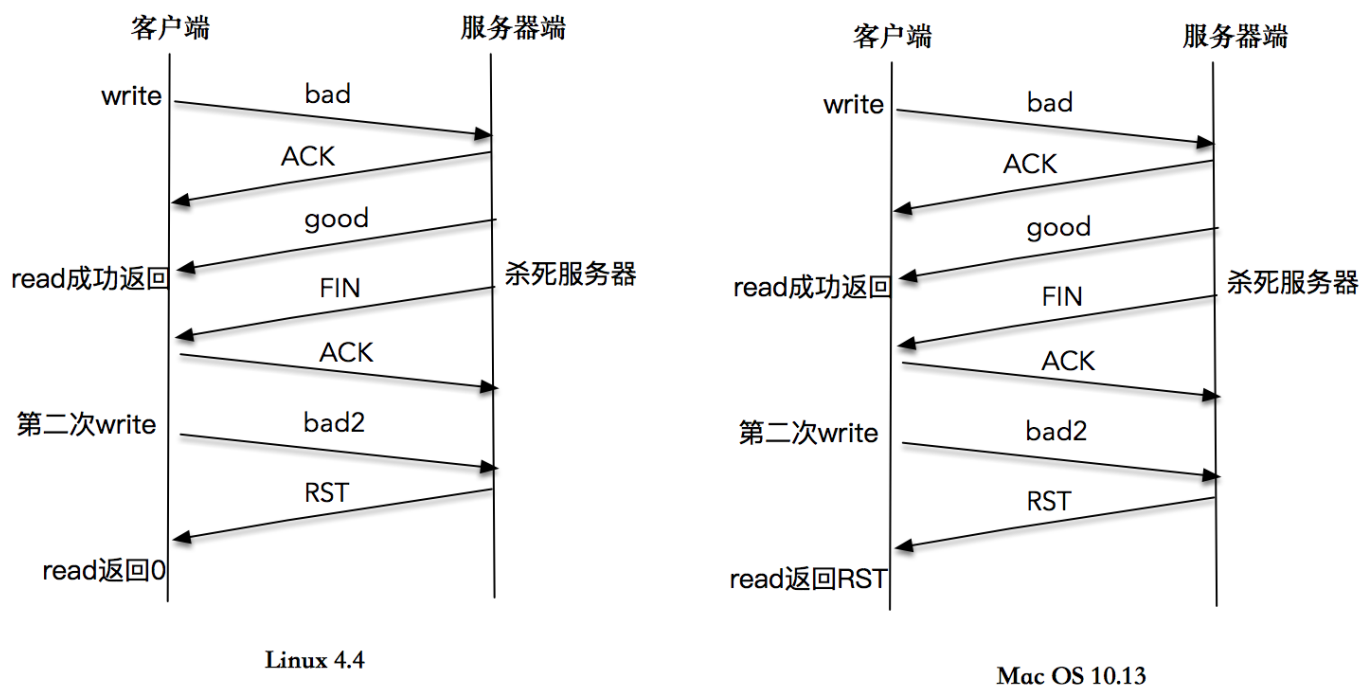
通过 write 产生 RST，read 调用感知 RST

这一次，我们仍然依次启动服务器端和客户端程序，在客户端输入 bad 字符之后，等待一段时间，直到客户端正确显示了服务端的回应 "bad" 字符之后，再杀死服务器程序。客户端再次输入 bad2，这时屏幕上打印出 "peer connection closed"。

我在文稿中给出了这个案例的屏幕输出和时序图。


```
1 $./reliable_client01 127.0.0.1
2 $bad
3 $bad
4 $bad2
5 $peer connection closed
```


 复制代码



在很多书籍和文章中，对这个程序的解读是，收到 FIN 包的客户端继续合法地向服务器端发送数据，服务器端在无法定位该 TCP 连接信息的情况下，发送了 RST 信息，当程序调用 read 操作时，内核会将 RST 错误信息通知给应用程序。这是一个典型的 write 操作造成异常，再通过 read 操作来感知异常的样例。

不过，我在 Linux 4.4 内核上实验这个程序，多次的结果都是，内核正常将 EOF 信息通知给应用程序，而不是 RST 错误信息。


我又在 Max OS 10.13.6 上尝试这个程序，read 操作可以返回 RST 异常信息。输出和时序图也已经给出。

 复制代码

```
1 $./reliable_client01 127.0.0.1
2 $bad
3 $bad
4 $bad2
5 $read failed: Connection reset by peer (54)
```

向一个已关闭连接连续写，最终导致 SIGPIPE


为了模拟这个过程，我对服务器端程序和客户端程序都做了如下修改。

 复制代码

```
1 nt main(int argc, char **argv) {
2     int connfd;
3     char buf[1024];
4     int time = 0;
5
6     connfd = tcp_server(SERV_PORT);
7
8     while (1) {
9         int n = read(connfd, buf, 1024);
10        if (n < 0) {
11            error(1, errno, "error read");
12        } else if (n == 0) {
13            error(1, 0, "client closed \n");
14        }
15
16        time++;
17        fprintf(stdout, "1K read for %d \n", time);
18        usleep(1000);
19    }
20
21    exit(0);
22 }
```

服务器端每次读取 1K 数据后休眠 1 秒，以模拟处理数据的过程。

客户端程序在第 8 行注册了 SIGPIPE 的信号处理程序，在第 14-22 行客户端程序一直循环发送数据流。

 复制代码

```
1 int main(int argc, char **argv) {
2     if (argc != 2) {
3         error(1, 0, "usage: reliable_client02 <IPaddress>");
4     }
5
6     int socket_fd = tcp_client(argv[1], SERV_PORT);
7 }
```



```

8     signal(SIGPIPE, SIG_IGN);
9
10    char *msg = "network programming";
11    ssize_t n_written;
12
13    int count = 10000000;
14    while (count > 0) {
15        n_written = send(socket_fd, msg, strlen(msg), 0);
16        fprintf(stdout, "send into buffer %ld \n", n_written);
17        if (n_written <= 0) {
18            error(1, errno, "send error");
19            return -1;
20        }
21        count--;
22    }
23    return 0;
24 }

```

如果在服务端读取数据并处理过程中，突然杀死服务器进程，我们会看到**客户端很快也会退出**，并在屏幕上打印出 “Connection reset by peer” 的提示。

 复制代码

```

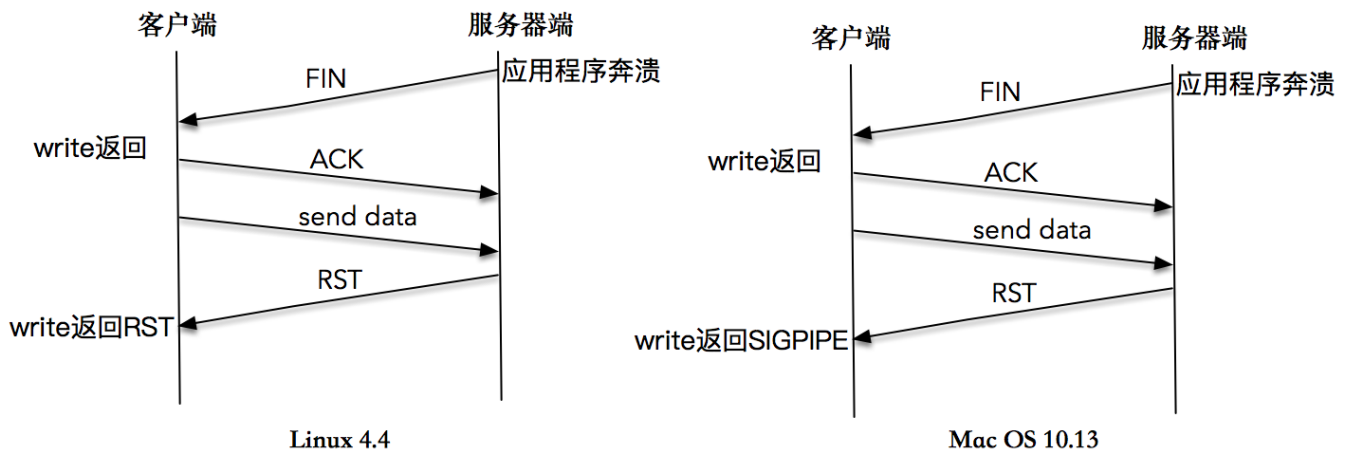
1 $./reliable_client02 127.0.0.1
2 $send into buffer 5917291
3 $send into buffer -1
4 $send: Connection reset by peer

```

这是因为服务端程序被杀死之后，操作系统内核会做一些清理的事情，为这个套接字发送一个 FIN 包，但是，客户端在收到 FIN 包之后，没有 read 操作，还是会继续往这个套接字写入数据。这是因为根据 TCP 协议，连接是双向的，收到对方的 FIN 包只意味着**对方不会再发送任何消息**。在一个双方正常关闭的流程中，收到 FIN 包的一端将剩余数据发送给对面（通过一次或多次 write），然后关闭套接字。

当数据到达服务器端时，操作系统内核发现这是一个指向关闭的套接字，会再次向客户端发送一个 RST 包，对于发送端而言如果此时再执行 write 操作，立即会返回一个 RST 错误信息。

你可以看到针对这个全过程的一张描述图，你可以参考这张图好好理解一下这个过程。



以上是在 Linux 4.4 内核上测试的结果。

在很多书籍和文章中，对这个实验的期望结果不是这样的。大部分的教程是这样说的：在第二次 write 操作时，由于服务器端无法查询到对应的 TCP 连接信息，于是发送了一个 RST 包给客户端，客户端第二次操作时，应用程序会收到一个 SIGPIPE 信号。如果不捕捉这个信号，应用程序会在毫无征兆的情况下直接退出。

我在 Max OS 10.13.6 上尝试这个程序，得到的结果确实如此。你可以看到屏幕显示和时序图。

 复制代码

```
1 #send into buffer 19
2 #send into buffer -1
3 #send error: Broken pipe (32)
```

这说明，Linux4.4 的实现和类 BSD 的实现已经非常不一样了。限于时间的关系，我没有仔细对比其他版本的 Linux，还不清楚是新的内核特性，但有一点是可以肯定的，我们需要记得为 SIGPIPE 注册处理函数，通过 write 操作感知 RST 的错误信息，这样可以保证我们的应用程序在 Linux 4.4 和 Mac OS 上都能正常处理异常。

总结

在这一讲中，我们意识到 TCP 并不是那么“可靠”的。我把故障分为两大类，一类是对端无 FIN 包，需要通过巡检或超时来发现；另一类是对端有 FIN 包发出，需要通过增强 read 或 write 操作的异常处理，帮助我们发现此类异常。

思考题

和往常一样，给大家布置两道思考题。

第一道，你不妨在你的 Linux 系统中重新模拟一下今天文章里的实验，看看运行结果是否和我的一样。欢迎你把内核版本和结果贴在评论里。

第二道题是，如果服务器主机正常关闭，已连接的程序会发生什么呢？

你不妨思考一下这两道题，欢迎你在评论区写下你的模拟结果和思考，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。




网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 如何理解TCP的“流”？

精选留言 (7)

 写留言



传说中的成大大

2019-09-09

看到后面我好像理解了我上面那个提问,当崩溃重启过后是重新三次握手建立连接,创建新的套接字,只是在网络上传输的包,因为是通过ip地址和端口方式进行的寻址,所以新连接上去的客户端会接收到之前还没接收到的包,然后新连接的客户端没有这些包的tcp分组信息所以就会给服务器端(对端)发送一个RST

展开 ∨



1



yusuf

2019-09-09

uname -a

```
Linux tst 3.10.0-957.21.3.el7.x86_64 #1 SMP Tue Jun 18 16:35:19 UTC 2019 x86_64
x86_64 x86_64 GNU/Linux
```

#

```
# ./reliable_client01 127.0.0.1...
```

展开 ∨



1



传说中的成大大

2019-09-09

```
Linux VM-0-13-ubuntu 4.15.0-54-generic #58-Ubuntu SMP Mon Jun 24 10:55:24 U
TC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

这个是客户端不停的发送然后服务器端突然关闭

```
./client: send error: Connection reset by peer
```

展开 ∨



传说中的成大大

2019-09-09

第一问因为要写代码所以等会来回答先回答第二问

其实服务器正常退出和异常突出我觉得都差不多,都需要靠read或者write去感知,如果对方已经断开连接就会发送一个fin到本方的接收缓冲区变为eof,read函数也返回0,调用write会失败

展开 ∨



传说中的成大大

2019-09-09

其实我也一直想问 比如 我客户端突然崩溃了 然后再启动客户端连接上服务器 到底是新建一个链接 还是老的连接 并且发送rst?

展开 ∨



石将从

2019-09-09

这篇读了几遍还是很懵，很多概念理不清楚，不知道对端到底是服务器端还是客户端，都混淆了

1



刘晓林

2019-09-09

没有看出“通过 write 产生 RST，read 调用感知 RST”和“向一个已关闭连接连续写，产生SIGPIPE”二者有什么区别呀。前者两个write之间多了一次read，为何在linux下的返回结果就不一样了呀？

1

