

13 | 小数据包应对之策：理解TCP协议中的动态数据传输

2019-08-30 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 12:31 大小 11.47M



你好，我是盛延敏，这里是网络编程实战第 13 讲，欢迎回来。

在上一篇文章里，我在应用程序中模拟了 TCP Keep-Alive 机制，完成 TCP 心跳检测，达到发现不活跃连接的目的。在这一讲里，我们将从 TCP 角度看待数据流的发送和接收。

如果你学过计算机网络的话，那么对于发送窗口、接收窗口、拥塞窗口等名词肯定不会陌生，它们各自解决的是什么问题，又是如何解决的？在今天的文章里，我希望能从一个更加通俗易懂的角度进行剖析。

调用数据发送接口以后.....

在前面的内容中，我们已经熟悉如何通过套接字发送数据，比如使用 write 或者 send 方法来进行数据流的发送。

我们已经知道，**调用这些接口并不意味着数据被真正发送到网络上，其实，这些数据只是从应用程序中被拷贝到了系统内核的套接字缓冲区中，或者说是发送缓冲区中**，等待协议栈的处理。至于这些数据是什么时候被发送出去的，对应用程序来说，是无法预知的。对这件事情真正负责的，是运行于操作系统内核的 TCP 协议栈实现模块。

流量控制和生产者 - 消费者模型

我们可以把理想中的 TCP 协议可以想象成一队运输货物的货车，运送的货物就是 TCP 数据包，这些货车将数据包从发送端运送到接收端，就这样不断周而复始。

我们仔细想一下，货物达到接收端之后，是需要卸货处理、登记入库的，接收端限于自己的处理能力和仓库规模，是不可能让这队货车以不可控的速度发货的。接收端肯定会和发送端不断地进行信息同步，比如接收端通知发送端：“后面那 20 车你给我等等，等我这里腾出地方你再继续发货。”

其实这就是发送窗口和接收窗口的本质，我管这个叫做“TCP 的生产者 - 消费者”模型。

发送窗口和接收窗口是 TCP 连接的双方，一个作为生产者，一个作为消费者，为了达到一致协同的生产 - 消费速率、而产生的算法模型实现。

说白了，作为 TCP 发送端，也就是生产者，不能忽略 TCP 的接收端，也就是消费者的实际状况，不管不顾地把数据包都传送过来。如果都传送过来，消费者来不及消费，必然会丢弃；而丢弃反过使得生产者又重传，发送更多的数据包，最后导致网络崩溃。

我想，理解了“TCP 的生产者 - 消费者”模型，再反过来看发送窗口和接收窗口的设计目的和方式，我们就会恍然大悟了。

拥塞控制和数据传输

TCP 的生产者 - 消费者模型，只是在考虑单个连接的数据传递，但是，TCP 数据包是需要经过网卡、交换机、核心路由器等一系列的网络设备的，网络设备本身的能力也是有限的，当多个连接的数据包同时在网络上传送时，势必会发生带宽争抢、数据丢失等，这样，TCP

就必须考虑多个连接共享在有限的带宽上，兼顾效率和公平性的控制，这就是拥塞控制的本质。

举个形象一点的例子，有一个货车行驶在半夜三点的大路上，这样的场景是断然不需要拥塞控制的。

我们可以把网络设备形成的网络信息高速公路和生活中实际的高速公路做个对比。正是因为有多个 TCP 连接，形成了高速公路上的多队运送货车，高速公路上开始变得熙熙攘攘，这个时候，就需要拥塞控制的接入了。

在 TCP 协议中，拥塞控制是通过拥塞窗口来完成的，拥塞窗口的大小会随着网络状况实时调整。

拥塞控制常用的算法有“慢启动”，它通过一定的规则，慢慢地将网络发送数据的速率增加到一个阈值。超过这个阈值之后，慢启动就结束了，另一个叫做“拥塞避免”的算法登场。在这个阶段，TCP 会不断地探测网络状况，并随之不断调整拥塞窗口的大小。

现在你可以发现，在任何一个时刻，TCP 发送缓冲区的数据是否能真正发送出去，**至少**取决于两个因素，一个是**当前的发送窗口大小**，另一个是**拥塞窗口大小**，而 TCP 协议中总是取两者中最小值作为判断依据。比如当前发送的字节为 100，发送窗口的大小是 200，拥塞窗口的大小是 80，那么取 200 和 80 中的最小值，就是 80，当前发送的字节数显然是大于拥塞窗口的，结论就是不能发送出去。

这里千万要分清楚发送窗口和拥塞窗口的区别。

发送窗口反应了作为单 TCP 连接、点对点之间的流量控制模型，它是需要和接收端一起共同协调来调整大小的；而拥塞窗口则是反应了作为多个 TCP 连接共享带宽的拥塞控制模型，它是发送端独立地根据网络状况来动态调整的。

一些有趣的场景

注意我在前面的表述中，提到了在任何一个时刻里，TCP 发送缓冲区的数据是否能真正发送出去，用了“至少两个因素”这个说法，细心的你有没有想过这个问题，除了之前引入的发送窗口、拥塞窗口之外，还有什么其他因素吗？

我们考虑以下几个有趣的场景：

第一个场景，接收端处理得急不可待，比如刚刚读入了 100 个字节，就告诉发送端：“喂，我已经读走 100 个字节了，你继续发”，在这种情况下，你觉得发送端应该怎么做呢？

第二个场景是所谓的“交互式”场景，比如我们使用 telnet 登录到一台服务器上，或者使用 SSH 和远程的服务器交互，这种情况下，我们在屏幕上敲打了一个命令，等待服务器返回结果，这个过程需要不断和服务端进行数据传输。这里最大的问题是，每次传输的数据可能都非常小，比如敲打的命令“pwd”，仅仅三个字符。这意味着什么？这就好比，每次叫了一辆大货车，只送了一个小水壶。在这种情况下，你又觉得发送端该怎么做才合理呢？

第三个场景是从接收端来说的。我们知道，接收端需要对每个接收到的 TCP 分组进行确认，也就是发送 ACK 报文，但是 ACK 报文本身是不带数据的分段，如果一直这样发送大量的 ACK 报文，就会消耗大量的带宽。之所以会这样，是因为 TCP 报文、IP 报文固有的消息头是不可或缺的，比如两端的地址、端口号、时间戳、序列号等信息，在这种情形下，你觉得合理的做法是什么？

TCP 之所以复杂，就是因为 TCP 需要考虑的因素较多。像以上这几个场景，都是 TCP 需要考虑的情况，一句话概况就是如何有效地利用网络带宽。

第一个场景也被叫做糊涂窗口综合症，这个场景需要在接收端进行优化。也就是说，接收端不能在接收缓冲区空出一个很小的部分之后，就急吼吼地向发送端发送窗口更新通知，而是需要在自己的缓冲区大到一个合理的值之后，再向发送端发送窗口更新通知。这个合理的值，由对应的 RFC 规范定义。

第二个场景需要在发送端进行优化。这个优化的算法叫做 Nagle 算法，Nagle 算法的本质其实就是限制大批量的小数据包同时发送，为此，它提出，在任何一个时刻，未被确认的小数据包不能超过一个。这里的小数据包，指的是长度小于最大报文段长度 MSS 的 TCP 分组。这样，发送端就可以把接下来连续的几个小数据包存储起来，等待接收到前一个小数据包的 ACK 分组之后，再将数据一次性发送出去。

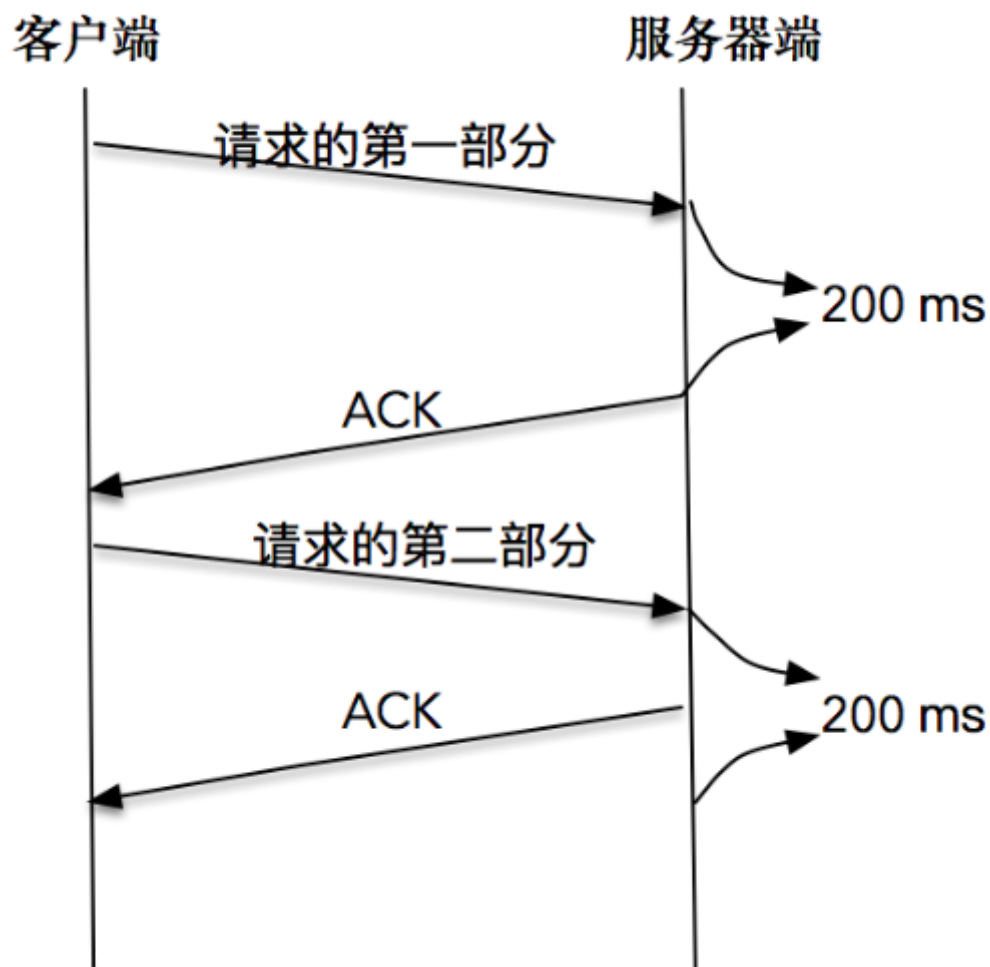
第三个场景，也是需要在接收端进行优化，这个优化的算法叫做延时 ACK。延时 ACK 在收到数据后并不马上回复，而是累计需要发送的 ACK 报文，等到有数据需要发送给对端时，将累计的 ACK **捎带一并发送出去**。当然，延时 ACK 机制，不能无限地延时下去，否则发送端误认为数据包没有发送成功，引起重传，反而会占用额外的网络带宽。

禁用 Nagle 算法

有没有发现一个很奇怪的组合，即 Nagle 算法和延时 ACK 的组合。

这个组合为什么奇怪呢？我举一个例子你来体会一下。


比如，客户端分两次将一个请求发送出去，由于请求的第一部分的报文未被确认，Nagle 算法开始起作用；同时延时 ACK 在服务器端起作用，假设延时时间为 200ms，服务器等待 200ms 后，对请求的第一部分进行确认；接下来客户端收到了确认后，Nagle 算法解除请求第二部分的阻止，让第二部分得以发送出去，服务器端在收到之后，进行处理应答，同时将第二部分的确认捎带发送出去。



你从这张图中可以看到，Nagle 算法和延时确认组合在一起，增大了处理时延，实际上，两个优化彼此在阻止对方。

从上面的例子可以看到，在有些情况下 Nagle 算法并不适用，比如对时延敏感的应用。

幸运的是，我们可以通过对套接字的修改来关闭 Nagle 算法。


 复制代码

```
1 int on = 1;
2 setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *)&on, sizeof(on));
```

值得注意的是，除非我们对此有十足的把握，否则不要轻易改变默认的 TCP Nagle 算法。因为在现代操作系统中，针对 Nagle 算法和延时 ACK 的优化已经非常成熟了，有可能在禁用 Nagle 算法之后，性能问题反而更加严重。


将写操作合并

其实前面的例子里，如果我们能将一个请求一次性发送过去，而不是分开两部分独立发送，结果会好很多。所以，在写数据之前，将数据合并到缓冲区，批量发送出去，这是一个比较好的做法。不过，有时候数据会存储在两个不同的缓存中，对此，我们可以使用如下的方法来进行数据的读写操作，从而避免 Nagle 算法引发的副作用。

 复制代码

```
1 ssize_t writev(int filesdes, const struct iovec *iov, int iovcnt)
2 ssize_t readv(int filesdes, const struct iovec *iov, int iovcnt);
```

这两个函数的第二个参数都是指向某个 iovec 结构数组的一个指针，其中 iovec 结构定义如下：

 复制代码

```
1 struct iovec {
2 void *iov_base; /* starting address of buffer */
3 size_t iov_len; /* size of buffer */
4 };
```

下面的程序展示了集中写的方式：

 复制代码


```

1 int main(int argc, char **argv) {
2     if (argc != 2) {
3         error(1, 0, "usage: tcpclient <IPAddress>");
4     }
5
6     int socket_fd;
7     socket_fd = socket(AF_INET, SOCK_STREAM, 0);
8
9     struct sockaddr_in server_addr;
10    bzero(&server_addr, sizeof(server_addr));
11    server_addr.sin_family = AF_INET;
12    server_addr.sin_port = htons(SERV_PORT);
13    inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
14
15    socklen_t server_len = sizeof(server_addr);
16    int connect_rt = connect(socket_fd, (struct sockaddr *) &server_addr, server_len);
17    if (connect_rt < 0) {
18        error(1, errno, "connect failed ");
19    }
20
21    char buf[128];
22    struct iovec iov[2];
23
24    char *send_one = "hello,";
25    iov[0].iov_base = send_one;
26    iov[0].iov_len = strlen(send_one);
27    iov[1].iov_base = buf;
28    while (fgets(buf, sizeof(buf), stdin) != NULL) {
29        iov[1].iov_len = strlen(buf);
30        int n = htonl(iov[1].iov_len);
31        if (writev(socket_fd, iov, 2) < 0)
32            error(1, errno, "writev failure");
33    }
34    exit(0);
35 }

```

这个程序的前半部分创建套接字，建立连接就不再赘述了。关键的是 24-33 行，使用了 `iovec` 数组，分别写入了两个不同的字符串，一个是 “hello,”，另一个通过标准输入读入。

在启动该程序之前，我们需要启动服务器端程序，在客户端依次输入 “world” 和 “network”：

 复制代码

1 world

接下来我们可以看到服务器端接收到了 `iovec` 组成的新的字符串。这里的原理其实就是在调用 `writenv` 操作时，会自动把几个数组的输入合并成一个有序的字节流，然后发送给对端。

[复制代码](#)

```
1 received 12 bytes: hello,world
2
3 received 14 bytes: hello,network
```

总结

今天的内容我重点讲述了 TCP 流量控制的生产者 - 消费者模型，你需要记住以下几点：

发送窗口用来控制发送和接收端的流量；阻塞窗口用来控制多条连接公平使用的有限带宽。

小数据包加剧了网络带宽的浪费，为了解决这个问题，引入了如 Nagle 算法、延时 ACK 等机制。

在程序设计层面，不要多次频繁地发送小报文，如果有，可以使用 `writenv` 批量发送。

思考题

和往常一样，给大家留两道思考题：

针对最后呈现的 `writenv` 函数，你可以查一查 Linux 下一次性最多允许数组的大小是多少？

另外 TCP 拥塞控制算法是一个非常重要的研究领域，请你查阅下最新的有关这方面的研究，看看有没有新的发现？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起来交流。

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 12 | 连接无效：使用Keep-Alive还是应用心跳来检测？

下一篇 14 | UDP也可以是“已连接”？

精选留言 (10)

写留言



d

2019-08-30

就拥塞控制算法这块，记得前一阵阿里发布一个HPCC算法，盛老师点评一下？谢谢



1



yusuf

2019-09-02

1)、iov_len的总和 ≤ 32767 , iovcnt ≤ 16

展开



yusuf

2019-09-01

场景二中，发送端把接下来连续的几个小数据包存储起来，等待接收到前一个小数据包的ACK 分组之后，再将数据一次性发送出去。

是将多个数据包合并成了1个包吗？

展开 ▾



谢童

2019-08-31

老师，writev 是原子操作吗？在多线程的情况下可以不加锁使用吗？



不动声色满心澎湃

2019-08-31

writev是减少write的使用次数吧。一次writev中的数据也有可能分多个包发出去。我说的对吗老师



锦

2019-08-30

请教老师一个问题，慢启动是因为需要探测网络带宽质量，快恢复，快重连是什么意思呢？



锦

2019-08-30

问题一：Linux中最多允许1024个元素

请教一个问题，tcp中有各种窗口，很头晕，比如，发送窗口，接收窗口，通告窗口（Advertised window），滑动窗口，拥塞窗口。为什么要弄这么多窗口呢？都是为了做流量控制吗？如何去理解呢？

展开 ▾



LDxy

2019-08-30

下载软件通常使用多线程建立多个TCP连接来下载一个大文件，是不是也是为了尽量避免TCP拥塞控制带来的影响，从而充分利用带宽？因为从实际使用来看，下载软件一旦跑满带宽，其他软件基本是抢不过它的





许童童

2019-08-30

TCP 拥塞控制算法,我知道最新的有BBR算法, 这个算法在网络包填满路由器缓冲区之前就触发流量控制, 而不在丢包后才触发, 有效的降低了延迟。

展开 ∨



张立华

2019-08-30

非阻塞socket, 对于write和send, 返回实际发送的字节数。所以一般在while里不断发送, 直到全部发送完毕。 send根据只要根据要发送的buf做个偏移, 很方便。 而writev 就很繁琐了啊

