

# 目录

|                                      |    |                                |    |
|--------------------------------------|----|--------------------------------|----|
| 第一章 概述                               | 5  | 12.5 参考文献 . . . . .            | 37 |
| 第二章 Internet 地址结构                    | 7  | 第十三章 TCP: 连接管理                 | 39 |
| 第三章 链路层                              | 9  | 13.1 引言 . . . . .              | 39 |
| 第四章 地址解析协议                           | 11 | 13.2 TCP 连接的建立与终止 . . . . .    | 39 |
| 第五章 Internet 协议                      | 13 | 13.2.1 TCP 半关闭 . . . . .       | 42 |
| 第六章 系统配置: DHCP 和自动配置                 | 15 | 13.2.2 同时打开与关闭 . . . . .       | 42 |
| 第七章 防火墙和网络地址转换                       | 17 | 13.2.3 初始序列号 . . . . .         | 43 |
| 第八章 ICMPv4 和 ICMPv6: Internet 控制报文协议 | 19 | 13.2.4 例子 . . . . .            | 44 |
| 第九章 广播和本地组播 (IGMP 和 MLD)             | 21 | 13.2.5 连接建立超时 . . . . .        | 45 |
| 第十章 用户数据报协议和 IP 分片                   | 23 | 13.2.6 连接与转换器 . . . . .        | 46 |
| 第十一章 名称解析和域名系统                       | 25 | 13.3 TCP 选项 . . . . .          | 47 |
| 第十二章 TCP: 传输控制协议                     | 27 | 13.3.1 最大段大小选项 . . . . .       | 47 |
| 12.1 引言 . . . . .                    | 27 | 13.3.2 选择确认选项 . . . . .        | 48 |
| 12.1.1 ARQ 和重传 . . . . .             | 27 | 13.3.3 窗口缩放选项 . . . . .        | 48 |
| 12.1.2 分组窗口和滑动窗口 . . . . .           | 29 | 13.3.4 时间戳选项与防回绕序列号 . . . . .  | 49 |
| 12.1.3 变量窗口: 流量控制和拥塞控制 . . . . .     | 30 | 13.3.5 用户超时选项 . . . . .        | 50 |
| 12.1.4 设置重传超时 . . . . .              | 31 | 13.3.6 认证选项 . . . . .          | 51 |
| 12.2 TCP 的引入 . . . . .               | 31 | 13.4 TCP 的路径最大传输单元发现 . . . . . | 51 |
| 12.2.1 TCP 服务类型 . . . . .            | 32 | 13.4.1 例子 . . . . .            | 53 |
| 12.2.2 TCP 中的可靠性 . . . . .           | 32 | 13.5 TCP 状态转换 . . . . .        | 54 |
| 12.3 TCP 头部和封装 . . . . .             | 34 | 13.5.1 TCP 状态转换图 . . . . .     | 54 |
| 12.4 总结 . . . . .                    | 36 | 13.5.2 TIME_WAIT 状态 . . . . .  | 56 |
|                                      |    | 13.5.3 静默时间的概念 . . . . .       | 59 |
|                                      |    | 13.5.4 FIN_WAIT_2 状态 . . . . . | 59 |
|                                      |    | 13.5.5 同时打开与关闭的转换 . . . . .    | 59 |
|                                      |    | 13.6 重置报文段 . . . . .           | 60 |
|                                      |    | 13.6.1 针对不存在端口的连接请求 . . . . .  | 60 |
|                                      |    | 13.6.2 终止一条连接 . . . . .        | 61 |
|                                      |    | 13.6.3 半开连接 . . . . .          | 61 |

|                                      |           |  |            |
|--------------------------------------|-----------|--|------------|
| 13.6.4 时间等待错误 . . . . .              | 62        | 14.7.3 前移 RTO 恢复 (F-RTO) . . . . .       | 94         |
| 13.7 TCP 服务器选项 . . . . .             | 63        | 14.7.4 Eifel 响应算法 . . . . .              | 94         |
| 13.7.1 TCP 端口号 . . . . .             | 63        | 14.8 包失序与包重复 . . . . .                   | 96         |
| 13.7.2 限制本地 IP 地址 . . . . .          | 64        | 14.8.1 . . . . .                         | 96         |
| 13.7.3 限制外部节点 . . . . .              | 65        | 14.8.2 重复 . . . . .                      | 97         |
| 13.7.4 进入连接队列 . . . . .              | 65        | 14.9 目的度量 . . . . .                      | 97         |
| 13.8 与 TCP 连接管理相关的攻击 . . . . .       | 68        | 14.10 重新组包 . . . . .                     | 98         |
| 13.8.1 syncookie . . . . .           | 68        | 14.11 与 TCP 重传相关的攻击 . . . . .            | 98         |
| 13.8.2 路径最大传输单元 . . . . .            | 69        | 14.12 总结 . . . . .                       | 99         |
| 13.8.3 hijacking tcp . . . . .       | 69        |  |            |
| 13.8.4 欺骗攻击 . . . . .                | 69        | <b>第十五章 TCP 数据流和与窗口管理</b>                | <b>101</b> |
| 13.9 总结 . . . . .                    | 70        | 15.1 引言 . . . . .                        | 101        |
| 13.10 参考文献 . . . . .                 | 70        | 15.2 交互式通信 . . . . .                     | 101        |
| <b>第十四章 TCP 超时与重传</b>                | <b>71</b> | 15.3 延时确认 . . . . .                      | 104        |
| 14.1 引言 . . . . .                    | 71        | 15.4 Nagle 算法 . . . . .                  | 104        |
| 14.2 简单的超时与重传举例 . . . . .            | 72        | 15.4.1 延时 ACK 与 Nagle 算法<br>结合 . . . . . | 106        |
| 14.3 设置重传超时 . . . . .                | 74        | 15.4.2 禁用 Nagle 算法 . . . . .             | 106        |
| 14.3.1 经典方法 . . . . .                | 74        | 15.5 流量控制与窗口管理 . . . . .                 | 107        |
| 14.3.2 标准方法 . . . . .                | 75        | 15.5.1 滑动窗口 . . . . .                    | 107        |
| 14.3.3 Linux 采用的方法 . . . . .         | 78        | 15.5.2 零窗口与 TCP 持续计时器 . . . . .          | 109        |
| 14.3.4 RTT 估计器行为 . . . . .           | 82        | 15.5.3 糊涂窗口综合征 . . . . .                 | 112        |
| 14.3.5 RTTM 对丢包和失序的<br>鲁棒性 . . . . . | 82        | 15.5.4 大容量缓存与自动调优 . . . . .              | 116        |
| 14.4 基于计时器的重传 . . . . .              | 83        | 15.6 紧急机制 . . . . .                      | 118        |
| 14.4.1 例子 . . . . .                  | 84        | 15.6.1 例子 . . . . .                      | 119        |
| 14.5 快速重传 . . . . .                  | 84        | 15.7 与窗口管理相关的攻击 . . . . .                | 121        |
| 14.5.1 例子 . . . . .                  | 85        | 15.8 总结 . . . . .                        | 122        |
| 14.6 带选择确认的重传 . . . . .              | 88        | <b>第十六章 TCP 拥塞控制</b>                     | <b>123</b> |
| 14.6.1 SACK 接收端行为 . . . . .          | 89        | 16.1 引言 . . . . .                        | 123        |
| 14.6.2 SACK 发送端行为 . . . . .          | 89        | 16.1.1 TCP 拥塞检测 . . . . .                | 124        |
| 14.6.3 例子 . . . . .                  | 90        | 16.1.2 减缓 TCP 发送 . . . . .               | 124        |
| 14.7 伪超时与重传 . . . . .                | 92        | 16.2 一些经典算法 . . . . .                    | 125        |
| 14.7.1 重复 SACK (DSACK) 扩展 . . . . .  | 93        | 16.2.1 慢启动 . . . . .                     | 127        |
| 14.7.2 Eifel 检测算法 . . . . .          | 93        | 16.2.2 慢启动和拥塞避免的选择 . . . . .             | 129        |

|                                       |     |  |            |
|---------------------------------------|-----|--|------------|
| 16.2.3 Tahoe、Reno 以及快速恢复算法 . . . . .  | 131 | 16.6 共享拥塞状态信息 . . . . .                        | 154        |
| 16.2.4 标准 TCP . . . . .               | 131 | 16.7 TCP 友好性 . . . . .                         | 156        |
| 16.3 对标准算法的改进 . . . . .               | 132 | 16.8 高速环境下的 TCP . . . . .                      | 157        |
| 16.3.1 NewReno . . . . .              | 133 | 16.8.1 高速 TCP 与受限的慢启动 . . . . .                | 158        |
| 16.3.2 采用选择确认机制的 TCP 拥塞控制 . . . . .   | 133 | 16.8.2 二进制增长拥塞控制 (BIC 和 CUBIC) . . . . .       | 160        |
| 16.3.3 转发确认 (FACK) 和速率减半 . . . . .    | 134 | 16.9 基于延迟的拥塞控制算法 . . . . .                     | 163        |
| 16.3.4 限制传输 . . . . .                 | 135 | 16.9.1 Vegas 算法 . . . . .                      | 163        |
| 16.3.5 拥塞窗口校验 . . . . .               | 135 | 16.9.2 FAST 算法 . . . . .                       | 164        |
| 16.4 伪 RTO 处理——Eifel 响应算法 . . . . .   | 136 | 16.9.3 TCP Westwood 算法和 Westwood+ 算法 . . . . . | 165        |
| 16.5 扩展举例 . . . . .                   | 137 | 16.9.4 复合 TCP . . . . .                        | 165        |
| 16.5.1 慢启动行为 . . . . .                | 139 | 16.10 缓冲区膨胀 . . . . .                          | 167        |
| 16.5.2 发送暂停和本地拥塞 (事件 1) . . . . .     | 140 | 16.11 积极队列管理和 ECN . . . . .                    | 168        |
| 16.5.3 延伸 ACK 和本地拥塞恢复 . . . . .       | 143 | 16.12 与 TCP 拥塞控制相关的攻击 . . . . .                | 170        |
| 16.5.4 快速重传和 SACK 恢复 (事件 2) . . . . . | 146 | 16.13 总结 . . . . .                             | 171        |
| 16.5.5 其他本地拥塞和快速重传事件 . . . . .        | 149 | 16.14 参考文献 . . . . .                           | 173        |
| 16.5.6 超时、重传和撤销 cwnd 修改 . . . . .     | 151 | <b>第十七章 TCP 保活机制 . . . . .</b>                 | <b>175</b> |
| 16.5.7 连接结束 . . . . .                 | 154 | 17.1 引言 . . . . .                              | 175        |
|                                       |     | 17.2 描述 . . . . .                              | 176        |
|                                       |     | 17.2.1 保活功能举例 . . . . .                        | 178        |
|                                       |     | 17.3 与 TCP 保活机制相关的攻击 . . . . .                 | 183        |
|                                       |     | 17.4 总结 . . . . .                              | 184        |
|                                       |     | 17.5 参考文献 . . . . .                            | 184        |



## *Chapter No. 1*

---

### 概述

---



*Chapter No. 2*

---

*Internet* 地址结构

---





*Chapter No. 3*

---

链路层

---



## *Chapter No. 4*

---

### 地址解析协议

---



## *Chapter No. 5*

---

### *Internet 协议*

---



## *Chapter No. 6*

---

### 系统配置： *DHCP* 和自动配置

---





*Chapter No. 7*

---

防火墙和网络地址转换

---



*Chapter No. 8*

---

*ICMPv4 和 ICMPv6: Internet 控制报文协议*

---



*Chapter No. 9*

---

广播和本地组播 (*IGMP* 和 *MLD*)

---



## *Chapter No. 10*

---

### 用户数据报协议和 *IP* 分片

---





*Chapter No. 11*

---

名称解析和域名系统

---



# TCP: 传输控制协议

---

## 12.1 引言

到目前为止，我们一直在讨论那些自身不包含可靠传递数据机制的协议。它们可能会使用一种想校验和或 CRC 这样的数学函数来检测接收到的有差错的数据，但是它们不尝试去纠正差错。对于 IP 和 UDP，根本没有差错纠正。对于以太网和基于其上的其他协议，协议提供一定次数的重试，如果还是不成功则放弃。

通信媒介可能会丢失或改变所传递的消息，在这种环境下的通信问题已经被研究了多年。关于这个课题的一些最重要的理论工作由克劳德·香农在 1948 年给出。这些工作普及了术语“比特”，并成为信息理论 (information theory) 领域的基础，帮助我们理解了在一个有损（可能删除或改变比特）信道里可通过的信息量的根本限制。信息理论与编码理论 (coding theory) 的领域密切相关，编码理论提供不同的信息编码手段，从而使得信息能够在通信信道里尽量免于出错。使用差错校正码（基本上是添加一些冗余的比特，使得即使某些比特被毁，真实的信息也可以被恢复过来）来纠正通信问题是处理差错的一种非常重要的方法。另一种方法是简单地“尝试重新发送”，直到消息最终被接收。这种方法，被称为自动重复请求 (Automatic Repeat Request, ARQ)，构成了许多通信协议的基础，包括 TCP 在内。

### 12.1.1 ARQ 和重传

如果我们考虑的不只是单个通信信道，而是几个的多跳级联，我们会发现不只会碰到前面提到的那几种差错类型（分组比特差错），而且还会有更多其他的类型。这些问题可能发生在中间路由器上，是几种在讨论 IP 时会遇到的问题：分组重新排序，分组复制，分组泯灭（丢失）。为在多跳通信信道（例如 IP）上使用而设计的带纠错的协议必须要处理这些问题。现在让我们来探讨能处理这些问题的协议机制。在概括性地讨论这些之后，我们会探究它们是如何被 TCP 在

互联网上使用的。

一个直接处理分组丢失（和比特差错）的方法是重发分组直到它被正确接收。这需要一种方法来判断：（1）接收方是否已收到分组；（2）接收方接收到的分组是否与之前发送方发送的一样。接收方给发送方发信号以确定自己已经接收到一个分组，这种方法称为确认（acknowledgment），或 ACK。最基本的形式是，发送方发送一个分组，然后等待一个 ACK。当接收方接收到这个分组时，它发送对应的 ACK。当发送方接收到这个 ACK，它再发送另一个分组，这个过程就这样继续。这里会有一些有意思的问题：（1）发送方对一个 ACK 应该等待多长时间？（2）如果 ACK 丢失了怎么办？（3）如果分组被接收到了，但是里面有错怎么办？

正如我们将看到的，第一个问题其实挺深奥的。决定去等待多长时间与发送方期待（expect）为一个 ACK 等待多长时间有关。现在确定这个时间可能比较困难，因此我们推迟对这个技术的讨论，直到我们在后面（见第 14 章）详细讨论 TCP。第二个问题的答案比较容易：如果一个 ACK 丢失了，发送方不能轻易地把这种情况与原分组丢失的情况区分开来，所以它简单地再次发送原分组。当然，这样的话，接收方可能会接收到两个或更多的拷贝，因此它必须准备好处理这种情况（见下一段）。至于第三个问题，我们可以借助在 12.1 节中提到的编码技术来解决。使用编码来检测一个大的分组中的差错（有很大的概率）一般都很简单，仅使用比其自身小很多的一些比特即可纠正。更简单的编码一般不能纠正差错，但是能检测它们。这就是校验和与 CRC 会如此受欢迎的原因。然后，为了检测分组里的差错，我们使用一种校验和形式。当一个接收方接收到一个含有差错的分组时，它不发送 ACK。最后，发送方重发完整到达的无差错的分组。

到目前为止即使这种简单的场景，接收方都可能接收到被传送分组的重复（duplicate）副本。这个问题要使用序列号（sequence number）来处理。基本上，在被源端发送时，每个唯一的分组都有一个新的序列号，这个序列号由分组自身一直携带着。接收方可以使用这个序列号来判断它是否已经见过这个分组，如果见过则丢弃它。

到目前为止介绍的协议是可靠的，但效率不太高。如果从发送方到接收方传递即使一个很小的分组都要用很长时间（推迟或延迟）的话（如一秒或两秒，对卫星链路来说并非不正常），考虑一下那会怎样。发送方可以注入一个分组到通信路径，然后停下来等待直到它收到 ACK。这个协议因此被称为“停止和等待”。假设没有分组在传输中丢失和无可挽回地损害，该协议的吞吐量性能（每单位时间发送在网络中的数据量）与  $M/R$  成正比， $M$  是分组大小， $R$  是往返时间（RTT）。如果有分组丢失和损害的话，情况甚至更糟糕：“吞吐质”（每单位时间传送的有用数据量）明显要比吞吐量要低。

对于不会损害和丢失太多分组的网络来说，低吞吐量的原因是网络经常没有处于繁忙状态。情况与使用装配流水线时不出一个完整产品就不准新的工作进入类似。流水线大部分时间是空闲的。我们进一步对比，很明显，如果我们允许同一时间有多个工作单元进入流水线，就可以做得更好。对网络通信来说也是一样的——如果我们允许多个分组进入网络，就可以使它“更繁忙”，从而得到更高的吞吐量。

很明显，允许多个分组同时进入网络使事情变得复杂。现在发送方必须不仅要决定什么时间注入一个分组到网络中，还要考虑注入多少个。并且必须要指出在等待 ACK 时，怎样维持计时器，同时还必须要保存每个还没确认的分组的一个副本以防需要重传。接收方需要有一个更复杂的 ACK 机制：可以区分哪些分组已经收到，哪些还没有。接收方可能需要一个更复杂的缓存（分组保存）机制——允许维护“次序杂乱”的分组（那些比预想要先到的分组更早到达的分组，因为丢包和次序重排的原因），除非简单地抛弃这些分组，而这样做是很没效率的。还有其他一些没有这么明显的问题。如果接收方的接收速率比发送方的发送速率要慢怎么办？如果发送方简单地以很高的速率发送很多分组，接收方可能会因处理或内存的限制而丢掉这些分组。中间的路由器也会有相同的问题。如果网络基础设施处理不了发送方和接收方想要使用的数据发送率怎么办？

### 12.1.2 分组窗口和滑动窗口

为了解决所有这些问题，我们以假设每个分组有一个序列号开始，正如前面所描述的。我们定义一个分组窗口（window）作为已被发送方注入但还没完成确认（如，发送方还从没收到过它们的 ACK）的分组（或者它们的序列号）的集合。我们把这个窗口中的分组数量称为窗口大小（window size）。术语窗口来自这样的想法：如果你把在一个通信对话中发送的所有分组排成长长的一行，但只能通过一个小孔来观察它们，你就只能看到它们的一个子集——像通过一个窗口观看一样。发送方的窗口（以及其他分组队列）可画图描述成 12-1 那样。

发送方窗口，显示了哪些分组将要被发送（或已经发送），哪些尚来发送，以及哪些已经发送并确认。在这个例子里，窗口大小被确定三个分组

这个图显示了当前三个分组的窗口，整个窗口大小是 3。3 号分组已经被发送和确认，所以由发送方保存的它的副本可以被释放。分组 7 在发送方已经准备好，但还没被发送，因为它还没“进入”窗口。现在如果我们想象数据开始从发送方流到接收方，ACK 开始以相反的方向流动，发送方可能下一步就接收到一个分组 4 的 ACK。当这发生时，窗口向右边“滑动”一个分组，意味着分组 4 的副本可以释放了，而分组 7 可以被发送了。窗口的这种滑动给这种类型的协议增加了一个名字，滑动窗口（sliding window）协议。

这种滑动窗口方法可用于对付到目前为止描述过的许多问题。一般来说，这个窗口结构在发送方和接收方都会有。在发送方，它记录着哪些分组可被释放，哪些分组正在等待 ACK，以及哪些分组还不能被发送。在接收方，它记录着哪些分组已经被接收和确认，哪些分组是下一步期望的（和已经分配多少内存来保存它们），以及哪些分组即使被接收也将会因内存限制而被丢弃。尽管窗口结构便于记录在发送方和接收方之间流动的数据，但是关于窗口应多大，或者如果接收方或者网络处理不过来发送方的数据率时会发生什么，它都没有提供指导建议。现在我们应该看看这些怎样关联在一起。

### 12.1.3 变量窗口：流量控制和拥塞控制

为了处理当接收方相对发送方太慢时产生的问题，我们介绍一种方法，在接收方跟不上时会强迫发送方慢下来。这称为流量控制（flow control），该控制经常以下述两种方式之一来进行操作。一种方式称为基于选率（rate-based）流量控制，它是给发送方指定某个速率，同时确保数据永远不能超过这个速率发送。这种类型的流量控制最适合流应用程序，可被用于广播和组播发现（见第 9 章）。

另一种流量控制的主要形式叫基于窗口（window-based）流量控制，是使用滑动窗口时最流行的方法。在这种方法里，窗口大小不是固定的，而是允许随时间而变动的。为了使用这种技术进行流量控制，必须有一种方法让接收方可以通知发送方使用多大的窗口。这一般称为窗口通告（window advertisement），或简单地称为窗口更新（window update）。发送方（即窗口通告的接收者）使用该值调整其窗口大小。逻辑上讲，一个窗口更新是与我们前面讨论过的 ACK 分离的，但是实际上窗口更新和 ACK 是由同一个分组携带的，意味着发送方往往会在它的窗口滑动到右边的同时调整它的大小。

如果我们考虑到在发送方修改窗口大小会带来的影响，就可以很明显地知道这是怎样达到流量控制的。在没收到它们中任何一个的 ACK 之前，发送方允许注入  $W$  个分组到网络中。如果发送方和接收方足够快，网络中设有丢失一个分组以及有无穷的空间的话，这意味粉通信速率正比于  $(SW/R)$  b/S，这里  $W$  是窗口大小， $S$  是分组大小（按比特算）， $R$  是往返时间（RTT）。当来自接收方的窗口通告夹带着发送方的位  $W$  时，那么发送方的全部速率就被限制而不能超越接收方。这种方法可以很好地保护接收方，但是对于中间的网络呢？在发送方和接收方之间可能会有有限内存的路由器，它们与低速网络链路抗争着。当这种情况出现时，发送方的速率可能超过某个路由器的能力，从而导致丢包。这由一种特殊的称为拥塞控制（congestion control）的流量控制形式来处理。

拥塞控制涉及发送方减低速度以不至于压垮其与接收方之间的网络。回想我们关于流量控制的讨论，我们使用一个窗口通告来告之发送方为接收方减慢速度。这称为明确（explicit）发信，因为有一个协议字段专用于通知发送方正在发生什么。另一个选项可能被发送方用于猜测（guess）它需要慢下来。这种方法涉及隐性（implicit）发信——涉及根据其他某些证据来决定减慢速度。

数据报类型的网络或更一般的与之关联的排队理论（queuing theory）中的拥塞控制问题仍然是这些年的一个主要研究课题，它甚至不太可能完全解决所有情况。在这里讨论关于流量控制的所有选择和方法也并不现实。有兴趣的读者可参考。在第 16 章我们将更详细地探讨实际用于 TCP 中的拥塞控制技术，以及这些年来出现的许多变体。

### 12.1.4 设置重传超时

基于重传的可靠协议的设计者要面对的一个最重要的性能问题是，要等待多久才能判定一个分组已丢失并将它重发。用另一种方式说，重传超时应该是多大？直观上看，发送方在重发一个分组之前应等待的时间量大概是下面时间的总和：发送分组所用的时间，接收方处理它和发送一个 ACK 所用的时间，ACK 返回到发送方所用的时间，以及发送方处理 ACK 所用的时间。不幸的是，实际上这些时间没有一个是确切知道的。更糟的是，它们中的某些或全部会随着来自终端主机或路由器的额外负载的增加或减少而随时改变。

让用户去告诉协议实现在所有情况下的每个时刻应取什么超时时间（或使它们保持最新），这是不现实的，一个更好的策略是让协议实现尝试去估计它们。这称为往返时间估计（round-trip-time estimation），这是一个统计过程。总的来说，选择一组 RTT 样本的样本均值作为真实的 RTT 是最有可能的。注意到这个平均值很自然地会随着时间而改变（它不是静态的），因为通信穿过网络的路径可能会改变。

做出对 RTT 的一些估计之后，关于设置实际的用于触发重传的超时取值问题依然存在。如果我们回想一下均值的定义，会知道它绝不可能是一组样本的极值（除非它们全部一样）。所以，把重传计时器的值设置成正好等于平均估计量是不合理的，因为很有可能许多实际的 RTT 将会比较大，从而会导致不必要的重传。很明显，超时应该设置成比均值要大的某个值，但是这个值与均值的确切关系是什么（或者甚至直接就使用均值）还不清楚。超时设算得太大也是不可取的，因为这反过来会导致网络变得空闲，从而降低吞吐率。对这个话题的进一步探究留到第 14 章，我们在那里会探讨 TCP 是怎样实际地处理这个问题的。

## 12.2 TCP 的引入

我们现在对影响可靠传输的问题有了大体的了解，下面看一下它们是怎样在 TCP 中体现的，以及 TCP 会给互联网应用程序提供什么类型的服务。我们还要查看 TCP 头部中的字段，了解有多少个到目前为止我们已经见过的概念（如 ACK、窗口通告）可在头部描述中捕捉到。在随后的章节，我们会更详细地检查所有这些头部字段。

我们对 TCP 的描述从这一章开始，并在接下来的五章中继续讨论。第 13 章描述一个 TCP 连接是怎样建立和结束的。第 14 章详细说明 TCP 是怎样估计每个连接的 RTT 和怎样基于这个估计设置重传超时的。第 15 章考察正常的数据传输，以“交互式”应用程序开始（例如聊天程序），然后是窗口管理和流量控制，这被应用于交互式 and “大块”数据流（比如文件传输）两种应用程序以及 TCP 的紧急机制（urgent mechanism）——它允许发送方指定数据流中的某些数据作为特殊数据。第 16 章考察 TCP 里的拥塞控制算法，这些算法在网络很繁忙的时候帮助降低丢包率。这一章还讨论了一些改动，这些改动被提出来以增加快速网络的吞吐量或改进易损耗（如无线）网络的弹性。最后，第 17 章显示 TCP 如何在没有数据流动时保持连接的活动性。

TCP 的原始规范是 [RFC0793]，尽管这个 RFC 的一些错误已经在主机请求 RFC 中被修改过来 [RFC1122]。从那以后，TCP 的一些规范就一直被修改和扩展以包含透明和改进的拥塞控制行为 [RFC5681] [RFC3782] [RFC3517] [RFC3390] [RFC3168]、重传超时 [RFC6298] [RFC5682] [RFC4015]、在 NAT 的操作 [RFC5382]、确认行为 [RFC2883]、安全 [RFC6056] [RFC5927] [RFC5926]、连接管理 [RFC5482] 以及紧急机制实现指导方针 [RFC6093]。同时还有丰富的实验性修改，覆盖了重传行为 [RFC5827] [RFC3708]、拥塞检测和控制 [RFC5690] [RFC562] [RFC4782] [RFC3649] [RFC2861] 以及其他特性。最后，在探究 TCP 如何利用多重并发网络层路径方面也做了工作 [RFC6182]。

### 12.2.1 TCP 服务类型

虽然 TCP 和 UDP 使用相同的网络层 (IPv4 或 IPv6)，但是 TCP 给应用程序提供了一种与 UDP 完全不同的服务。TCP 提供了一种面向连接的 (connection-oriented)、可靠的字节流服务。术语“面向连接的”是指使用 TCP 的两个应用程序必须在它们可交换数据之前，通过相互联系来建立一个 TCP 连接。最典型的比喻就是拨打一个电话号码，等待另一方接听电话并说“喂”，然后再说“找谁？”。这正是一个 TCP 连接的两个端点在互相通信。像广播和组播（见第 9 章）这些概念在 TCP 中都不存在。

TCP 提供一种字节流抽象概念给应用程序使用。这种设计方案的结果是，没有由 TCP 自动插入的记录标志或消息边界（见第 1 章）。一个记录标志对应着一个应用程序的写范围指示。如果应用程序在一端写入 10 字节，随后写入 20 字节，再随后写入 50 字节，那么在连接的另一端的应用程序是不知道每次写入的字节是多少的。例如，另一端可能会以每次 20 字节分四次读入这 80 字节或以其他一些方式读入。一端给 TCP 输入字节流，同样的字节流会出现在另一端。每个端点独立选择自己的读和写大小。

TCP 根本不会解读字节流里的字节内容。它不知道正在交换的数据字节是不是二进制数据、ASCII 字符、EBCDIC 字符或其他东西。对这个字节流的解读取决于连接中的每个端点的应用程序。尽管不再推荐使用，可 TCP 确实是支持以前提到过的紧急机制。

### 12.2.2 TCP 中的可靠性

通过使用刚才描述过的那些技术的特定变种，TCP 提供了可靠性。因为它提供一个字节流接口，TCP 必须把一个发送应用程序的字节流转换成一组 IP 可以携带的分组。这被称为组包 (packetization)。这些分组包含序列号，该序列号在 TCP 中实际代表了每个分组的第一个字节在整个数据流中的字节偏移，而不是分组号。这允许分组在传送中是可变大小的，并允许它们组合，称为重新组包 (repacketization)。应用程序数据被打散成 TCP 认为的最佳大小的块来发送，一般使得每个报文段按照不会被分片的单个 IP 层数据报的大小来划分。这与 UDP 不同，应用程序每次写入通常就产生一个 UDP 数据，其大小就是写入的那么大（加上头部）。由 TCP 传给



IP 的块称为报文段 (segment, 见图 12-2)。在第 15 章我们会看到 TCP 如何判定一个报文段的大小。

TCP 维持了一个强制的校验和, 该校验和涉及它的头部、任何相关应用程序数据和 IP 头部的所有字段。这是一个端到端的伪头部校验和, 用于检测传送中引入的比特差错。如果一个带无效校验和的报文段到达, 那么 TCP 会丢弃它, 不为被丢弃的分组发送任何确认。然而, TCP 接收端可能会对一个以前的 (已经确认的) 报文段进行确认, 以帮助发送方计算它的拥塞控制 (见第 16 章)。TCP 校验和使用的数学函数与其他互联网协议 (UDP、ICMP 等) 一样。对于大数据的传送, 对这个校验和是否不够强壮的担心是存在的 [SP00], 所以仔细的应用程序应该应用自己的差错保护方法 (如, 更强的校验和或 CRC), 或者使用一种中间层来达到同样的效果 (如, 见 [RFC5044])。

当 TCP 发送一组报文段时, 它通常设置一个重传计时器, 等待对方的确认接收。TCP 不会为每个报文段设置一个不同的重传计时器。相反, 发送一个窗口的数据, 它只设置一个计时器, 当 ACK 到达时再更新超时。如果有一个确认没有及时接收到, 这个报文段就会被重传。在第 14 章我们将更详细地查看 TCP 的自适应超时和重传策略。

当 TCP 接收到连接的另一端的数据时, 它会发送一个确认。这个确认可能不会立即发送, 而一般会延迟片刻。TCP 使用的 ACK 是累积的, 从某种意义上讲, 一个指示字节号 N 的 ACK 暗示着所有直到 N 的字节 (但不包含 N) 已经成功被接收了。这对于 ACK 丢失来说带来了一定的鲁棒性—如果一个 ACK 丢失, 很有可能后续的 ACK 就足以确认前面的报文段了。

TCP 给应用程序提供一种双工服务。这就是说数据可向两个方向流动, 两个方向互相独立。因此, 连接的每个端点必须对每个方向维持数据流的一个序列号。一旦建立了一个连接, 这个连接的一个方向上的包含数据流的每个 TCP 报文段也包含了相反方向上的报文段的一个 ACK。每个报文段也包含一个窗口通告以实现相反方向上的流量控制。为此, 在一个连接中, 当一个 TCP 报文段到达时, 窗口可能向前滑动, 窗口大小可能改变, 同时新数据可能已到达。正如我们将在第 13 章所见, 一个完整的 TCP 连接是双向和对称的, 数据可以在两个方向上平等地流动。

使用序列号, 一个 TCP 接收端可丢弃重复的报文段和记录以杂乱次序到达的报文段。回想一下, 任何反常情况都会发生, 因为 TCP 使用 IP 来传递它的报文段, IP 不提供重复消除或保证次序正确的功能。然而, 因为 TCP 是一个字节流协议, TCP 绝不会以杂乱的次序给接收应用程序发送数据。因此, TCP 接收端可能会被迫先保持大序列号的数据不交给应用程序, 直到缺失的小序列号的报文段 (一个“洞”) 被填满。

我们现在开始观察 TCP 的一些细节。这一章将只介绍 TCP 的封装和头部结构, 其他细节出现在后面的五章中。TCP 可与 IPv4 或 IPv6 一起使用, 同时它使用的伪头部校验和 (与 UDP 的类似) 在 IPv4 中和 IPv6 中都是强制使用的。

## 12.3 TCP 头部和封装

图 12-2 显示了 TCP 在 IP 数据报中的封装。TCP 头部紧跟着 IP 头部或 IPv6 扩展头部，经常是 20 字节长（不带 TCP 选项）。带选项的话，TCP 头部可达 60 字节。常见选项包括最大段大小、时间戳、窗口缩放和选择性 ACK

头部本身明显要比在第 10 章我们见过的 UDP 的头部更复杂。这并不很令人惊讶，因为 TCP 是一个明显要更复杂的协议，它必须保持连接的每一端知道（同步）最新状态。如图 12-3 所示。

TCP 头部。它的标准长度是 20 字节，除非出现选项。头部长度（Header Length）字段以 32 位字为单位给出头部的大小（最小值是 5）。带阴影的字段（确认号（Acknowledgment Number）、窗口大小（Window Size）以及 ECE 位和 ACK 位）用于与该报文段的发送方关联的相反方向上的数据流

每个 TCP 头部包含了源和目的端口号。这两个值与 IP 头部中的源和目的 IP 地址一起，唯一地标识了每个连接。在 TCP 术语中，一个 IP 地址和一个端口的组合有时被称为一个端点（endpoint）或套接字（socket）。后者出现在 [RFC0793] 中，最终被 Berkeley 系列的网络通信编程接口所采用（现在经常被称为“Berkeley 套接字”）。每个 TCP 连接由一对套接字或端点（四元组，由客户机 IP 地址、客户机端口号、服务器 IP 地址以及服务器端口号组成）唯一地标识。这个事实在我们观察一个 TCP 服务器是如何做到与多个客户机通信的时候将会变得很重要（见第 13 章）。

序列号（Sequence Number）字段标识了 TCP 发送端到 TCP 接收端的数据流的一个字节，该字节代表着包含该序列号的报文段的数据中的第一个字节。如果我们考虑在两个应用程序之间的一个方向上流动的数据流，TCP 给每个字节赋予一个序列号。这个序列号是一个 32 位的无符号数，到达  $2^{32}-1$  后再循环回到 0。因为每个被交换的字节都已编号，确认号字段（也简称 ACK 号或 ACK 字段）包含的值是该确认号的发送方期待接收的下一个序列号。即最后被成功接受的数据字节的序列号加 1。这个字段只有在 ACK 位字段被启动的情况下才有效，这个 ACK 位字段通常用于除了初始和来尾报文段之外的所有报文段。发送一个 ACK 是发送任何一个 TCP 报文段的开销是一样的，因为那个 32 位的 ACK 号字段一直都是头部的一部分，ACK 位字段也一样。

当建立一个新连接时，从客户机发送至服务器的第一个报文段的 SYN 位字段被启用。这样的报文段称为 SYN 报文段，或简单地称为 SYN。然后序列号字段包含了在本次连接的这个方向上要使用的第一个序列号，后续序列号和返回的 ACK 号也在这个方向上（回想一下，连接都是双向的）。注意这个数字不是 0 和 1，而是另一个数字，经常是随机选择的，称为初始序列号（Initial Sequence Number, ISN）。ISN 不是 0 和 1，是因为这是一种安全措施，将会在第 13 章讨论。发送在本次连接的这个方向上的数据的第一个字节的序列号是 ISN 加 1，因为 SYN 位字段

会消耗一个序列号。正如我们稍后将见到的，消耗一个序列号也意味着使用重传进行可靠传输。因此，SYN 和应用程序字节（还有 FIN，稍后我们将会见到）是被可靠传输的。不消耗序列号的 ACK 则不是。

TCP 可以被描述为“一种带累积正向确认的滑动窗口协议”。ACK 号字段被构建用于指明在接收方已经顺序收到的最大字节（加 1）。例如，如果字节 1 1024 已经接收成功，而下一个报文段包含字节 2049 3072，那么接收方不能使用规则的 ACK 号字段去发信告诉发送方它接收到了这个新报文段。然而，现代 TCP 有一个选择确认（Selective ACKnowledgment, SACK）选项，可以允许接收方告诉发送方它正确地接收到了次序杂乱的数据。当与一个具有选择重发（selective repeat）能力的 TCP 发送方搭配时，就可以实现性能的显著改善 [FF96]。在第 14 章我们将会看到 TCP 是如何使用重复确认（duplicate acknowledgments）以帮助它的拥塞控制和差错控制过程的。

头部长度的字段给出了头部的长度，以 32 位字为单位。它是必需的，因为选项字段的长度是可变的。作为一个 4 位的字段，TCP 被限制为只能带 60 字节的头部。而不带选项，大小是 20 字节。

当前，为 TCP 头部定义了 8 位的字段，尽管一些老的实现只理解它们中的最后 6 位<sup>o</sup>。它们中的一个或多个可被同时启用。我们在这里大致提一下它们的用法，在后面的几章里再对每个进行详细的讨论。

1. CWR——拥塞窗口减小（发送方降低它的发送速率）；见第 16 章。
2. ECE—ECN 回显（发送方接收到了一个更早的拥塞通告）；见第 16 章。
3. URG 紧急（紧急指针字段有效——很少被使用）；见第 15 章。
4. ACK——确认（确认号字段有效——连接建立以后一般都是启用状态）；见第 13 章和第 15 章。
5. PSH——推送（接收方应尽快给应用程序传送这个数据——没被可靠地实现或用到）；见第 15 章。
6. RST—重置连接（连接取消，经常是因为错误）；见第 13 章。
7. SYN——用于初始化一个连接的同步序列号；见第 13 章。
8. FIN——该报文段的发送方已经结束向对方发送数据；见第 13 章。

TCP 的流量控制由每个端点使用窗口大小字段来通告一个窗口大小来完成。这个窗口大小是字节数，从 ACK 号指定的，也是接收方想要接收的那个字节开始。这是一个 16 位的字段，限制了窗口大小到 65535 字节，从而限制了 TCP 的吞吐量性能。在第 15 章我们将看到窗口缩放

(Window Scale) 选项可允许对这个值进行缩放, 给高速和大延迟网络提供了更大的窗口和改进性能。

TCP 校验和字段覆盖了 TCP 的头部和数据以及头部中的一些字段, 使用一个与我们在第 8 章和第 10 章讨论的 ICMPv6 与 UDP 使用的相类似的伪头部进行计算。这个字段是强制的, 由发送方进行计算和保存, 然后由接收方验证。TCP 校验和的计算算法与 ICMP 和 UDP (“互联网”) 校验和一样。

紧急指针 (Urgent Pointer) 字段只有在 URG 位字段被设置时才有效。这个“指针”是一个必须要加到报文段的序列号字段上的正偏移, 以产生紧急数据的最后一个字节的序列号。TCP 的紧急机制是一种让发送方给另一端提供特殊标志数据的方法。

最常见的选项字段就是“最大段大小”选项, 称为 MSS。连接的每个端点一般在它发送的第一个报文段 (为了建立该连接, SYN 位字段被设置的那个报文段) 上指定这个选项。MSS 指定该选项的发送者在相反方向上希望接收到的报文段的最大值。在第 13 章我们将更详细地描述 MSS 选项, 并在第 14 章和第 15 章描述其他一些 TCP 选项。我们考查的其他普通选项还包括 SACK、时间戳和窗口缩放。

在图 12-2 中我们注意到 TCP 报文段的数据部分是可选的。在第 13 章我们将看到当一个连接被建立和终止时, 交换的报文段只包含 TCP 头部 (带或不带选项) 而没有数据。如果这个方向上没有数据被传输, 那么一个不带任何数据的头部也会被用于确认接收到的数据 (称为一个纯 (pure) ACK), 同时通知通信方改变窗口大小 (称为一个窗口更新 (window update))。当一个报文段可不带数据发送时, 超时操作会因此而产生一些新情况。

## 12.4 总结

在有损通信信道上提供可靠通信的问题已经被研究了许多年。处理差错的两种主要方法是差错校正码和数据重传。使用重传的协议必须也要处理数据丢失, 经常通过设置一个计时来进行, 同时还必须要给接收方安排一些方法来告知发送方它已接收了什么。判定等待一个 ACK 要多长时间是比较棘手的, 因为合适的时间会随着网络路由或端点上负载的变动而改变。现代协议用基于这些测量值的一些函数来估计往返时间以及设置重传计时器。

不考虑设置重传计时器的话, 当同一时间只有一个分组在网络中时, 重传协议是很简单, 但对于延迟很高的网络, 它们的性能会很差。为了更有效率, 在一个 ACK 被接收到之前, 多个分组必须被注入网络中。这种方法更有效率, 但也更复杂。一种管理这些复杂性的典型方法是使用滑动窗口, 其中分组用序列号标志, 窗口大小限制分组数量。当窗口大小基于来自接收方或其他信号 (比如被丢弃的分组) 的回馈而改变时, 流量控制和拥塞控制两者就都被实现了。

TCP 提供一种可靠、面向连接、字节流、传输层的服务 (通过使用许多这些技术而构建)。我们简单地看了 TCP 头部里的所有字段, 了解到它们中的大多数都与这些可靠传递的抽象概念有

着直接关系。我们将在接下来的章节里详细考查它们。TCP 把应用程序数据组包成报文段，发送数据时设置超时，确认被其他端点接收到的数据，给次序杂乱的数据进行重新排序，丢弃重复的数据，提供端到端的校验和。TCP 在互联网中被广泛使用，不仅许多流行的应用程序使用它、例如 HTTP、SSH、Telnet、FTP 以及电子邮件 (SMTP)，许多分布式文件共享程序（如，BitTorrent, Shareaza）也使用它。

## 12.5 参考文献



# TCP: 连接管理

---

## 13.1 引言

TCP 是一种面向连接的单播协议。在发送数据之前，通信双方必须在彼此间建立一条连接。本章将详细地介绍 TCP 连接的概念，以及它的建立与终止过程。如前文所述，TCP 服务模型是一个字节流。TCP 必须检测并修补所有在 IP 层（或下面的层）产生的数据传输问题，比如丢包、重复以及错误。

由于需要对连接状态进行管理（通信双方都需要维护连接的信息），TCP 被认为是一个比 UDP 协议（参见第 10 章）复杂得多的协议。UDP 是一种无连接的协议，因此它不需要连接的建立与终止过程。与 UDP 相比，TCP 在妥善处理多种 TCP 状态时需要面对大量的细节问题，比如一个连接何时建立、正常地终止，以及在无警告的情况下重新启动。因此，这也被认为是两个协议的主要区别之一。在后续章节，我们将探讨一旦建立连接并传输数据将会发生什么。

在连接建立的过程中，通信双方需要交换一些选项。这些选项被认为是连接的参数。一些选项只被允许在连接建立时发送，而其他一些选项则能够稍后发送。根据第 12 章的介绍，TCP 头部已设置了一个有限的空间（40 字节）来处理这些选项。

## 13.2 TCP 连接的建立与终止

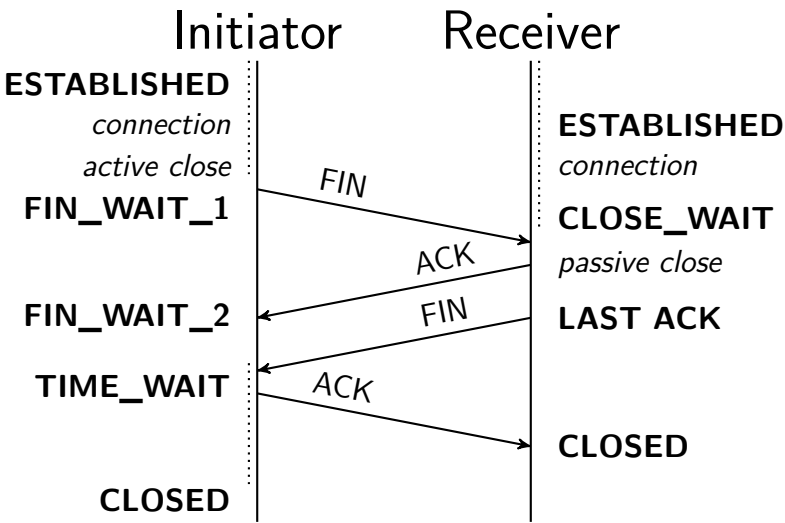
一个 TCP 连接由一个 4 元组构成，它们分别是两个 IP 地址和两个端口号。更准确地说，一个 TCP 连接是由一对端点或套接字构成，其中通信的每一端都由一对（IP 地址，端口号）所唯一标识。

一个 TCP 连接通常分为 3 个阶段：启动、数据传输（也称作“连接已建立”）和退出（关闭）。下文我们将会发现正确地处理上述三个阶段之间的转换是创建一个强健的 TCP 连接的困难所

在。图 13-1 显示了一个典型的 TCP 连接的建立与关闭过程（不包括任何数据传输）。

图 13-1 中的时间轴描绘了一个连接建立过程中的相关事宜。为了建立一个 TCP 连接，需要完成以下步骤：

- 主动开启者（通常称客户端）发送一个 SYN 报文段（即一个在 TCP 头部的 SYN 位字段置位的 TCP/IP 数据包），并指明自己想要连接的端口号和它的客户端初始序列号（记为 ISN (c)，参见13.2.3）。通常，客户端还会借此发送一个或多个选项（参见13.3节）。客户端发送的这个 SYN 报文段称作段 1。
- 服务器也发送自己的 SYN 报文段作为响应，并包含了它的初始序列号（记作，ISN (S)）。该段称作段 2。此外，为了确认客户端的 SYN，服务器将其包含的 ISN (c) 数值加 1 后作为返回的 ACK 数值。因此，每发送一个 SYN，序列号就会自动加 1。这样如果出现丢失的情况，该 SYN 段将会重传。
- 为了确认服务器的 SYN，客户端将 ISN (S) 的数值加 1 后作为返回的 ACK 数值。这称作段 3。



一个普通 TCP 连接的建立与终止。通常，由客户端负责发起一个三次握手过程。在该过程中，客户端与服务器利用 SYN 报文段交换彼此的初始序列号（包括客户端的初始序列号和服务器的初始序列号）。在通信双方都发送了一个 FIN 数据包并收到来自对方的相应的确认数据包后，该连接终止

通过发送上述 3 个报文段就能够完成一个 TCP 连接的建立。它们也常称作三次握手。三次握手的目的不仅在于让通信双方了解一个连接正在建立，还在于利用数据包的选项来承载特殊的信息，交换初始序列号（Initial Sequence Number, ISN）。

发送首个 SYN 的一方被认为是主动地打开一个连接。如上文所述，它通常是一个客户端。连接的另一方会接收这个 SYN，并发送下一个 SYN，因此它是被动地打开一个连接。通常，这



一方称为服务器。(13.2.2 节将会介绍一种客户端与服务器同时打开一个连接的情况。这种情况可以作为上文所介绍内容的补充,但非常少见。)

### 注意

TCP 的 SYN 段也能够承载应用数据。由于伯克利的套接字 API 不支持这种方式,因此它也很少为人所用。

图 13-1 还描绘了一个 TCP 连接是怎样关闭的(也称清除或终止)。连接的任何一方都能够发起一个关闭操作。此外,该过程还支持双方同时关闭连接的操作,但这种情况非常少见。在传统的状况下,负责发起关闭连接的通常是客户端(如图 13-1 所示)。然而,一些服务器(例如 Web 服务器)在对请求做出响应之后也会发起一个关闭操作。通常一个关闭操作是由应用程序提出关闭连接的请求而引发的(例如使用系统调用 `close()`)。TCP 协议规定通过发送一个 FIN 段(即 FIN 位字段置位的 TCP 报文段)来发起关闭操作。只有当连接双方都完成关闭操作后,才构成一个完整关闭:

1. 连接的主动关闭者发送一个 FIN 段指明接收者希望看到的自己当前的序列号(K,如图 13-1 所示)。FIN 段还包含了一个 ACK 段用于确认对方最近一次发来的数据(图 13-1 中标记为 L)。
2. 连接的被动关闭者将 K 的数位加 1 作次响应的 ACK 位,以表明它已经成功接收到主动关闭者发送的 FIN。此时,上层的应用程序会被告知连接的另一端已经提出了关闭的请求。通常,这将导致应用程序发起自己的关闭操作。接着,被动关闭者将身份转变为主动关闭者,并发送自己的 FIN。该报文段的序列号为 L。
3. 为了完成连接的关闭,最后发送的报文段还包含一个 ACK 用于确认上一个 FIN。值得注意的是,如果出现 FIN 丢失的情况,那么发送方将重新传输直到接收到一个 ACK 确认为止。

综上所述,建立一个 TCP 连接需要 3 个报文段,而关闭一个 TCP 连接需要 4 个报文段。TCP 协议还支持连接处于半开启状态(参见 13.6.3 节),但这种情况并不常见。存在上述半开启状态的原因在于 TCP 的通信模型是双向的。这也意味着在两个方向中可能会出现只有一个方向正在进行数据传输的情况。TCP 的半关闭操作是指仅关闭数据流的一个传输方向,而两个半关闭操作合在一起就能够关闭整个连接。因此 TCP 协议规定通信的任何一方在完成数据发送任务后都能够发送一个 FIN。当通信的另一方接收到这个 FIN 时,就会告知应用程序对方已经终止了对应方向的数据传输。由此可见,当程序发布关闭操作请求后,通信双方往往通过发送 FIN 段来关闭双向的数据传输。

如上文所述,7 个报文段是每一个 TCP 连接在正常建立与关闭时的基本开销(下文还会介绍一些突然关闭 TCP 连接的方式)。因此当只需要交换少量的数据时,一些应用程序更愿意选

择在发送与接收数据之前不需要建立连接的 UDP 协议。然而，这些应用程序也会面对由此引入的错误修复、拥塞管理以及流量控制等诸多问题。

### 13.2.1 TCP 半关闭

如前文所述，TCP 支持半关闭操作。虽然一些应用需要此项功能，但它并不常见。为了实现这一特性，API 必须为应用程序提供一种基本的表达方式。例如，应用程序表明“我已经完成了数据的发送工作，并发送一个 FIN 给对方，但是我仍然希望接收来自对方的数据直到它发送一个 FIN 给我”。伯克利套接字的 API 提供了半关闭操作。应用程序只需要调用 `shutdown()` 函数来代替基本的 `close()` 函数，就能实现上述操作。然而，绝大部分应用程序仍然会调用 `close()` 函数来同时关闭一条连接的两个传输方向。图 13-2 展示了一个正在使用的半关闭示例。图中左侧的客户端负责发起半关闭操作，然而在实际应用中，通信的任何一方都能完成这项工作。

首先发送的两个报文段与 TCP 正常关闭完全相同：初始者发送的 FIN，接着是接收者回应该 FIN 的 ACK。由于接收到半关闭的一方仍能够发送数据，因此图 13-2 中的后续操作与图 13-1 不同。虽然图 13-2 在 ACK 之后只描述了一个数据段的传输过程，但实际应用时可以传输任意数址的数据段（第 1S 章将会详细地讨论数据段的交换与确认细节）。当接收半关闭的一方完成数据发送后，它将会发送一个 FIN 半关闭本方的连接，同时向发起半关闭的应用程序签出一个文件尾指示。当第 2 个 FIN 被确认之后，整个连接完全关闭。

### 13.2.2 同时打开与关闭

虽然两个应用程序同时主动打开连接看似不大可能，但是在特定安排的情况下是有可能实现的。通信双方在接收到来自对方的 SYN 之前必须先发送一个 SYN；两个 SYN 必须经过网络送达对方。该场景还要求通信双方都拥有一个 IP 地址与端口号，并且将其告知对方。上述情况十分少见（第 7 章介绍的防火墙“打孔”技术除外），一旦发生，可称其同时打开。

例如，主机 A 的一个应用程序通过本地的 7777 端口向主机 B 的 8888 端口发送一个主动打开请求，与此同时主机 B 的一个应用程序也通过本地的 8888 端口向主机 A 的 7777 端口提出一个主动打开请求，此时就会发生一个同时打开的情况。这种情况不同于主机 A 的一个客户端连接主机 B 的一个服务器，而同时又有主机 B 的一个客户端连接主机 A 的一个服务器的情况。在这种情况下，服务器始终是连接的被动打开者而非主动打开者，而各自的客户端也会选择不同的端口号。因此，它们可以被区分为两个不同的 TCP 连接。图 13-3 显示了在一个同时打开过程中报文段的交换情况。

一个同时打开过程需要交换 4 个报文段，比普通的三次握手增加了一个。由于通信双方都扮演了客户端与服务器的角色，因此不能够将任何一方称作客户端或服务器。同时关闭并没有太大区别。如前文所述，通信一方（通常是客户端，但不一定总是）提出主动关闭请求，并发送首

个 FIN。在同时关闭中，通信双方都会完成上述工作。图 13-4 显示了一个同时关闭中需要交换的报文段。

在同时打开中交换的报文段。与正常的连接建立过程相比，需要增加一个报文段。数据包的 SYN 位将置位直到接收到一个 ACK 数据包止

在同时关闭中交换的报文段。与正常关闭相似，只是报文段的顺序是交叉的

同时关闭需要交换与正常关闭相同数量的报文段。两者真正的区别在于报文段序列是交叉的还是顺序的。下文将会介绍 TCP 实现中同时打开与同时关闭操作使用特殊状态这一不常见的方法。

### 13.2.3 初始序列号

当一个连接打开时，任何拥有合适的 IP 地址、端口号、符合逻辑的序列号（即在窗口中）以及正确校验和的报文段都将被对方接收。然而，这也引入了另一个问题。在一个连接中，TCP 报文段在经过网络路由后可能会存在延迟抵达与排序混乱的情况。为了解决这一问题，需要仔细选择初始序列号。本节将详细介绍这一过程。

在发送用于建立连接的 SYN 之前，通信双方会选择一个初始序列号。初始序列号会随时间而改变，因此每一个连接都拥有不同的初始序列号。[\[RFC793\]](#) 指出初始序列号可被视为一个 32 位的计数器。该计数器的数值每 4 微秒加 1。此举的目的在于为一个连接的报文段安排序列号，以防止出现与其他连接的序列号重叠的情况。尤其对于同一连接的两个不同实例而言，新的序列号也不能出现重叠的情况。

由于一个 TCP 连接是被一对端点所唯一标识的，其中包括由 2 个 IP 地址与 2 个端口号构成的 4 元组，因此即便是同一个连接也会出现不同的实例。如果连接由于某个报文段的长时间延迟而被关闭，然后又以相同的 4 元组被重新打开，那么可以相信延迟的报文段又会被视为有效数据重新进入新连接的数据流中。上述情况会令人十分烦恼。通过采取一些步骤来避免连接实例间的序列号重叠问题，能够将风险降至最低。即便如此，一个对数据完整性有较高要求的应用程序也可以在应用层利用 CRC 或校验和保证所需数据在传输过程中没有出现任何错误。在任何情况下这都是一种很好的方法，并已普遍用于大文件的传输。

如前文所述，一个 TCP 报文段只有同时具备连接的 4 元组与当前活动窗口的序列号，才会在通信过程中被对方认为是正确的。然而，这也从另一个侧面反映了 TCP 的脆弱性：如果选择合适的序列号、IP 地址以及端口号，那么任何人都能伪造出一个 TCP 报文段，从而打断 TCP 的正常连接 [\[RFC5961\]](#)。一种抵御上述行为的方法是使初始序列号（或者临时端口号 [TRFC6056](#)）变得相对难以被猜出，而另一种方法则是加密（参见第 18 章）。

现代系统通常采用半随机的方法选择初始序列号。证书报告 CA-2001-09 [\[CERTISN\]](#) 讨论了这一方法的具体实现细节。Linux 系统采用一个相对复杂的过程来选择它的初始序列号<sup>1</sup>。它采用基于时钟的方案，并且针对每一个连接为时钟设置随机的偏移量。随机偏移量是在连接标识

(即 4 元组)的基础上利用加密散列函数得到的。散列函数的输入每隔 5 分钟就会改变一次。在 32 位的初始序列号中, 最高的 8 位是一个保密的序列号, 而剩余的各位则由散列函数生成。上述方法所生成的序列号很难被猜出, 但依然会随着时间而逐步增加。据报告显示, Windows 系统使用了一种基于 RC4 [S94] 的类似方案。

### 13.2.4 例子

前文介绍了一个 TCP 连接的建立和退出过程, 本节将从数据包(分组)的角度进一步介绍相关细节。为此我们尝试对邻近的 Web 服务器进行 TCP 连接。该主机的 IPv4 地址为 10.0.0.2, 而客户端则采用了基于 Windows 的 `telnet` 应用。

`telnet`命令是建立在 TCP 连接的基础上的。在上述例子中, 该 TCP 连接必须与服务器的 IPv4 地址 10.0.0.2 以及 http 或 Web 服务的端口号(80 端口)相关联。当 `telnet` 应用程序连接 23 (`telnet` 协议的众所周知端口 [RFC0854]) 以外的端口, 它将不能用于应用协议。它仅仅将自己的字节输入拷贝至 TCP 连接中, 反之亦然。当一个 Web 服务器接收到进入的连接请求时, 它首先需要等待对 web 页面的请求。在这种情况下, 我们不能提供这样的请求, 因此服务器不会产生任何数据。这些均符合我们的期望, 因为我们只对连接建立与终止过程中的数据包交换感兴趣。图 13-5 展示了 Wireshark 软件对该命令所产生的报文段的输出结果。

如图 13-5 所示, 客户端发送的 SYN 报文段所包含的初始序列号为 685506836, 通告窗口为 65535。该报文段还包含了若干其他选项。13.3 节将详细地讨论这些选项。第二个报文段既包含了服务器的 SYN 还包含了对客户端请求的 ACK 确认。它的序列号(服务器的初始序列号)为 1479690171, ACK 号为 685506837。ACK 号仅比客户端的初始序列号大 1, 说明服务器已经成功接收到了客户端的初始序列号。该报文段同样也包含了一个通告窗口以表明服务器愿意接收 64240 个字节。第三个数据包将最终完成三次握手, 它的 ACK 为 1479690172。ACK 号是不断累积的, 并且总是表明 ACK 发送者希望接收到的下一个序列号(而不是它上一个接收到的序列号)。

在 4.4 秒暂停之后, `telnet` 应用程序被要求关闭连接。这使得客户端发送第 4 个报文段 FIN。FIN 的序列号为 685506837, 并由第 5 个报文段确认(ACK 号为 685506838)。稍后, 服务器会发送自己的 FIN, 对应的序列号为 1479690172。该报文段对客户端的 FIN 进行了再次确认。值得注意的是, 该报文段的 PSH 位被置位。虽然这样并不会对连接的关闭过程产生实质影响, 但通常用于说明服务器将不会再发送任何数据。最后一个报文段用于对服务器的 FIN 进行确认, ACK 号为 1479690173。

注意 [RFC1025] 将拥有最多特性(例如标记与选项)的报文段称为“神风”(kamikaze)数据包。其他生动的术语还包括“丑恶报文”、“圣诞树数据包”、“灯测试”报文段。

从图 13-5 中我们还会发现 SYN 报文段包含了一个或多个选项。这些选项需要占用 TCP 头

---

<sup>1</sup>详细代码见 [linux isn generate](#), linux 4.4 中使用的散列函数为 `md5`

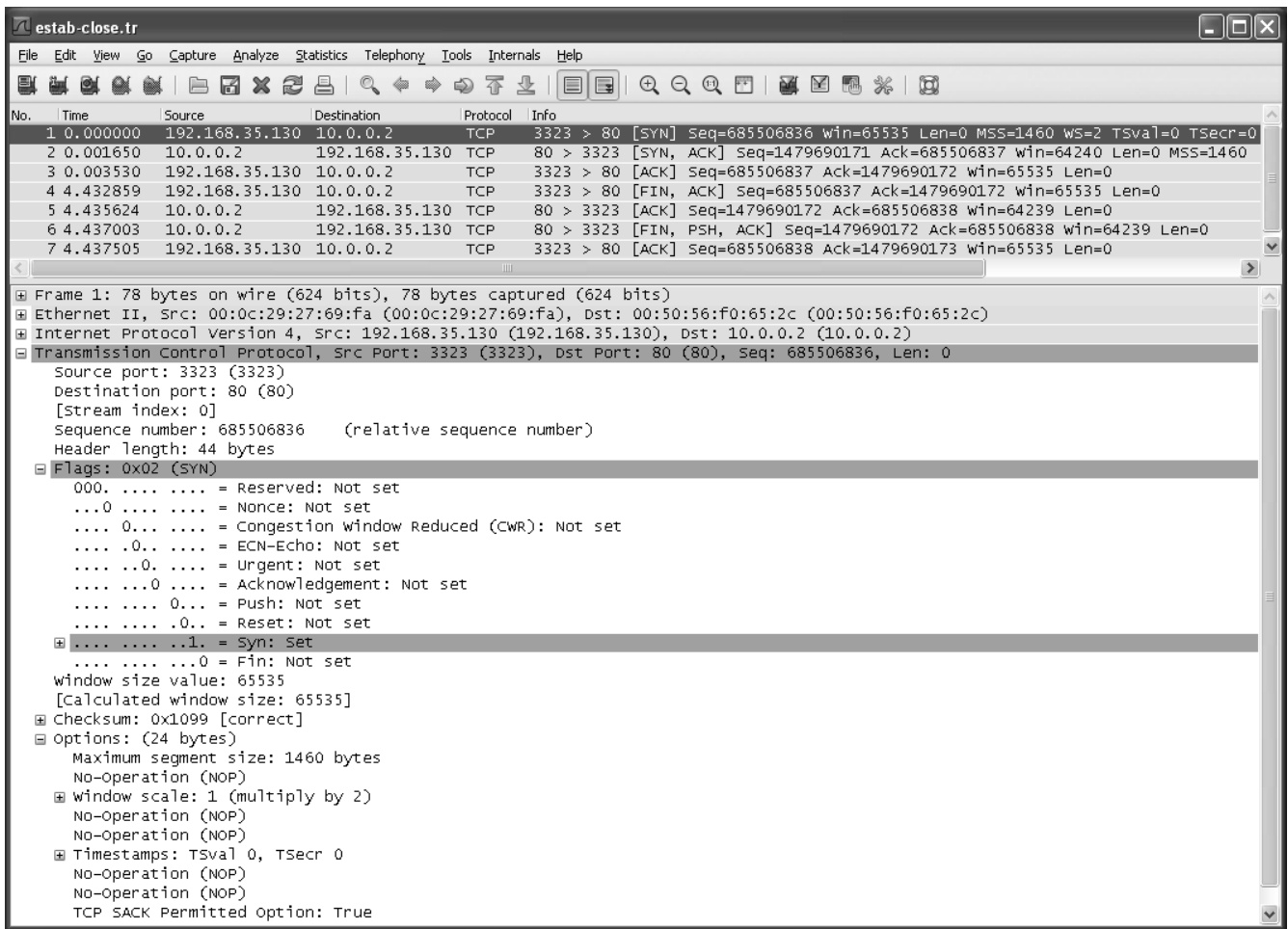


图 13.1: 在主机 192.168.35.130 与 10.0.0.2 之间建立一条 TCP 连接，并不发送任何数据的情况下关闭。PSH（推送）位说明第 6 个报文段正在发送所有来自缓存的数据（而缓存为空）

部额外的空间。例如，第一个 TCP 头部的长度为 44 字节，比最小的长度长 24 字节。TCP 也提供了若干选项，下文将详细介绍当一个连接无法建立时如何使用这些选项。

### 13.2.5 连接建立超时

本节的若干实例将会展示连接不能建立的情况。一种显而易见的情况是服务器关闭。为了模拟这种情况，我们将 `telnet` 命令发送给一个处于同一子网的不存在的主机。在不修改 ARP 表的情况下，上述做法会使客户端收到一个“无法到达主机”的错误消息后退出。由于没有接收到针对之前发送的 ARP 请求而返回的 ARP 响应（参见第 4 章），因此会产生“无法到达主机”的消息。如果我们能事先在 ARP 表中为这个不存在的主机添加一条记录，那么系统就不需要发送 ARP 请求，而会马上根据 TCP/IP 协议尝试与这个不存在的主机建立联系。相关的命令如下：

上述例子选择的 MAC 地址 00:00:1a:1b:1c:1d 不能与局域网中其他主机的 MAC 冲突，除此

之外并无特别。超时发生在发送初始命令后的 3.2 分钟。由于没有主机响应，例子中所有的报文段都是由客户端产生的。清单 13-1 显示了使用 Wireshark 软件在摘要模式下获得的输出结果。

有趣的是这些输出结果显示了客户端 TCP 为了建立连接频繁地发送 SYN 报文段。在首个报文段发送后仅 3 秒第二个报文段就被发送出去，第三个报文段则是这之后的 6 秒，而第四个报文段则在第三个报文段发送 12 秒以后被发送出去，以此类推。这一行为被称作指数回退。在讨论以太网 CSMA/CD 介质访问控制协议时（参见第 3 章）我们曾见过这样的行为。然而，这两种指数回退也略有不同。此处的每一次回退数值都是前一次数值的两倍，而在以太网中最大的回退数值是上一次的两倍，实际的回退数值则需要随机选取。

一些系统可以配置发送初始 SYN 的次数，但通常选择一个相对较小的数值 5。在 Linux 系统中，系统配置变量 `net.ipv4.tcp_syn_retries` 表示了在一次主动打开申请中尝试重新发送 SYN 报文段的最大次数。相应地，变量 `net.ipv4.tcp_synack_retries` 表示在响应对方的一个主动打开请求时尝试重新发送 SYN+ACK 报文段的最大次数。此外，它能够在设定 Linux 专有的 TCP\_SYNCNT 套接字选项的基础上用于个人连接。正如上面所介绍的，默认的数值为重试 5 次。两次重新传输之间的指数回退时间是 TCP 拥塞管理响应的一部分。当我们讨论 Kam 算法时再仔细研究。

### 13.2.6 连接与转换器

在第 7 章，我们已经讨论了一些协议（比如 TCP 和 UDP）如何利用传统的 NAT 转换地址与端口号。我们还讨论了 IP 数据包如何在 IPv6 与 IPv4 两个版本间进行转换。当 TCP 使用 NAT 时，伪头部的校验和通常需要调整（使用校验和中立地址修改器的情况除外）。其他协议也使用伪头部校验和，因为计算包含了与传输层、网络层相关的信息。

当一个 TCP 连接首次被建立时，NAT 能够根据报文段的 SYN 位探明这一事实。同样可以通过检查 SYN+ACK 报文段与 ACK 报文段所包含的序列号来判断一个连接是否已经完全建立。上述方法还适用于连接的终止。通过在 NAT 中实现一部分 TCP 状态机（参 5IRFC6146] 的 3.5.2.1 节与 3.5.2.2 节）能够跟踪连接，包括当前状态、各方向的序列号以及相关的 ACK 号。这种状态跟踪是典型的 NAT 实现方法。

当 NAT 扮演编辑者的角色并且向传输协议的数据负载中写入内容时，就会涉及一些更复杂的问题。对于 TCP 而言，它将会包括在数据流中添加与删除数据，并由此影响序列号（与报文段）的长度。此举会影响到校验和，但也会影响数据的顺序。如果利用 NAT 在数据流中插入或删除数据，这些数值都要做出适当调整。如果 NAT 的状态与终端主机的状态不同步，连接就无法正确进行下去。因此，上述做法会带来一定的脆弱性。

## 13.3 TCP 选项

如图 12-3 所示，TCP 头部包含了多个选项。选项列表结束 (End of Option List, EOL) 无操作 (No Operation, NOP) 以及最大段大小 (Maximum Segment Size, MSS) 是定义于原始 TCP 规范中的选项。自那时起，又有若干选项被定义。整个选项列表是由互联网编号分配机构 (IANA) 维护的 [TPARAMS]。表 13-1 列举了一些目前有趣的选项 (即，符合 RFC 标准化描述的选项)。

每一个选项的头一个字节为“种类” (kind)，指明了该选项的类型。根据 [RFC1122]，不能被理解的选项会被简单地忽略掉。种类值为 0 或 1 的选项仅占用一个字节。其他的选项会根据种类来确定自身的字节数 len。选项的总长度包括了种类与 len 个字节。设置 NOP 选项的目的是允许发送者在必要的时候用多个 4 字节组填充某个字段。需要记住的是 TCP 头部的长度应该是 32 比特的倍数，因为 TCP 头部长度字段是以此为单位的。EOL 指出了选项列表的结尾，说明无需对选项列表再进行处理。在下文中，我们将详细地探究其他选项。

### 13.3.1 最大段大小选项

最大段大小是指 TCP 协议所允许的从对方接收到的最大报文段，因此这也是通信对方在发送数据时能够使用的最大报文段。根据 [RFC0879]，最大段大小只记录 TCP 数据的字节数而不包括其他相关的 TCP 与 IP 头部。当建立一条 TCP 连接时，通信的每一方都要在 SYN 报文段的 MSS 选项中说明自己允许的最大段大小。这 16 位的选项能够说明最大段大小的数值。在没有事先指明的情况下，最大段大小的默认数值为 596 字节。前文曾介绍过，任何主机都应该能够处理至少 576 字节的 IPv4 数据报。如果按照最小的 IPv4 与 TCP 头部计算，TCP 协议要求在每次发送时的最大段大小为 536 字节，这样就正好能够组成一个  $576(20 + 20 + 536 = 576)$  字节的 IPv4 数据报。

图 13-5 中，最大段大小的数值均为 1460。这是 IPv4 协议中的典型值，因此 IPv4 数据数据报的大小也相应增加 40 个字节 (总共 1500 字节，以太网中最大传输单元与互联网路径最大传输单元的典型数值)：20 字节的 TCP 头部加 20 字节的 IP 头部。当使用 IPv6 协议时，最大段大小通常为 1440 字节。由于 IPv6 的头部比 IPv4 多 20 个字节，因此最大段大小的数值相应减少 20 字节。在 [RFC2675] 中 65535 是一个特数值，与 IPv6 超长数据报一起用来指定一个表示无限大的有效最大段大小值。在这种情况下，发送方的最大段大小等于路径 MTU 的数值减去 60 字节 (40 字节用于 IPv6 头部，20 字节用于 TCP 头部)。值得注意的是，最大段大小并不是 TCP 通信双方的协商结果，而是一个限定的数值。当通信的一方将自己的最大段大小选项发送给对方时，它已表明自己不愿意在整个连接过程中接收任何大于该尺寸的报文段。



### 13.3.2 选择确认选项

第 12 章介绍了滑动窗口的概念, 并描述了 TCP 协议是如何管理序列号与确认的。由于采用累积 ACK 确认, TCP 不能正确地确认之前已经接收的数据。由于接收的数据是无序的, 所以接收到数据的序列号也是不连续的。在这种情况下, TCP 接收方的数据队列中会出现空洞的情况。因此在提供字节流传输服务时, TCP 接收方需要防止应用程序使用超出空洞的数据。

如果 TCP 发送方能够了解接收方当前的空洞 (以及在序列空间中超出空洞的乱序数据块), 它就能在报文段丢失或被接收方遗漏时更好地进行重传工作。根据 [RFC2018] 与 [RFC2883], TCP“选择确认”(SACK) 选项提供了上述功能。如果 TCP 接收方能够提供选择确认信息, 并且发送方能够合理有效地利用这些信息, 那么上述方案将会十分高效。

通过接收 SYN (或者 SYN +ACK) 报文段中的“允许选择确认”选项, TCP 通信方会了解到自身具有了发布 SACK 信息的能力。当接收到乱序的数据时, 它就能提供一个 SACK 选项来描述这些乱序的数据, 从而帮助对方有效地进行重传。SACK 信息保存于 SACK 选项中, 包含了接收方已经成功接收的数据块的序列号范围。每一个范围被称作一个 SACK 块, 由一对 32 位的序列号表示。因此, 一个 SACK 选项包含了  $n$  个 SACK 块, 长度为  $(8n + 2)$  个字节。增加的 2 个字节用于保存 SACK 选项的种类与长度。

由于 TCP 头部选项的空间是有限的, 因此一个报文段中发送的最大 SACK 块数目为 3 (假设使用了时间戳选项。根据 13.3.4 节的介绍, 这是现代 TCP 实现中的典型情况)。虽然只有 SYN 报文段才能包含“允许选择确认”选项, 但是只要发送方已经发送了该选项, SACK 块就能够通过任何报文段发送出去。由于选择确认的操作相对于 TCP 的错误和拥塞控制的操作而言更简单, 本书将在第 14 章与第 16 章再介绍相关的细节。

### 13.3.3 窗口缩放选项

根据 [RFC1323], 窗口缩放选项 (表示 WSCALE 或 WSOPT) 能够有效地将 TCP 窗口广告字段的范围从 16 位增加至 30 位。TCP 头部不需要改变窗口通告字段的大小, 仍然维持 16 位的数值。同时, 使用另一个选项作为这 16 位数值的比例因子。该比例因子能够使窗口字段值有效地左移。这样事实上将窗口数值大至原先的  $2^n$  倍, 其中  $n$  为比例因子。一个字节的移动可以用 0 至 14 (包含 14) 来计数。计数的移动表示没有任何比例。最大的比例数值是 14, 它能够提供一个最大为 1 073 725 440 字节 ( $65535 \times 2^{14}$ ) 的窗口。该数值接近  $1073741823(2^{30} - 1)$ , 正好 1GB。因此, TCP 使用一个 32 位的值来维护这个“真实”的窗口大小。

该选项只能出现于一个 SYN 报文段中, 因此当连接建立以后比例因子是与方向绑定的。为了保证窗口调整, 通信双方都需要在 SYN 报文段中包含该选项。主动打开连接的一方利用自己的 SYN 中发送该选项, 但被动打开连接的一方只能在接收到的 SYN 中指出该选项时才能发送。每个方向的比例因子可各不相同。如果主动打开连接的一方发送了一个非 0 的比例因子但却没



有接收到来自对方的窗口缩放选项，它会将自己发送与接收的比例因子数值都设为 0。这样使得系统不需要理解这些系统间的选项互操作。

假设我们正在使用窗口缩放选项，发送出去的窗口移动数值  $S$ ，而接收到的窗口移动数值为  $R$ 。这样，我们从对方接收到每一个 16 位的广告窗口都需要左移  $R$  位才能获得真实窗口大小。每次向对方发送窗口通告时，都会将 32 位的窗口大小向右移动  $S$  位，然后将 16 位的数值填充到 TCP 头部。

窗口的移动数值是由 TCP 通信方根据接收缓存的大小自动选取的。缓存的大小是由系统设定的，但是应用程序通常都具有改变其大小的能力。当 TCP 协议被用于在大带宽、高延迟网络（即，往返时间与带宽都相对较大的网络）上提供海量数据传输服务时，窗口缩放选项就非常有意义。因此，第 16 章将会进一步讨论该选项的重要性与使用方法。

### 13.3.4 时间戳选项与防回绕序列号

时间戳选项（记作 `TSOPT` 或 `TSopt`）要求发送方在每一个报文段中添加 2 个 4 字节的时间戳数值。接收方将会在确认中反映这些数值，允许发送方针对每一个接收到的 ACK 估算 TCP 连接的往返时间（由于 TCP 协议经常利用一个 ACK 来确认多个报文段，此处必须指出是“每个接收到的 ACK”而不是“每个报文段”。本书第 15 章将会详细地讨论这一问题）。当使用时间戳选项时，发送方将一个 32 位的数值填充到时间戳数值字段（称作 `TSV` 或 `TSval`）作时间戳选项的第一个部分；而接收方则将收到的时间戳数值原封不动地填充至第二部分的时间戳回显重试字段（称作 `TSBR` 或 `TSecr`）。由于包含了时间戳选项，TCP 头部的长度将会增长 10 字节（8 字节用于保存 2 个时间戳数值，而另 2 个数值则用于指明选项的数值与长度）。

时间戳是一个单调增加的数值。由于接收者只会对它接收到的信息做出响应，所以它并不关心时间戳单元或数值到底是什么。该选项并不要求在两台主机之间进行任何形式的时钟同步。[RFC1323] 推荐发送者每秒钟至少将时间戳数值加 1。图 13-6 显示了通过 Wireshark 软件获得的时间戳选项。

上述例子中，通信双方都产生并回应了对方的时间戳。第一个报文段（客户端的 SYN）使用一三个初始的时间戳数位 81813090。该数值被填充在时间戳数值子段中。由于客户端并不知道服务器的时间戳数值，所以该报文段的第二部分时间戳回显重试字段的数值为 0。

一个使用了时间戳、窗口缩放以及最大段大小选项的 TCP 连接。TCP 头部长度为 44 字节。初始 SYN（第 1 个数据包）的时间戳数值为 81813090。图中高亮标记的第二个数据包将这一数值作为响应返回给主动打开连接的一方，同时包含了自己的数值 349742014。

估算一条 TCP 连接的往返时间主要是为了设置重传超时。重传超时用于告知 TCP 通信方何时应该重新发送可能已经丢失的报文段。第 12 章已经讨论了在一些往返时间函数的基础上设置此项超时的必要性。借助时间戳选项，我们能够获得往返时间相对精确的测量结果。在使用时间戳选项之前，大多 TCP 通信会针对每个窗口的数据抽取一个往返时间样本。时间戳选项使我

们获得了更多的样本，从而提升了精确估算往返时间的能力（参见 [RFC1323] 与 [RFC6298]）。

由于时间戳选项与重传计时器的设置紧密相关，我们将在第 14 章讨论重传问题时详细地介绍时间戳选项的这一用途。“这一用途”是为了强调虽然时间戳选项允许更高频率的往返时间样本，但它也为接收者提供了避免接收旧报文段与判断报文段正确性的方法。这被称作防回绕序列号（Protection Against Wrapped Sequence numbers, PAWS），它与时间戳选项一起记录于 [RFC1323] 中。现在，我们要继续探究一下它是如何工作的。

假设一个 TCP 连接使用了窗口缩放选项，并将其设置为可能最大的窗口，大约 1GB。再假设使用了时间戳选项，并且发送者针对发送的每个窗口分配的时间戳数值都会加 1。（这一假设是保守的。正常情况下时间戳数值的增长速度要远快于此。）表 13-2 显示了当传输 6GB 数据时两个主机之间可能的数据流。为了避免过多的 10 位数字，我们使用符号 G 表示  $1073\ 741\ 824$  的倍数。我们还再次采用了 tcpdump 中的符号 J:K 来表示从第 J 个字节到第 K-1 个字节的数据。

TCP 时间戳选项通过提供一个额外的 32 位有效序列号空间清除了具有相同序列号的报文段之间的二义性

32 位序列号字段在时刻 D 和时刻 E 间回绕。假设在时刻 B 有一个报文段丢失并被重传。又假设这一丢失的报文段在时刻 F 重新出现。假设报文段丢失与重新出现的时间差小于一个报文段在网络中存在的最大时间（称为 MSL，参见 13.5.2 节），否则当路由器发现 TTL 期满后就会丢弃该报文段。正如我们之前提到的，旧的报文段重新出现并包含当前正在传输的序列号的问题只会发生在相对高速的连接中。

由表 13-2 可以看出，使用时间戳选项能够有效地防止上述问题。接收者可以将时间戳看作一个 32 位的扩展序列号。丢失的报文段会在时刻 F 重新出现，由于它的时间戳为 2，小于最近的有效时间戳（5 或 6），因此防回绕序列号算法会将其丢弃。防回绕序列号算法并不要求在发送者与接收者之间有任何形式的时钟同步。接收者所需要的是保证时间戳数值单调增长，并且每一个窗口的数据至少增加 1。

### 13.3.5 用户超时选项

根据 [RFC5482] 的描述，用户超时（UTO）选项是一个相对较新的 TCP 的功能。用户超时数值（也被称作 USER\_TIMEOUT）指明了 TCP 发送者在确认对方未能成功接收数据之前愿意等待该数据 ACK 确认的时间。根据 [RFC0793]，USER\_TIMEOUT 是 TCP 协议本地配置的一个参数。用户超时选项允许 TCP 通信方将自己的 USER\_TIMEOUT 数值告知连接的对方。这样就方便了 TCP 接收方调整自己的行为（例如，在终止连接之前容忍一段较长时间的连接中断）。NAT 设备也能够解释这些信息以帮助设置它们的连接活动计时器。

用户超时选项的数值是建议性的，因为即便连接的一端希望使用一个大的或小的数值也不意味着另一端就必须遵从。[RFC1122] 提炼了 USER\_TIMEOUT 的定义，并且建议当 TCP 连接达到 3 次重传阈值时应该通知应用程序（规则 1），而当超时大于 100 秒时应该关闭连接（规则

2)。某些实现会提供 API 函数来修改规则 1 与规则 2。由于长的用户超时设置会导致资源耗尽，而短的用户超时设置可能会导致一些连接过早地断开（例如，拒绝服务攻击），因此需要为用户超时选项的可能数值设置上下边界。设置 USER\_TIMEOUT 的具体方法如下： $USER\_TIMEOUT = \min(U\_LIMIT, \max(ADV\_UTO, REMOTE\_UTO, L\_LIMIT))$

其中 ADV\_UTO 是本端告知远端通信方的用户超时选项数值，而 \_UTO 是远端通信方告知的用户超时选项数值，U\_LIMIT 是本地系统对用户超时选项设定的数值上边界，而 L\_LIMIT 则是下边界。值得注意的是上式并不能保证同一连接的两端会获得相同的用户超时数值。在任何情况下，L\_LIMIT 的数值必须大于对应连接的重传超时数值（参见第 1 章）。L\_LIMIT 的数值一般推荐设为 100 秒，这样可以保持与『RFC1122』相兼容。

建立连接的 SYN 报文段、首个非 SYN 报文段以及 USER\_TIMEOUT 的数值发生任何改变的报文段，都会包含用户超时选项。该选项的数值由一个 15 位的数值部分与一个 1 位的单位部分构成。单位部分用于说明数值的计量单位是分钟（1）还是秒（0）。UTO 作为一个相对较新的选项还没有得到广泛使用。

### 13.3.6 认证选项

TCP 设置了一个选项用于增强连接的安全性。设计该选项的目的在于增强与替换较早的 TCP-MD5 机制 [RFC2385]。这一选项被称作 TCP 认证选项（TCP Authentication Option, TCP-AO）[RFC5925]，它使用了一种加密散列算法（参见第 18 章）以及 TCP 连接双方共同维护的一个秘密值来认证每一个报文段。TCP 认证选项不仅提供各种加密算法，还使用“带内”信令来确认密钥是否改变，因此它与 TCP-MD5 相比有很大的提高。然而，TCP 认证选项没有提供一个全面密钥管理方案。也就是说，通信双方不得不采用一种方法在 TCP 认证选项运行之前建立出一套共享密钥。

当发送数据时，TCP 会根据共享的密钥生成一个通信密钥，并根据一个特的加密算法[RFC5926]计算散列值。接收者装配有相同的密钥，同样也能够生成通信密钥。借助通信密钥，接收者可以确认到达的报文段是否在传输过程中被篡改过（有非常高的可能性）。设置该选项是为了针对各种 TCP 欺骗攻击提供强有力的抵御策略（参见 13.8 节）。然而，由于需要创建并分发一个共享密钥（这是一个相对新的选项），该选项并没有得到广泛使用。

## 13.4 TCP 的路径最大传输单元发现

第 3 章介绍了路径最大传输单元（MTU）的概念。它是指经过两台主机之间路径的所有网络报文段中最大传输单元的最小值。知道路径最大传输单元后能够有助于一些协议（比如 TCP）避免分片。第 10 章介绍了基于 ICMP 消息的路径最大传输单元发现（PMTUD）过程是如何实现的，但由于应用程序已经指定了尺寸（即，非传输层协议），UDP 协议一般不会采用上述发现

过程获得的数据报大小。TCP 在支持字节流抽象的实现过程中能够决定使用多大的报文段，因此它很大程度上控制了最后生成的 IP 数据包。

在本节中，我们将会进一步探寻 TCP 是如何使用路径最大传输单元的。我们的讨论内容适用于 TCP/IPv4 与 TCP/IPV6。[RFC1191] 与 [RFC1981] 能分别提供更多的细节。一种称作分组层路径最大传输单元发现（Packetization Layer Path MTU Discovery, PLPMTUD）的算法能够避免对 ICMP 的使用。根据 [RFC4821]，该算法也能够被 TCP 或其他传输协议使用。我们可以利用 IPv6 协议中“数据包太大”（Packet Too Big, PTB）的术语来代表 ICMPv4 地址不可达（需要分片）或 ICMPv6 数据包太大的消息。

TCP 常规的路径最大传输单元发现过程如下：在一个连接建立时，TCP 使用对外接口的最大传输单元的最小值，或者根据通信对方声明的最大段大小来选择发送方的最大段大小（SMSS）。路径最大传输单元发现不允许 TCP 发送方有超过另一方所声明的最大段大小的行为。如果对方没有指明最大段大小的数值，发送方将假设采用默认的 536 字节，但是这种情况比较少见。如果为每一个目的地保存对应的路径最大传输单元，那么就能方便地对段大小进行选择。值得注意的是，一条连接的两个方向的路径最大传输单元是不同的。

一旦为发送方的最大大小选定了初始值，TCP 通过这条连接发送的所有 IPv4 数据报都会对 DF 位字段进行设置。TCP/IPv6 没有 DF 位字段，因此只需要假设所有的数据报都已经设置了该字段而不必进行实际操作。如果接收到 PTB 消息，TCP 就会减少段的大小，然后用修改过的段大小进行重传。如果在 PTB 消息中已包含了下一跳推荐的最大传输单元，段大小的数值可以设置为下一跳最大传输单元的数值减去 IPv4（或 IPv6）与 TCP 头部的大小。如果下一跳最大传输单元的数值不存在（例如，一个之前的 ICMP 错误被返回时会缺乏这一信息），发现者可能需要尝试多个数值（例如，采用二分搜索法选择一个可用的数值）。这也会影响到 TCP 的拥塞控制管理（参见第 16 章）。对于分组层路径最大传输单元发现而言，除了 PTB 的消息不被使用以外其他情况基本类似。相反，执行路径最大传输单元发现的协议必须能够快速地检测消息丢弃并调整自己的数据报大小。

由于路由是动态变化的，在减少段大小的数值一段时间后需要尝试一个更大的数值（接近初始的发送方最大段大小）。根据 [RFC1191] 与 [RFC1981] 的指导意见，该时间间隔大约为 10 分钟。

在互联网环境中，由于防火墙阻塞 PTB 消息 [RFC2923]，路径最大传输单元发现过程会存在一些问题。在各种操作问题中，黑洞问题的情况虽有所好转（在 [LS10] 中，80% 被调查的系统都能够正确地处理 PTB 消息），但仍悬而未决。在 TCP 实现依靠传输 ICMP 消息来调整它的段大小的情况下，如果 TCP 从未接收到任何 ICMP 消息，那么在路径最大传输单元发现过程中就会造成黑洞问题。这种情况可能由多方面的原因造成，其中包括了防火墙或 NAT 配置为禁止转发 ICMP 消息。其后果在于一旦 TCP 使用了更大的数据包将不能被正确处理。由于只是不能转发大数据包，所以诊断出这一问题是十分困难的。那些较小的数据包（比如用于建立连接

的 SYN 与 SYN +ACK 数据包) 是能够成功处理的。一些 TCP 实现具有“黑洞探测”功能。当一个报文段在反复重传数次后, 将会尝试发送一个较小的报文段。

### 13.4.1 例子

当中间路由器的最大传输单元小于任何一个通信端的最大段大小时, TCP 就会执行路径最大传输单元发现过程。为了创造上述条件, 本文使用一台路由器 (一台本地地址为 10.0.0.1 的 Linux 主机) 通过 PPPoE 接口连接 DSL 服务提供商。PPPoE 链路使用的最大传输单元为 1492 字节 (以太网的最大传输单元为 1500 字节, 减去 PPPOE 协议的 6 字节负载, 再减去 2 字节的 PPP 负载。参见第 13 章)。图 13-7 显示了上述例子的拓扑结构。

PPPoE 封装使 TCP 连接的路径最大传输单元从 1500 字节 (以太网最大传输单元的典型值) 减至 1492 字节。为了证明 TCP 的路径 MTU 发现功能, 此例设置了更小的最大传输单元 (288 字节)

为了更加明显地看出这一行为, 将 PPPOE 链路最大传输单元的数值以 1492 减少至 288 字节。通过在图 13-7 的 GW 主机上执行下面的命令来完成这项工作:

此外, 还需要告知客户端 (图 13-7 中的 c) 系统允许的最小报文段大小:

如果不执行第 2 步操作, Linux 系统会将路径最大传输单元的最小值设置为默认的 552 字节, 这样就能够避免某些较小的最大传输单元攻击 (参见 13.8 节)。这样做的后果是此例中任何大于 288 字节的数据包都将被分片。为了避免这种情况的发生, 并证明路径最大传输单元发现是有效的, 需要修改这一最小值。然后, 我们开始通过互联网从主机 C (地址 10.0.0.123) 向服务器 S 传输文件 (地址 169.229.62.97)。清单 13-2 显示了利用 tcpdump 记录下的数据包交换过程。为了清楚起见, 隐藏了一些行并删除了一些不相关的字段。

在网络传输过程中, 如果中间链路的最大传输单元小于两端通信节点的, 路径 MTU 发现机制能够找出合适的段大小以供传输使用

根据 tcpdump 的输出结果, 连接已经建立并且最大段大小的数值已经交换。连接中所有的数据包都将 DF 位置位, 所以两端都能够使用路径 MTU 发现机制。较远一方的首个数据包的长度为 588 字节。尽管中间 PPPoE 链路的最大传输单元已配置为 288 字节, 但该数据包仍能够成功地通过路由器转发而不被分片。产生这种情况的原因是不对称的最大传输单元配置。虽然 PPPoE 链路的本地端使用了 288 字节的最大传输单元, 但是另一端仍然将发送方最大段大小的数值设置为一个较大的值, 大概是 1492 字节。这样就会造成下述情况, 向外发送的数据包需要较小的段大小 (288 字节或更小), 而反方向进入的数据包可以拥有较大的段大小。

当本地通信端尝试发送一个 588 字节且 DF 位置位的较大数据包时, 路由器 (10.0.0.1) 将会产生一个 PTB 消息, 指出适合下一跳链路的最大传输单元大小为 288 字节。在收到这条 PTB 消息后, TCP 在发送下一个数据包时会按照指示选择 288 字节作为响应。对于那些原本打算以 588 字节的大小发送的剩余的数据包, TCP 也会按照最大传输单元的数值将它们重新划分, 另

外发送两个大小分别为 288 字节与 116 字节的数据包。在文件传输的过程中，类似的数据包大小会不断地重复出现。

路径 MTU 发现过程是一种 TCP 明确地尝试调整段大小的方法。它适用于 TCP 连接建立后，至少是在传输大量数据时。报文段的大小能够影响吞吐量的总体性能以及 TCP 窗口大小。第 15 章将会继续讨论它是如何影响总体性能的。

## 13.5 TCP 状态转换

我们已经介绍了许多关于一个 TCP 连接启动与终止的规则，也看到了在一个连接的不同阶段需要发送的各种类型的报文段。这些决定 TOP 应该做什么的规则其实是由 TCP 所属的状态决定的。当前的状态会在各种触发条件下发生改变，例如传输或接收到的报文段、计时器超时、应用程序的读写操作，以及来自其他层的信息。这些规则可以概括为 TCP 的状态转换图。

### 13.5.1 TCP 状态转换图

图 13-8 展示了 TCP 的状态转换图。图中的状态用椭圆表示，而状态之间的转换则用箭头表示。TCP 连接的每一端都可以在这些状态中进行转换。有些转换是由于接收到某个控制位字段置位的报文段而引发的（例如，SYN,ACK,FIN）；而有些转换又会要求发送一些控制位字段置位的报文段。另外还有一些转换是由应用程序的动作或计时器超时引发的。上述各种情况都以文本注释的方式标记在转换图的相关箭头旁边。当初初始化时，TCP 从 CLOSED 状态启动。通常根据是执行主动打开操作还是被动打开操作，TCP 将分别快速转换到 SYN\_SENT 或 LISTEN 状态。

TCP 状态转换图（也称作有限状态机）。箭头表示因报文段传输、接收以及计时器超时而引发的状态转换。粗箭头表示典型的客户端的行为，虚线箭头表示典型的服务器行为。粗体指令（例如 open、close）是应用程序执行的操作

图 13-8 中值得注意的是只有一部分状态转移被认为是“典型的”。我们已经将普通的客户端状态转移用深黑的实线箭头表示，而普通的服务器状态转换用虚线箭头表示。导向 ESTABLISHED 状态的两种转换与打开一个连接相关，从 ESTABLISHED 状态导出的两种转换则用于终止一个连接。ESTABLISHED 是通信双方双向传输数据的状态。第 14 17 章将详细地介绍该状态。

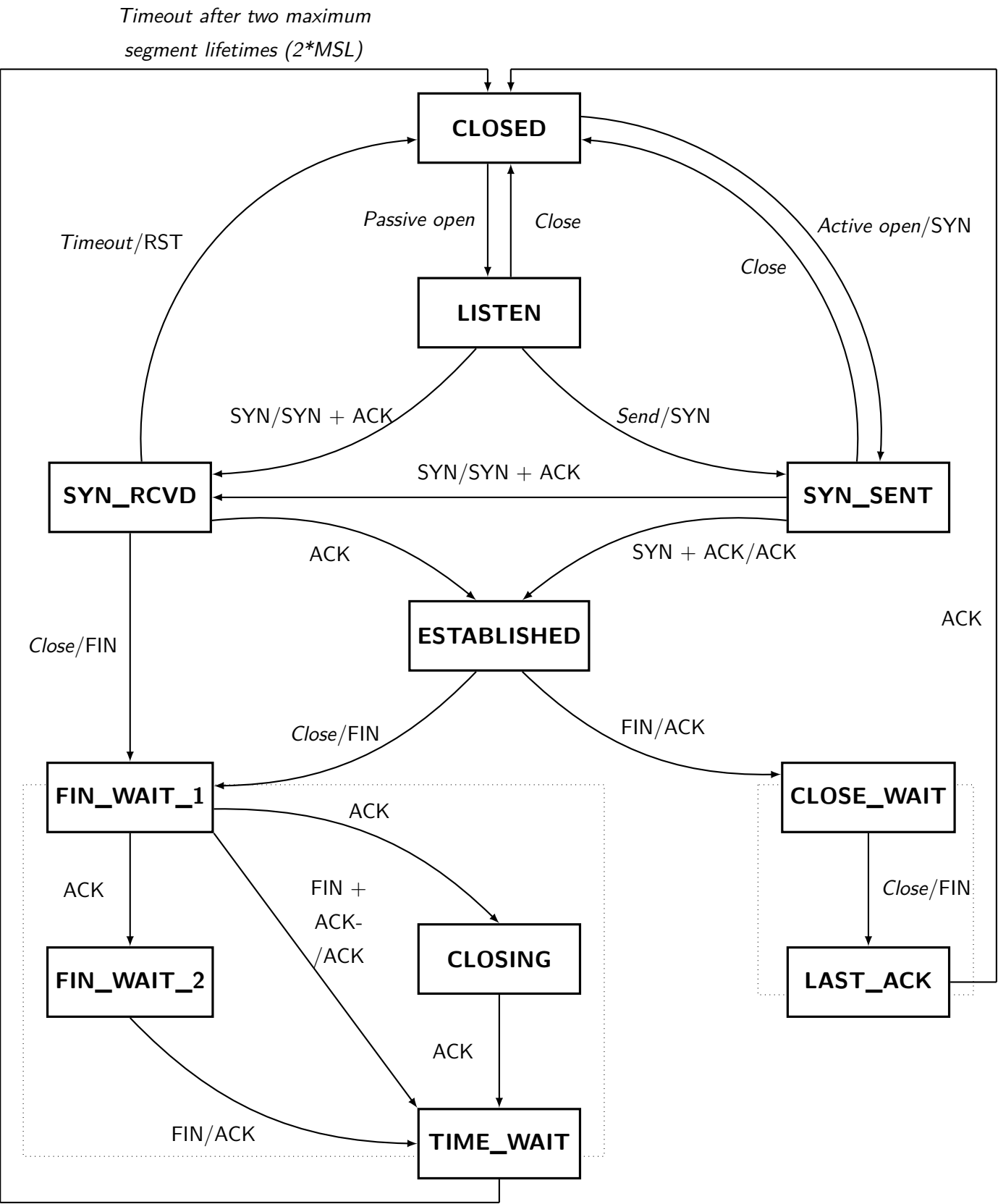


图 13-8 将 FIN\_WAIT\_1、FIN\_WAIT\_2 以及 TIME\_WAIT 状态用一个方框括起来（至少是

部分被括起来), 称作“主动关闭”。它们表示当本地应用程序发起一个关闭请求时会进入的状态集合。另外两个状态 (CLOSE\_WAIT 与 LAST\_ACK) 被一个虚线框括起来, 并标记为“被动关闭”。这些状态与等待一个节点确认一个 FIN 报文段并进行关闭相关。同时关闭可以视为一种包含两个主动关闭的形式。此外, 还使用了 CLOSING 状态。

图 13-8 中的 11 种状态名称 (CLOSED、LISTEN、SYN\_SENT 等) 都是基于 UNIX、Linux、Windows 系统中 netstat 命令所输出的名称。而这些名称则是参考了 [RFC0793] 中的名称。CLOSED 状态并不能算作一个“官方”的状态, 但在图 13-8 中却被作为一个开始状态点和一个终止状态点。

从 LISTEN 到 SYN\_SENT 的状态转换在 TCP 协议中是合法的, 但却不被伯克利套接字所支持, 因此比较少见。从 SYN\_RCVD 返回到 LISTEN 的状态转换只有在 SYN\_RCVD 状态是由 LISTEN 状态 (在正常的场景中) 而非 SYN\_SENT 状态转换而来的情况下才是正确的。这意味着, 如果我们执行一个被动打开操作 (进入 LISTEN 状态), 接收一个 SYN, 发送一个带有 ACK 确认的 SYN (进入 SYN\_RCVD 状态), 然后收到一个重置消息而非 ACK, 端点就会返回到 LISTEN 状态, 并且等待另一个连接请求的到来。

图 13-9 不仅显示了正常 TCP 连接的建立与终止过程, 还详细介绍了客户端与服务器经历的各种状态。它是图 13-1 的简略版本, 在显示相关状态的同时省略了选项与初始序列号等细节。假设在图 13-9 中左侧的客户端执行一个主动打开操作, 而右侧的服务器执行一个被动打开操作。虽然根据前文的介绍由客户端负责执行主动关闭操作, 但是通信的任何一方都能够进行这项工作。

### 13.5.2 TIME\_WAIT 状态

TIME\_WAIT 状态也称为 2MSL 等待状态。在该状态中, TCP 将会等待两倍于最大段生存期 (Maximum Segment Lifetime, MSL) 的时间, 有时也被称作加倍等待。每个实现都必须为最大段生存期选择一个数值。它代表任何报文段在被丢弃前在网络中被允许存在的最长时间。我们知道这个时限是有限制的, 因为 TCP 报文段是以 IP 数据报的形式传输的, IP 数据报拥有 TTL 字段和跳数限制字段。这两个字段限制了 IP 数据报的有效生存时间 (参见第 5 章)。[RFC0793] 将最大段生存期设为 2 分钟。然而在常见实现中, 最大段生存期的数值可以为 30 秒、1 分钟或者 2 分钟。在绝大多数情况下, 这一数值是可以修改的。在 Linux 系统中, net.ipv4.tcp\_fin\_timeout 的数值记录了 2MSL 状态需要等待的超时时间 (以秒为单位)。在 Windows 系统, 下面的注册键值也保存了超时时间:

```
1 HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\TcpTimedWaitDelay
```

该键值的取值范围是 30 300 秒。对于 IPv6 而言, 只需要将键值中的 Topip 替换为 Topip6 即可。

假设已设定 MSL 的数值, 按照规则: 当 TCP 执行一个主动关闭并发送最终的 ACK 时, 连接必须处于 TIME\_WAIT 状态并持续两倍于最大生存期的时间。这样就能够让 TCP 重新发送最终



的 ACK 以避免出现丢失的情况。重新发送最终的 ACK 并不是因为 TCP 重传了 ACK（它们并不消耗序列号，也不会被 TCP 重传），而是因为通信另一方重传了它的 FIN（它消耗一个序列号）。事实上，TCP 总是重传 FIN，直到它收到一个最终的 ACK。

另一个影响 2MSL 等待状态的因素是当 TCP 处于等待状态时，通信双方将该连接（客户端地址、客户端端口号、服务器 I 地址、服务器端口号）定义为不可重新使用。只有当 2MSL 等待结束时，或一条新连接使用的初始序列号超过了连接之前的实例所使用的最高序列号时 [RFC1122]，或者允许使用时间戳选项来区分之前连接实例的报文段以避免混淆时 [RFC619]，这条连接才能被再次使用。不幸的是，一些实现施加了更加严格的约束。在这些系统中，如果一个端口号被处于 2MSL 等待状态的任何通信端所用，那么该端口号将不能被再次使用。清单 13-3 与清单 13-4 展示了这一约束的相关例子。

许多实现与 API 都提供了绕开这一约束的方法。在伯克利套接字 API 中，SO\_RBUSEADDR 套接字选项就支持绕开操作。它允许调用者为自己分配本地端口号，即使这个端口号是某个处于 2MSL 等待状态连接的一部分。我们还会发现，即使套接字（地址、端口号对）具有绕开机制，TCP 的规则仍会防止该端口号被处于 2MSL 等待状态的同一连接的其他实例重新使用。当一个连接处于 2MSL 等待状态时，任何延迟到达的报文段都将被丢弃。因为一条连接是通过地址和端口号的 4 元组定义的。如果该连接处于 2MSL 等待状态，那么它在这段时间内将不能被重新使用。当这条正确的连接最终被建立起来后，这条连接之前的实例所传输的延迟报文段是不能被当作新连接的一部分来解读的。

对于交互式应用程序而言，客户端通常执行主动关闭操作并进入 TIME\_WAIT 状态，服务器通常执行被动关闭操作并且不会直接进入 TIME\_WAIT 状态。其中的含义是，如果我们终止一个客户端后立即重新启动同一客户端，那么新的客户端也不能重新使用相同的本地端口号。通常来说，这并不成问题。因为客户端通常使用的是由操作系统分配的临时端口号，而且它们也并不关心被分配的端口号究竟是什么（回忆一下，实际上出于安全考虑有一种推荐的随机方法 [RFC6056]）。值得注意的是，由于一个客户端能够快速产生大量的连接（尤其是针对同一个服务器），因此它不得不在临时端口号供应紧张时延迟一会儿来等待其他连接的终止。

对于服务器而言，情况则大不相同。它们通常使用一些知名的端口。如果我们终止一个已经建立了一条连接的服务器进程，然后立即尝试重新启动它，服务器不能为该程序的通信端分配对应的端口号（它将会收到一个“地址已占用”的绑定错误）。这是因为当连接进入 2MSL 等待状态时，端口号仍然是连接的一部分。根据系统对 MSL 数值的不同设定，服务器可能需要花费 1 4 分钟才能重新启动。我们可以利用 sock 程序观察这一场景。在清单 13-3 中，我们启动服务器，从客户端连接该服务器，然后终止服务器。

如果一个 TCP 连接要被其他进程重新使用，它必须在 TIME\_WAIT 状态下完成 2MSL 的延迟

当重新启动服务器时，程序会输出一条错误信息，显示由于地址已经被占用而导致它的端口

号不能被绑定。实际上这也意味着该地址与端口号的组合已经被使用。这是由于前一个连接处于 2MSL 的等待状态而造成的。根据前文所述, 这是对于端口号的重复使用最严厉的限制。netstat 命令输出的结果显示连接处于 TIME\_WAIT 状态。虽然客户端不像服务器一样会遇到如此多的关于 2MSL 等待状态的问题, 但是依然能够通过让客户端指定自己的端口号来证明相同的问题是存在的, 如清单 13-4 所示。

当端口号被其他处于 2MSL 等待状态的连接使用时, 它不能被客户端重复使用

在第一次执行客户端时, 我们利用 -v 选项查出分配给客户端的 (临时的) 本地端口号为 2091。在第二次执行客户端时, 我们利用 -b 选项告诉客户端将 2091 作为自己的本地端口号来替代操作系统分配的临时端口号。正如我们所期望的, 客户端不能完成上述操作。因为端口 2091 是一个处于 2MSL 等待状态的连接的不可分割部分。一旦等待结束 (在 Linux 机器上为 1 分钟), 客户端会尝试再次连接, 但是服务器会在连接被首次中断后退出, 所以该次尝试会被拒绝。本书将在 13.6 节继续介绍如何利用 TCP 的重置报文段表明连接被拒绝的条件。

前文曾经提到过, 大多数系统会提供一种优先级高于默认行为的方式, 这样即使某些端口是处于 2MSL 等待状态的连接的一部分, 仍然允许进程对其进行绑定。现在我们在与之前相同的场景下, 利用 sock 的 -A 选项来实现上述的绕开机制:

在此例中, 我们在启动服务器时使用了 -A 选项, 从而激活了之前提到过的 SO\_RBUSEADDR 套接字选项。通过这种方法, 即使连接处于 2MSL 等待状态, 它的端口仍然能够被服务器绑定。如果我们马上使用具有相同端口号的客户端, 将会发生下面的情况:

系统再一次提示了端点 127.0.0.1.32840 正被使用, 启动客户端失败。但是, 如果我们也对客户端使用 -A 选项, 则能够强制该连接工作:

此处说明了即使重新使用处于 2MSL 等待状态的同一个连接 (包括 4 元组), 利用 -A 选项也能够强制该连接执行。当然, 这些都发生在同一台计算机上, 操作系统能够查明那些处于 2MSL 等待状态的 (至少是潜在的) 进程对应着连接的哪一端, 并且使它们彼此间相互独立。如果在另一台主机上尝试相同的操作来建立一个连接会出现什么情况呢? 下面将根据这一想法进行测试:

除了客户端与服务器在不同的主机上之外, 上述例子与之前的类似。如果不考虑客户端的 -A 选项, 2MSL 等待时间是存在的。此处, 2MSL 持续了 30s 的等待时间。此后, 客户端才会尝试联系服务器, 然而此时服务器已经退出。

如果将客户端与服务器的计算机对换将会出现一些有趣的情况。现在, 我们将 Windows 系统作为服务器而 Linux 系统作客户端, 然后让它们重复上面的实验:

此时, 我们希望本地端口 32843 是不可用的, 但是由于在 Linux 上运行了 -A 选项, 所以可以使用这个端口。这一点违背了最初的 TCP 规范, 但如前文所述, 它是被 [RFC1122] 与 [RFC6191] 所允许的。如果有充足的理由相信新连接的报文段不会因为序列号、时间戳等问题与之前连接实例的报文段相混淆, 那么在当前连接尚处在 TIME\_WAIT 状态时上述说明会允许一条新连接的到达。[RFC1337] 与 [RFC1323] 的附录部分列出了与上述这条规则相关的常见错误。

### 13.5.3 静默时间的概念

在本地与外地的 IP 地址、端口号都相同的情况下，2MSL 状态能够防止新的连接将前一个连接的延迟报文段解释成自身数据的状况。然而，上述方法只有在与处于 2MSL 等待状态的连接相关的主机未关闭的条件下才具有意义。

如果一台与处于 TIME\_WAIT 状态下的连接相关联的主机崩溃，然后在 MSL 内重新启动，并且使用与主机崩溃之前处于 TIME\_WAIT 状态的连接相同的 IP 地址与端口号，那么将会怎样处理呢？在上述情况下，该连接在主机崩溃之前产生的延迟报文段会被认为属于主机重启后创建的新连接。这种处理方式将不会考虑在主机重启之后新连接是如何选择初始序列号的。

为了防止上述情况的发生，[RFC0793] 指出在崩溃或者重启后 TCP 协议应当在创建新的连接之前等待相当于一个 MSL 的时间。该段时间被称作静默时间。然而只有极少数实现遵循了这一点。因为绝大多数的主机在崩溃之后都需要超过一个 MSL 的时间才能重新启动。此外，如果上层应用程序自身已采用了校验和或者加密手段，那么此类错误会很容易检测出来。

### 13.5.4 FIN\_WAIT\_2 状态

在 FIN\_WAIT\_2 状态，某 TCP 通信端已发送一个 FIN 并已得到另一端的确认。除非出现半关闭的情况，否则该 TCP 端将会等待另一端的应用程序识别出自己已接收到一个文件末尾的通知并关闭这一端引起发送 FIN 的连接。只有当应用程序完成了这一关闭操作（它的 FIN 已经被接收），正在关闭的 TCP 连接才会从 FIN\_WAIT\_2 状态转移至 TIME\_WAIT 状态。这意味着连接的一端能够依然永远保持这种状态。另一端也会依然处于 CLOSE\_WAIT 状态，并且能永远维持这一状态直到应用程序决定宣布它的关闭。

目前有许多方法都能够防止连接进入 FIN\_WAIT\_2 这一无限等待状态：如果负责主动关闭的应用程序执行的是一个完全关闭操作，而不是用一个半关闭来指明它还期望接收数据，那么就会设置一个计时器。如果当计时器超时时连接是空闲的，那么 TCP 连接就会转移到 CLOSED 状态。在 Linux 系统中，能够通过调整变量 `net.ipv4.tcp_fin_timeout` 的数值来设置计时器的秒数。它的默认值是 60s。

### 13.5.5 同时打开与关闭的转换

前文已经分别介绍了在发送与接收 SYN 报文段时 SYN\_SENT 状态与 SYN\_RCVD 状态的用途。如图 13-3 所示，TCP 协议经过专门的设计后能够处理同时打开的情况，并且只建立一条连接。当同时打开的情况发生时，TCP 的状态迁移过程不同于图 13-9 的例子。通信的两端几乎在相同的时刻都会发送一个 SYN 报文段，然后它们进入 SYN\_SENT 状态。当它们接收到对方发来的 SYN 报文段时会将状态迁移至 SYN\_RCVD，然后重新发送一个新的 SYN 并确认之前接收到的 SYN。当通信两端都接收到了 SYN 与 ACK，它们的状态将都会迁移至 ESTABLISHED。

图 13-4 描述了 TCP 同时关闭的情况。当应用程序发布关闭连接的消息后, 通信两端的状况都会从 ESTABLISHED 迁移至 FIN\_WAIT\_1。与此同时, 它们都会向对方发送一个 FIN。在接收到对方发来的 FIN 后, 本地通信端的状态将从 FIN\_WAIT\_1 迁移至 CLOSING。然后, 通信双方还会发送最终的 ACK。当接收到最终的 ACK 后, 每个通信端会将状态更改为 TIME\_WAIT、从而初始化 2MSL 等待过程。

## 13.6 重置报文段

第 12 章介绍了 TCP 头部的 RST 位字段。一个将该字段置位的报文段被称作“重置报文段”或简称为“重置”。一般来说, 当发现一个到达的报文段对于相关连接而言是不正确的时, TCP 就会发送一个重置报文段。(此处, 相关连接是指由重置报文段的 TCP 与 IP 头部的 4 元组所指定的连接)。重置报文段通常会导致 TCP 连接的快速拆卸。本文将构建一些场景来证明重置报文段的用途。

### 13.6.1 针对不存在端口的连接请求

通常情况下, 当一个连接请求到达本地却没有相关进程在目的端口侦听时就会产生一个重置报文段。之前在“连接被拒绝”的错误消息中已经介绍过这种情况。这些均与 TCP 协议相关。根据第 10 章的内容, UDP 协议规定, 当一个数据报到达一个不能使用的目的端口时就会生成一个 ICMP 目的地不可达 (端口不可达) 的消息。TCP 协议则使用重置报文段来代替完成相关工作。

这样的例子不胜枚举。例如, 我们可以使用 `telnet` 客户端并指定一个在目的主机上尚未使用的端口号。这台目的主机也可以是本地的计算机:

`telnet` 客户端会快速地输出上述错误消息。清单 13-5 显示了与此命令相关的数据包交换过程。

清单 13-5 中需要查看的数值包括重置 (第 2 个) 报文段中的序列号字段与 ACK 号字段。由于到达的 SYN 报文段未打开 ACK 位字段, 重置报文段的序列号字段被设置为 0, 而 ACK 号的大小则等于接收到的初始序列号加上该报文段中数据的字节数。虽然到达的报文段中并不含有任何数据, SYN 位在逻辑上仍会占用一个字节的序列号空间。因此, 在这个例子中重置报文段中的 ACK 号等于初始序列号加上数据长度 0, 再加上 SYN 位的 1 字节。

对于一个被 TCP 端接收的重置报文段而言, 它的 ACK 位字段必须被置位, 并且 ACK 号字段的数值必须在正确窗口的范围内 (参见第 12 章)。这样有助于防止一种简单的攻击。在这种攻击中任何人都能够生成一个与相应连接 (4 元组) 匹配的重置报文段, 从而扰乱这个连接 [RFC5961]。

### 13.6.2 终止一条连接

从图 13-1 可以看出终止一条连接的正常方法是由通信一方发送一个 FIN。这种方法有时也被称为有序释效。因为 FIN 是在之前所有排队数据都已发送后才被发送出去，通常不会出现丢失数据的情况。然而在任何时刻，我们都可以通过发送一个重置报文段替代 FIN 来终止一条连接。这种方式有时被称作终止释放。

终止一条连接可以为应用程序提供两大将性：(1) 任何排队的的数据都将被抛弃，一个重置报文段会被立即发送出去；(2) 重置报文段的接收方会说明通信另一端采用了终止的方式而不是一次正常关闭。API 必须提供一种实现上述终止行为的力式来取代正常的关闭操作。

套接字 API 通过将“逗留于关闭”套接字选项 (SO\_LINGER) 的数位设置为 0 来实现上述功能。从本质上说，这意味着“不会在终止之前为了确定数据是否到达另一端而逗留任何时间”。下面的例子尽示了当一个产生大量输出的远程命令被用户取消时所发生的状况：

此处，用户决定终止这条命令的输出行为。由于单词文件中包含了 45 427 个字，这个命令很可能是某种错误。当用户键入中断字符时，系统显示对应进程（此处为 ssh 程序）已经被 2 号信号终止。该信号被称作 SIGINT，常用于终止一些交付的程序。清单 13-6 显示了上述例子中 tcpdump 的相应输出结果（由于与本文的讨论无关，大量的中间数据包已被删除）。

报文段 13 显示了正常连接的建立过程。当中断字符被键入之后，连接被终止。重置报文段中包含了一个序列号写一个确认号。还需要注意的是重置报文段不会令通信另一端做出任何响应——它不会被确认。接收重置报文段的一端会终止连接并通知应用程序当前连接已被重置。这样通常会造成“连接被另一端重置”的错误提示或类似的消息。

### 13.6.3 半开连接

如果在未告知另一端的情况下通信的一端关闭或终止连接，那么就认为该条 TCP 连接处于半开状态。这种情况发生在通信一方的主机崩溃的情况下。只要不尝试通过半开连接传输数据，正常工作的一端将不会检测出另一端已经崩溃。

产生半开连接的另一个共同原因是某一台主机的电源被切断而不是被正常关机。这种情况可能发生于下面的例子中：某些个人电脑运行了远程登录客户端，并且在一天结束时关闭。如果在电源被切断时没有数据在传输，那么服务器永远也不会知道该客户端已经消失（它可能还一直会认为该连接处于 ESTABLISHED 状态）。当第二天早晨用户回来，启动电脑并开始一个新的会话时，服务器会启动一个新的服务进程。这样会导致服务器上有很多半开的 TCP 连接（第 17 章将会介绍一种方法，使 TCP 连接的一端能够利用 TCP 的 keepalive 选项发现另一端已经消失）。

我们能够很容易地创建一个半开连接。在这种情况下，将在客户端而不是服务器做一些尝试。我们在主机 10.0.0.1 上执行 telnet 客户端程序，连接 Sun 公司提供远程过程调用服务 (sunrpc, 端口号 111) 的服务器。如清单 13-7 所示，该服务器的 IP 为 10.0.0.7。我们键入一行输

人, 并利用 tcpdump 观察它们的交互过程, 然后断开服务器主机的以太网连接并重新启动这台主机。这样就能模拟服务器崩溃的情况。(我们在重启服务器之前断开以太网连接是为了防止其通过已开启的连接发送一个 FIN。一些 TCP 会在其关闭时完成上述操作。) 在服务器重启之后, 我们重新连接以太网并且尝试从客户端向服务器发送另一行命令。在重启之后, 服务器的 TCP 会丢失之前所有连接的记忆, 因此它对数据段中指出的这条连接一无所知。此时, TCP 规定接收者将回复一个重置报文段作为响应。

服务器主机被切断连接后重启, 留给客户端一个半开的连接。当再次从这条连接上接收到其他数据时, 服务器对其一无所知。在服务器回复一个重置报文段作为响应之后, 两端之间的连接会被关闭

清单 13-8 显示了该例的 tcpdump 输出。

报文段 13 描述了正常的连接建立过程。报文段 4 将“foo”行发送至 sunrpe 服务器 (包括回车符与换行符共需要 5 个字节)。报文段 5 则完成确认工作。

此时, 我们断开服务器端 (地址 10.0.0.7) 的以太网连接, 然后重启服务器并重新接入以太网。上述操作大约花费 90 秒的时间。然后, 我们在客户端键入一行新的输入 (“bar”)。当我们按下回车键后, 这一行输入将会发送至服务器 (如清单 13-9 所示, 在 ARP 流量之后的第一个 TCP 报文段中)。由于该服务器不再记得之前的这条连接, 所以上述操作将引起服务器端的重置响应。

需要注意的是当主机重新启动时, 它将使用免费的 ARP 协议 (参见第 4 章) 来确定自己的 IPv4 地址是否已经被其他报文段使用, 而这些报文段是属于其他主机的。它还会请求与 I 地址 10.0.0.1 对应的 MAC 地址, 因为这是它对于 Internet 的默认路由。

### 13.6.4 时间等待错误

如前文所述, 设计 TIME\_WAIT 状态的目的是允许任何受制于一条关闭连接的数据报被丢弃。在这段时期, 等待的 TCP 通常不需要做任何操作, 它只需要维持当前状态直到 2MSL 的计时结束。然而, 如果它在这段时期内接收到来自于这条连接的一些报文段, 或是更加特殊的重置报文段, 它将会被破坏。这种情况被称作时间等待错误 (TIME-WAIT AsSassination TWA, 参见 [RFC1337])。图 13-10 展示了数据包的交换过程。

一个重置报文段能“破坏”TIME\_WAIT 状态并强制连接提前关闭。目前有很多方法来阻止这一问题, 其中包括在处于 TIME\_WAIT 状态时忽略重置报文段

在图 13-10 的例子中, 服务器完成了其在连接中的角色所承担的工作并清除了所有状态。客户端依然保持 TIME\_WAIT 状态。当完成 FIN 交换, 客户端的下一个序列号为 K, 而服务器的下一个序列号为 L。最近到达的报文段是由服务器发送至客户端, 它使用的序列号为 L-100, 包含的 ACK 号为 K-200。当客户端接收到这个报文段时, 它认为序列号与 ACK 号的数值都是“旧”的。当接收到旧报文段时, TCP 会发送一个 ACK 作为响应, 其中包含了最新的序列号与 ACK

号（分别是 K 与 L）。然而，当服务器接收到这个报文段以后，它没有关于这条连接的任何信息，因此发送一个重置报文段作为响应。这并不是服务器的问题，但它却会使客户端过早地从 `TIME_WAIT` 状态转移至 `CLOSED` 状态。许多系统规定当处于 `TIME_WAIT` 状态时不对重置报文段做出反应，从而避免了上述问题。

## 13.7 TCP 服务器选项

第 1 章曾介绍过，大多数 TCP 服务器是并发的。当一个新的连接请求到达服务器时，服务器接受该连接，并调用一个新的进程或线程来处理新的客户端。根据不同的操作系统，各种其他的资源也可以被分配来调用新的服务器。我们对多个并发服务器间的 TCP 交互非常感兴趣，尤其希望了解 TCP 服务器是如何使用端口号的，以及如何处理多个并发客户端的。

### 13.7.1 TCP 端口号

可通过观察任何一台 TCP 服务器来了解 TCP 是如何处理端口号的。在一台拥有 IPv4 与 IPv6 双协议栈的主机上利用 `netstat` 命令观察安全外壳服务器（也称作 `sshd`）。`sshd` 应用程序执行的是安全外壳协议 [RFC4254]。该协议能够提供可加密认证的远程终端功能。下面的输出结果来自于没有主动安全外壳连接的系统（除了与服务器相关的输出外，其他所有的输出行都已被删除）：`-a` 选项能够报告所有的网络节点，包括那些处于侦听状态和未处于侦听状态的节点。`-n` 选项以点分十进制（或十六进制）数的形式打印 IP 地址，而不会试图利用 DNS 将地址转换为一个域名。此外，该选项还会打印数值端口号（例如 22），而不是服务名（例如 `ssh`）。`-t` 选项用于只选择 TCP 节点。

本地地址（这实际意味着本地节点）的输出结果为 `:: 22`。这种面向 IPv6 的方式表示的是全 0 地址，也被称作通配符地址，并使用端口号 22。这意味着一个针对 22 号端口的连接进入请求（即一个 `SYN`）会被任何本地接口接受。如果主机是多宿主的（此例即是），我们可以为本地 IP 地址指定一个单一的地址（主机 IP 地址中的一个地址），并且只有被该接口接收到的连接才能够被接受（参见本节后面的例子）。端口号 22 是为安全外壳协议预留的知名端口号。其他端口号是由网络编号分配机构（ITNA）来维护的。

外部地址的输出结果为：`*`，这表示一个通配符地址与端口号（即，一个通配符节点）。此处由于本地节点处于 `LISTEN` 状态，正等待一个连接的到来，因此外部地址与外部端口号尚不知晓。现在我们在主机 10.0.0.3 上启动一个安全外壳客户端来连接该服务器。下面是从 `netstat` 输出的相关行（`RECV-Q` 与 `Send-Q` 列只包含零值，为清楚起见，将其删除）：

标有端口号“22”的第 2 行是一个 `ESTABLISHED` 连接。本地与外地节点相关的 4 元组都会填写在这个连接中，其中包括：本地 IP 地址与端口号，外地 IP 地址与端口号。本地 IP 地址与连接请求到达的接口相关（以太网接口通过与 IPv4 地址映射的 IPv6 地址：`ff::10.0.0.1` 进行标

识)。

处于 LISTEN 状态的本地节点会独自地运行。它用于为并发服务器接收未来可能出现的请求。当有新的连接请求到达并被接收时，操作系统中的 TCP 模块创建处于 ESTABLISHED 状态的新节点。同样需要注意的是，当连接处于 ESTABLISHED 状态时，它的端口号仍为 22，与 LISTEN 状态时相同。

现在我们从同一个系统 (10.0.0.3) 向服务器发送另一个客户端请求。相关的 netstat 输出如下：

现在我们有二个处于 ESTABLISHED 状态的连接。它们以同一个客户端到相同的服务器端。两条连接的服务器端口号均为 22。由于外地端口号不同，因此这并不算是一个 TCP 错误。这两个外地端口必须是不同的。因为每个安全外壳程序使用的是一个临时端口，而每一个临时端口在定义时都是主机 (10.0.0.3) 尚未使用的端口。

这个例子再次说明 TCP 依靠 4 元组多路分解 (demultiplex) 了获得的报文段。目的 IP 地址与目的端口号、源 IP 地址与源端口号，这 4 元组共同构成了本地与外地节点。TCP 协议不能仅仅根据目的端口号来决定哪一个进程该得到接收的报文段。在所有三个节点中只有位于端口 22 的节点会处于 LISTEN 状态并接收进入的连接请求。处于 ESTABLISHED 状态的节点不能接收 SYN 报文段，而处于 LISTEN 状态的节点则不能接收数据段。例子中主机的操作系统已经证实了这一点。(如果不能如此，TCP 就会变得非常混乱，从而不能正常地工作)。

下面我们发起第三个客户端连接。这条连接来自 IP 地址 169.229.62.97，通过 DSLPPPoE 链路与服务 10.0.0.1 相连。因此这个客户端与服务器不处于同一个以太网。(为了清楚起见，下面的输出结果移除了 Proto 列，只留下了 tcp 部分。)

在这台多宿主的主机上，第三条 ESTABLISHED 连接的 IP 地址与 PPPoE 链路的接口地址 (67.125.227.195) 相关联。值得注意的是 Send-Q 状态的数值并不是 0，而是被 928 字节所代替。这意味着服务器已经在这条连接上发送了 928 字节的数据但仍然未收到任何确认。

### 13.7.2 限制本地 IP 地址

本节将介绍当服务器不能借助通配符处理某个本地 IP 地址而将其设置为一个特殊的本地地址时发生的情况。如果我们将 sock 程序当作服务器运行并为其提供一个特殊的 IP 地址，那么该地址将成为监听端的本地 IP 地址。例如：

这样就限制了服务器只能使用到达本地 IPv4 地址 10.0.0.1 的连接。下面 netstat 的输出结果反映了这一点：

在上述例子中特别有趣的是我们的 sock 程序只与本地 IPv4 地址 10.0.0.1 绑定，所以 netstat 的输出结果与之前不同。在之前的例子中，通配符地址与端口号标识了两种版本的 IP 地址。在这种情况下，我们绑定了一个特殊的地址、端口或地址族 (只适用 IPv4)。如果我们从本地网络连接这台服务器，比如从主机 10.0.0.3，那么正常工作的记录情况如下：



如果我们从一台目的地址不是10.0.0.1（甚至包括本地地址 127.0.0.1）的主机连接服务器，连接请求将不会被 TCP 模块接收。如果我们查看 `tcpdump`，SYN 会引发一个 RST 报文段，如清单 13-9 所示。

服务器的应用程序将不会察觉到连接请求—因为拒绝接收的操作是由操作系统的 TCP 模块根据该应用程序指定的本地 IP 地址与 SYN 报文段中包含的目的地址做出的。我们发现限制本地 IP 地址的能力是相当严格的。

### 13.7.3 限制外部节点

根据第 10 章的介绍，一台 UDP 服务器不仅能够指定本地 IP 地址与本地端口号，还能够指定外部 IP 地址与外部端口号。[RFC0793] 所介绍的 TCP 的抽象接口函数允许一台服务器为一个完全指定的外部节点（等待一个特定的客户端以发起一个主动打开）或者一个未被指定的外部节点（等待任何客户端）执行被动打开。

不幸的是，普通的伯克利套接字 API 没有提供实现这一点的方法。服务器不必指定客户端的节点，而是等待连接的到来，然后检查客户端的 IP 地址与端口号。表 13-3 概括了 TCP 服务器能够建立的三种类型的地址绑定。

在上述例子中，`local_port`是服务器被分配的端口号，而`local_IP`必须是一个应用于本地系统的单播 IP 地址。表 13-3 中三行的排列顺序显示了当收到一个连接请求时 TCP 模块决定选择哪一个节点的次序。最明确的绑定（第 1 行，如果支持的话）将会被首先尝试，而最不明确的绑定（最后一行，所有的 IP 地址都用通配符表示）将会被最后尝试。对于同时支持 IPv4 与 IPv6 双协议栈的系统，它的端口号空间可能会出现混合的情况。从本质上说，这意味着如果服务器使用 IPv6 地址绑定了一个端口号，那么也就将该端口号应用于了 IPv4 地址。

### 13.7.4 进入连接队列

一个并发的服务器会为每一个客户端分配一个新的进程或线程，这样负责侦听的服务器能够始终准备着处理下一个到来的连接请求。这是使用并发服务器的根本原因。然而，在侦听服务器正创建一个新进程时，或者在操作系统忙于运行其他高优先级的进程时，甚至更糟的是在服务器正在被伪造的逐接请求（这些伪造的建立连接请求是不被允许的）攻击时，多个连接请求可能会到达。在这些情况下，TCP 应当如何处理呢？

为了充分探讨这个问题，我们首先必须认识到，在被用于应用程序之前新的连接可能会处于下述两个状态。一种是连接尚未完成但是已经接收到 SYN（也就是处于 `SYN_RCVD` 状态）。另一种是连接已经完成了三次握手并且处于 `ESTABLISHED` 状态，但还未被应用程序接受。因此在内部操作系统通常会使用两个不同的连接队列分别对应上述两种不同的情况。

应用程序通过限制这些队列的大小来进行控制。传统上，使用伯克利套接字 API 应用程序只能间接地控制这两个队列的大小总和。在现代的 Linux 内核中，这种行为已更改为第二种状

况下的连接数目 (ESTABLISHED 状态的连接)。因此, 应用程序能够限制完全形成的等待处理的连接数目。在 Linux 中, 将会适用以下规则:

1. 当一个连接请求到达(即, SYN 报文段), 将会检查系统范围的参数 `net.ipv4.tcp_max_syn_backlog` (默认为 1000)。如果处于 SYN\_RCVD 状态的连接数目超过了这一阈值, 进入的连接将会被拒绝。
2. 每一个处于侦听状态下的节点都拥有一个固定长度的连接队列。其中的连接已经被 TCP 完全接受 (即三次握手已经完成), 但未被应用程序接受。应用程序会对这一队列做出限制, 通常称为未完成连接 (backlog)。backlog 的数目必须在 0 与一个系统指定的最大值之间。该最大值称为 `net.core.somaxconn`, 默认值 128 (包含)。需要记住的是 backlog 的数值指出了侦听节点中排队连接的最大数目, 所有这些连接已经被 TCP 接受并等待应用程序接受。无论对系统所允许的已经建立连接的最大数目, 还是对一个并行服务器所能同时处理的客户端数目, backlog 都不会造成影响。
3. 如果侦听节点的队列中仍然有空间分配给新的连接, TCP 模块会应答 SYN 并完成连接。直到接收到三次握手中的第 3 个报文段之后, 与侦听节点相关的应用程序才会知道新的连接。当客户端的主动打开操作顺利完成之后, 客户端可能会认为服务器已经准备好接收数据, 然而服务器上的应用程序此时可能还未收到关于新连接的通知。如果这种情况发生, 服务器的 TCP 模块将会把到来的数据存入队列中。
4. 如果队列中已没有足够的空间分配给新的连接, TCP 将会延迟对 SYN 做出响应, 从而给应用程序一个跟上节奏的机会。Linux 在这一方面有着独特的行为——它坚持在能力允许的范围内不忽略进入的连接。如果系统控制变量 `net.ipv4.tcp_abort_on_overflow` 已被设定, 新进入的连接会被重置报文段重新置位。

在队列溢出的情况下, 发送重置报文段通常是不可取的, 而且默认情况下这项功能也不会打开。客户端会尝试与服务器联系, 如果它在交换 SYN 期间接收到一个重置报文段, 那么它可能会错误地认为没有服务器存在 (而不是认为有一台服务器存在并且十分繁忙)。太忙实际上是一种“软”的或者暂时的错误, 而不是一种硬性的错误。正常情况下, 当队列已满, 应用程序或操作系统会十分繁忙, 此时应当阻止应用程序再去服务那些进入的连接。上述状况可能会在短时间内得到改善。然而, 如果一台服务器的 TCP 使用重登报文段进行回复, 那么客户端将会放弃主动打开的操作 (这与服务器没有启动时所看到的情况是类似的)。在不发送重置报文段的情况下, 如果一台侦听的服务器始终无法抽出时间来接受那些已经被 TCP 接受却超出队列保存上限的连接, 那么根据正常的 TCP 机制, 客户端的主动打开操作将会最终超时。在 Linux 中, 连接的客户端将会明显地放缓一段时间, 它们既不会超时也不会重置。

借助我们的 sock 程序, 大家会看到当进入连接队列溢出后会发生的情况。我们利用一个新的选项 (-O) 来调用程序, 并且告诉它在创建完侦听节点后暂停, 直到接收到任何连接请求。如

果之后我们在暂停期间再调用多个客户端，服务器的接收连接队列将会被填满，因此可以借助 tcpdump 检查所发生的一切。

-q1 选项将侦听节点的 backlog 数值设置 1。-o30000 选项使程序在接收任何客户端连接之前先休眠 30000 秒（基本上是一个很长的时间，大约 8 小时）。如果我们现在不断地尝试连接这台服务器，最早的 4 个连接将会被立即完成。此后两个连接需要 9 秒才能完成。其他操作系统在处理这种情况时会有明显的不同。例如在 Solaris8 和 FreeBSD 4.7 中，两个连接会被立即处理而第 3 个连接将会超时，而随后的连接也将超时。

清单 13-10 显示了用一台 Linux 客户端连接一台 FreeBSD 服务器的 tcpdump 输出结果。FreeBSD 服务器上运行着符合上文参数设定的 sock 程序（在 TCP 连接建立时，即三次握手完成时，已经用黑体标记了客户端的端口号）。

TCP 接受的第 1 个客户端连接请求来自端口 2461（报文段 1 3）。第 2 个客户端的连接请求来自端口 2462，也被 TCP 接受（报文段 4 6）。服务器的应用程序仍处于睡眠状态，不能够接受任何连接。所有的工作都是由操作系统的 TCP 模块来完成的。由于三次握手过程均已完成，这两个客户端都从主动打开成功地返回。

我们尝试开启第 3 个客户端，它的 SYN 报文段如清单 13-10 中的第 7 个报文段所示（端口号 2463），但由于侦听节点的队列已满，服务器端的 TCP 忽略了该 SYN 报文段。客户端根据二进制指数退避机制重新传输了它的 SYN 报文段，如清单 13-10 中的第 8 12 个报文段所示。在 FreeBSD 与 Solaris 系统中，TCP 会在队列满后忽略进入的 SYN 报文段。

据前文所述，如果侦听者的队列有足够的空间，TCP 将会接受一个进入的连接请求（即一个 SYN 报文段），而不会让应用程序去识别这条连接来自何方（源 IP 地址与源端口号）。这一点并不是 TCP 协议所要求的，而是通用的实现技术（即伯克利套接字的工作方式）。如果采用替代伯克利套接字 API 的方法（例如 TLI/XTI），能够为应用程序提供一种了解何时连接到达的方法，并允许它们选择是否接受到达的连接请求。TUI 只是在理论上提供这种能力，但未能完全付诸实践，因此伯克利套接字能够更有效地为 TCP 接口提供支持。

在借助伯克利套接字实现的 TCP 中，当应用程序被告知一条连接已经到达时，TCP 的三次握手过程已经完成。上述行为也意味着一个 TCP 服务器无法让一个客户端的主动打开操作失败。当一个新的客户端连接传达至服务器应用程序时，TCP 的三次握手过程已经结束，而且客户端的主动打开操作已经成功完成。如果此后服务器查看了客户端的 IP 地址与端口号，并且决定不向该客户端提供服务，那它只能关闭（发送一个 FIN）或者重置这系连接（发送一个 RST）。无论处于上述哪一种情况，客户端在完成主动打开操作后都会认为一切正常，甚至已经向服务器发出了请求。因此，需要其他传输层协议来为应用程序提供区分连接到达与接受的功能（即 OSI 模型的传输层），但不是 TCP。

## 13.8 与 TCP 连接管理相关的攻击

SYN 泛洪是一种 TCP 拒绝服务攻击，在这种攻击中一个或多个恶意的客户端产生一系列 TCP 连接尝试（SYN 报文段），并将它们发送给一台服务器，它们通常采用“伪造”的（例如，随机选择）源 IP 地址。服务器会为每一条连接分配一定数量的连接资源。由于连接尚未完全建立，服务器为了维护大量的半打开连接会在耗尽自身内存后拒绝力后续的合法连接请求服务。

### 13.8.1 syncookie

因为区分合法的连接尝试与 SYN 泛洪并不是一件容易的事情，所以抵御上述攻击存在一定的难度。一种针对此问题的机制被称作 SYN cookies [RFC4987]。SYN cookies 的主要思想是，当一个 SYN 到达时，这条连接存储的大部分信息都会被编码并保存在 SYN +ACK 报文段的序列号字段。采用 SYN cookies 的目标主机不需要进入的连接请求分配任何存储资源—只有当 SYN +ACK 报文段本身被确认后（并且已返回初始序列号）才会分配真正的内存。在这种情况下，所有重要的连接参数都能够重新获得，同时连接也能够被设置为 ESTABLISHED 状态。

在执行 SYN cookies 过程中需要服务器仔细地选择 TCP 初始序列号。本质上，服务器必须将任何必要的状态编码并存储于 SYN+ACK 报文段的序列号字段。这样一个合法的客户端会将其值作为报文段的 ACK 号字段返回给服务器。很多方法都能够完成这项工作，下面将具体介绍 Linux 系统所采用的技术。

服务器在接收到一个 SYN 后会采用下面的方法设置初始序列号（保存于 SYN +ACK 报文段，供于客户端）的数值：首 5 位是  $t$  模 32 的结果，其中  $t$  是一个 32 位的计数器，每隔 64 秒增 1；接着 3 位是对服务器最大段大小（8 种可能之一）的编码值；剩余的 24 位保存了 4 元组与  $t$  值的散列值。该数值是根据服务器选定的散列加密算法计算得到的。

#### Note

linux 4.4 syncookie [syncookie](#) 使用散列函数 sha1

在采用 SYN cookies 方法时，服务器总是以一个 SYN +ACK 报文段作为响应（符合任何典型的 TCP 连接建立过程）。在接收到 ACK 后，如果根据其中的：值可以计算出与加密的散列值相同的结果，那么服务器才会为该 SYN 重新构建队列。这种方法至少有两个缺陷。首先，由于需要对最大段大小进行编码，这种方法禁止使用任意大小的报文段。其次，由于计数器会回绕，连接建立过程会因周期非常长（长于 64 秒）而无法正常工作。基于上述原因，这一功能并未作为默认设置。

### 13.8.2 路径最大传输单元

另一种攻击方法与路径最大传输单元发现过程相关。在这种攻击中，攻击者伪造一个 ICMP PTB 消息。该消息包含了一个非常小的 MTU 值（例如，68 字节）。这样就迫使受害的 TCP 尝试采用非常小的数据包来填充数据，从而大大降低了它的性能。最粗暴的解决方法是简单地禁用主机的路径最大传输单元发现功能。当接收到的 ICMP PTB 消息的下一跳最大传输单元小于 576 字节时，其他选项会禁用路径最大传输单元发现功能。还有一种 Linux 实现的方法，前文曾介绍过，是使最小的数据包大小（对 TCP 使用的大数据包）固定为某一数据，并使较大的数据包不将 IPv4 的 DF 位置位。这种方法虽然比完全禁用路径最大传输单元发现功能更具吸引力，但也与其十分类似。

### 13.8.3 hijacking tcp

另一种类型的攻击涉及破坏现有的 TCP 连接，甚至可能将其劫持 (hijacking)。这一类攻击通常包含的第一步是使两个之前正在通信的 TCP 节点“失去同步”。这样它们将使用不正确的序列号。它们是序列号攻击的典型例子 [RFC1948]。至少有两种方法能实现上述攻击：在连接建立过程中引发不正确的状态传输（类似于 13.6.4 小节介绍的时间等待错误），在 ESTABLISHED 状态下产生额外的数据。一旦两端不能再进行通信（但却认为它们间拥有一个打开的连接），攻击者就能够在连接中注入新的流量，而且这些注入的流量会被 TCP 认为是正确的。

### 13.8.4 欺骗攻击

有一类攻击被称作欺骗攻击。这类攻击所涉及的 TCP 报文段是由攻击者精心定制的，目的在于破坏或改变现有 TCP 连接的行为。在 [RFC4953] 中大量讨论了此类攻击及它们的防治技术。攻击者可以生成一个伪造的重置报文段并将其发送给一个 TCP 通信节点。假定与连接相关的 4 元组以及校验和都是正确的，序列号也处在正确的范围。这样就会造成连接的任意一端失败。随着互联网变得更快，为了维持性能被认为“处于窗口”的序列号范围也在不断地扩大（参见第 15 章），上述攻击也受到越来越多的关注。欺骗攻击还存在于其他类型的报文段（SYN，甚至 ACK）中（有时会与泛洪攻击结合使用），引发大量的问题。相关的防御技术包括：认证每一个报文段（例如，使用 TCP-AO 选项）；要求重置报文段拥有一个特殊的序列号以代替处于某一范围的序列号；要求时间戳选项具有特定的数值；使用其他形式的 cookie 文件，让非关键的数据依赖于更加准确的连接信息或一个秘密数值。

欺骗攻击虽然不是 TCP 协议的一部分，但是能够影响 TCP 的运行。例如，ICMP 协议能够被用于修改路径最大传输单元的发现行为。它也能够被用于指出一个端口号或一台主机已失效，从而终止一个 TCP 连接。[RFC5927] 介绍了大量的此类攻击，并且还提出了一些防御 ICMP 欺骗消息、提高鲁棒性的方法。这些建议不仅局限于验证 ICMP 消息，而且还涉及其可能包含

的 TCP 报文段。例如，包含的报文段应该拥有正确的 4 元组与序列号。

## 13.9 总结

在两个进程使用 TCP 协议交换数据之前，它们必须要在彼此间建立一条连接。当数据传输完毕，它们将终止这条连接。本章详细介绍了连接是如何借助三次握手过程建立的，而且又是如何利用 4 个报文段终止的。本章还介绍了 TCP 是如何处理同时打开与关闭操作的，以及如何管理各个选项，其中包括选择性确认、时间戳、最大段大小、TCP 认证以及用户超时选项。

本章使用 tcpdump 与 Wireshark 来显示 TCP 协议的行为以及 TCP 头部字段的使用情况。还展示了连接的建立过程是如何超时的，重置报文段是如何发送与解析的，以及 TCP 是如何提供半打开与半关闭连接的。TCP 既约束了在一个主动打开操作中尝试连接的改数，又约束了在一次被动打开操作后能服务的尝试连接次数。

TCP 状态转换图对理解其运行非常重要。本章依次介绍了连接的建立、终止以及状态迁移的各个步骤。本章还介绍了并发 TCP 服务器设计中 TCP 连接建立的相关工作。

一条 TCP 连接是由一个 4 元组唯一定义的，包括：本地 IP 地址，本地端口号，外部 IP 地址，外部端口号。每次连接终止时，通信一端必须维护这些相关信息。根据本章的介绍，TCP 的 TIME\_WAIT 状态负责完成这些工作。规则是执行主动关闭操作的一端进入 TIME\_WAIT 状态并维持两倍的最大段生存时间。这样有助于防止 TCP 处理同一条连接中旧实例的报文段。当新的连接尝试使用相同的 4 元组时，使用时间戳选项能够减少等待时间，另外它还有助于探测回绕的序列号以及更好地测量往返时间。

TCP 在资源耗尽与欺骗等攻击面前是十分脆弱的，但已研究出一些方法来抵御上述问题。此外，TCP 还会受到其他协议的影响，比如 ICMP。通过仔细地分析 ICMP 消息所返回的原始数据报可以加强对 ICMP 的防御。最后，TCP 可以与其他协议结合使用，为协议栈的其他层提供安全支持（例如，IPsec 与 TLS/SSL，参见第 18 章），这已成为标准的做法。

## 13.10 参考文献

# TCP 超时与重传

---

## 14.1 引言

到目前为止，我们并没有过多地涉及效率与性能，而主要关注操作的正确性。在本章及接下来的两章中，我们不仅讨论 TCP 执行的基本任务，还关心其执行效率。由于下层网络层 (IP) 可能出现丢失、重复或失序包的情况，TCP 协议提供可靠数据传输服务。为保证数据传输的正确性，TCP 重传其认为已丢失的包。TCP 根据按收端返回至发送端的一系列确认信息来判断是否出现丢包。当数据段或确认信息丢失，TCP 启动重传操作，重传尚未确认的数据。TCP 拥有两套独立机制来完成重传，一是**基于时间**，二是**基于确认信息的构成**。第二种方法通常比第一种更高效。

TCP 在发送数据时会设置一个计时器，若至计时器超时仍未收到数据确认信息，则会引发相应的超时或基于计时器的重传操作，计时器超时称为重传超时 (RTO)。另一种方式的重传称为快速重传，通常发生在没有延时的情况下。若 TCP 累积确认无法返回新的 ACK，或者当 ACK 包含的选择确认信息 (SACK) 表明出现失序报文段时，快速重传会推断出现丢包。通常来说，当发送端认为接收端可能出现数据丢失时，需要决定发送新 (未发送过的) 数据还是重传。本章内容将详细讨论 TCP 怎样判断出现报文段丢失及其响应操作。发送数据量问题，即由丢包而引发的拥塞控制机制，将会在第 16 章具体介绍。这里，我们探讨如何根据某个连接的 RTT 来设置 RTO，基于计时器的重传机制，以及 TCP 快速重传操作。另外我们也会看到 SACK 怎样帮助确定丢失数据、失序和重复 IP 包对 TCP 行为的影响，以及 TCP 重传时改变包大小的方法。最后我们简要讨论一些可能导致 TCP 出现过分积极或被动行为的攻击方法。

## 14.2 简单的超时与重传举例

我们已经看到一些超时和重传的例子。

1. 在第 8 章 ICMP 目的不可达（端口不可达）的例子中，采用 UDP 的 TFTP 客户端使用简单（且低效）的超时和重传策略：设置足够大的超时间隔，每 5 秒进行一次重传。
2. 第 13 章的尝试与不存在的主机建立连接中，我们看到 TCP 在尝试建立连接的过程中，在每次重传时采用比上次更大的延时间隔。
3. 在第 3 章的以太网冲突中，我们也可以看到相关操作。

上述机制都是由计时器超时引发的。

我们首先来看 TCP 的基于计时器的重传策略。先建立一个连接，并发送一些数据验证连接正常。然后断开连接的一端，这时再发送一些数据，观察 TCP 的操作。这里我们采用 Wireshark 来跟踪记录连接状况（见图 14-1）。

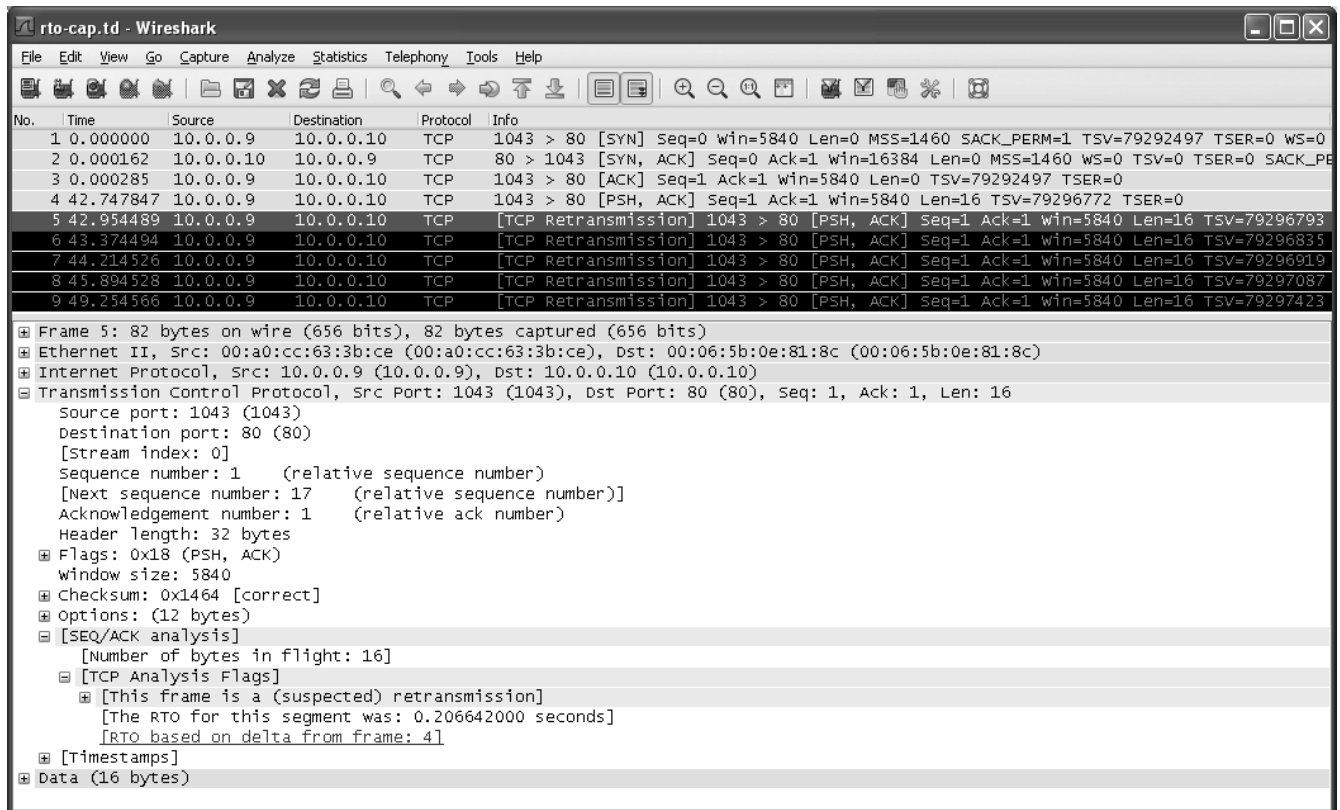


图 14.1: tcp 重传简单的例子



报文段 1、2、3 为 TCP 建立连接的握手过程。连接建立完成后，Web 服务器处于等待 Web 请求的状态。在发出请求前，我们先断开服务器端主机的连接。在客户端输入如下命令：

```
Linux% telnet 10.0.0.10 80
Trying 10.0.0.10...
Connected to 10.0.0.10.
Escape character is '^]'.
GET / HTTP /1.0
Connection closed by foreign host.
```

该请求无法传输至服务器端，因此它会在客户端的 TCP 队列中存储一段时间。此时采用 netstat 命令读取客户端队列状态显示为非空：

Active Internet connections (w/o servers)

| Proto | Recv-Q | Send-Q | Local Address | Foreign Address | State       |
|-------|--------|--------|---------------|-----------------|-------------|
| tcp   | 0      | 18     | 10.0.0.9:1043 | 10.0.0.10:www   | ESTABLISHED |

可以看到发送队列中有 18 字节的数据，等待被传送至 Web 服务器。这 18 个字节包含了之前显示的请求命令以及回车换行。其他的输出细节（包括地址和状态信息）会在下面涉及。

报文段 4 为客户端首次尝试发送 Web 请求，时间为 42.748s。0.206s 以后，在 42.9545 发送了第二次请求。接着在 43.374s，即 0.420s 后，再次尝试请求。后续的请求（重传）时刻分别为 44.215、45.895 以及 49.255s，间隔分别为 1.680 和 3.360s。

每次重传间隔时间加倍称为**二进制指数退避**（binary exponential backoff），我们在第 13 章的 TCP 尝试建立连接失败时提到过，后面将会详细讨论。自首次请求至连接完全失效，总时间为 15.5min。随后，客户端会显示如下错误信息：

```
Connection closed by foreign host.
```

逻辑上讲，TCP 拥有两个阈值来决定如何重传同一个报文段。主机需求 RFC [RFC1122] 描述了这两个阈值，在第 13 章中也提到过。R1 表示 TCP 在向 IP 层传递“消极建议”（如重新评估当前的 I 路径）前，愿意尝试重传的次数（或等待时间）。R2（大于 R1）指示 TCP 应放弃当前连接的时机。R1 和 R2 应分别至少设为三次重传和 100 秒。对连接的建立过程（发送 SYN 报文段），阈值设置与数据段传输有所区别，针对 SYN 报文段的 R2 应最少设为 3 分钟。

Linux 系统中，对一般数据段来说，R1 和 R2 的值可以通过应用程序，或使用系统配置变量 net.ipv4.tcp\_retries1 和 net.ipv4.tcp\_retries2 设置。变量值为重传次数，而不是以时间为单位。tcp\_retries2 默认值为 15，对应约为 13 – 30 分钟，根据具体连接的 RTO 而定。net.ipv4.tcp\_retries1 默认值为 3。对于 SYN 报文段，变量 net.ipv4.tcp\_syn\_retries 和 net.ipv4.tcp\_synack\_retries 限定重传次数，默认值为 5（约 180 秒）。Windows 也有相应控制 TCP 行为的变量，包括 R1 和 R2。通过下列注册表项可以修改对应值 [WINREG]：

```
HKLM\System\CurrentControlSet\Services\Tcpip\Parameters  
HKLM\System\CurrentControlSet\Services\Tcpip6\Parameters
```

最主要的值为 `TcpMaxDataRetransmissions`，对应 Linux 中的 `tcp_retries2` 变量，其默认值为 5。至此我们看到，TCP 需要为重传计时器设置超时值，指示发送数据后等待 ACK 的时间。假设 TCP 只工作在静态环境中，那么很容易为超时设置一个合适的值。由于 TCP 需要适应不同环境进行操作，可能随着时间不断变化，因此需基于当前状态设定超时值。例如，若某个网络连接失败，需重新建立，RTT 也会随之改变（可能变化很大）。也就是说，TCP 需要动态设置 RTO（重传超时）。下面我们讨论这一问题。

## 14.3 设置重传超时

TCP 超时和重传的基础是怎样根据给定连接的 RTT 设置 RTO。若 TCP 先于 RTT 开始重传，可能会在网络中引入不必要的重复数据。反之，若延迟至远大于 RTT 的间隔发送重传数据，整体网络利用率（及单个连接吞吐量）会随之下降。由于 RTT 的测量较为复杂，根据路由与网络资源的不同，它会随时间而改变。TCP 必须跟踪这些变化并适时做出调整来维持好的性能。

TCP 在收到数据后会返回确认信息，因此可在该信息中携带一个字节的的数据（采用一个特殊序列号）来测量传输该确认信息所需的时间。每个此类的测量结果称为 RTT 样本（RTT sample）。TCP 首先需要根据一段时间内的样本值建立好的估计值。第二步是怎样基于估计值设置 RTO。RTO 设置得当是保证 TCP 性能的关键。

每个 TCP 连接的 RTT 均独立估算，并且重传计时器会对任何占用序列号的在传数据（包括 SYN 和 FIN 报文段）计时。如何恰当设置计时器一直以来都是研究的热点问题，近年来也取得了一些成果。本章节将探讨计算 RTO 计算方法在演进历程中的一些重要里程碑。首先我们介绍第一个（“经典”）方法，详见 [RFC0793]。

### 14.3.1 经典方法

最初的 TCP 规范 [RFC0793] 采用如下公式计算得到平滑的 RTT 估计值（称为 SRTT）：

$$SRTT = \alpha(SRTT) + (1 - \alpha)RTT, \quad (14.1)$$

这里，SRTT 是基于现存值和新的样本值 RTT 得到更新结果的。常量  $\alpha$  为平滑因子，推荐值 0.8 0.9。每当得到新的样本值，SRTT 就会做出相应的更新。从  $\alpha$  的设定值可以看到，新的估计值有 80% 90% 来自现存值，10% 20% 来自新测量值。这种估算方法称为指数加权移动平均（Exponentially Weighted Moving Average, EWMA）或低通过滤器（low-pass filter）。该方法实现起来较为简单，只要保存 SRTT 的先前值即可得到新的估计值。

考虑到 SRTT 估计器得到的估计值会随 RTT 而变化, [RFC0793] 推荐根据如下公式设置 RTO:

$$RTO = \min(ubound, \max(lbound, (SRTT)B)) \quad (14.2)$$

这里的 B 为时延离散因子, 推荐值为 1.3 2.0。ubound 为 RTO 的上边界 (可设定建议值, 如 1 分钟), lbound 为 RTO 的下边界 (可设定建议值, 如 1 秒)。我们称该方法为经典方法, 它使得 RTO 的值设置为 1 秒, 或约两倍的 SRTT。对于相对稳定的 RTT 分布来说, 这种方法能取得不错的性能。然而, 若 TCP 运行于 RTT 变化较大的网络中, 则无法获得期望的效果。

### 14.3.2 标准方法

在 [88] 中, Jacobson 进一步分析了上述经典方法, 即按照 [RFC0793] 设置计时器无法适应 RTT 的大规模变动 (特别是, 当实际的 RTT 远大于估计值时, 会导致不必要的重传)。增大的 RTT 样本值表明网络已出现过载, 此时不必要的重传无疑会进一步加重网络负担。

为解决上述问题, 可对原方法做出改进以适应 RTT 变动较大的情况。可通过记录 RTT 测量值的变化情况以及均值来得到较为准确的估计值。基于均值和估计值的变化来设置 RTO, 将比仅使用均值的常数倍来计算 RTO 更能适应 RTT 变化幅度较大的情况。

[88] 中的图 5 和图 6 显示了采用 [RFC0793] 与同时考虑 RTT 变化值的方法计算 RTO 的对比情况。如果我们将 TCP 得到的 RTT 测量样本值考虑为一个统计过程, 那么同时测量均值和方差 (或标准差) 能更好地估计将来值。对 RTT 的可能值范围做出好的预测可以帮助 TCP 设定一个能适应大多数情况的 RTO 值。

正如 Jacobson 所述, 平均偏差 (mean deviation) 是对标准差的一种好的逼近, 但计算起来却更容易、更快捷。计算标准差需要对方差进行平方根运算, 对于快速 TCP 实现来说代价较大。(但这并非全部原因, 可参见 [G04] 中所述的有趣的“争论”历史。) 因此我们需要结合平均值和平均偏差来进行估算。可对每个 RTT 测量值 M (前面称为 RTT.) 采用如下算式:

$$srtt \leftarrow (1 - g)(srtt) + (g)M \quad (14.3)$$

$$rttvar \leftarrow (1 - h)(rttvar) + (h)(|M - srtt|) \quad (14.4)$$

$$Rto = srtt + 4(rttvar) \quad (14.5)$$

这里, srt 值替代了之前的 SRTT, 且 rttvar 为平均偏差的 EWMA, 而非采用先前的 B 来设置 RTO。这组等式也可以写成另一种形式, 对计算机实现来说操作较为方便:

$$Err = M - srtt \quad (14.6)$$

$$srtt \leftarrow srtt + g(Err) \quad (14.7)$$

$$rttvar \leftarrow rttvar + h(|Err| - rttvar) \quad (14.8)$$

$$RTO = srtt + 4(rttvar) \quad (14.9)$$

如前所述,  $srtt$  为均值的 EWMA,  $rttvar$  为绝对误差  $|Err|$  的 EWMA。Err 为测量值  $M$  与当前 RTT 估计值  $srtt$  之间的偏差。 $srtt$  与  $rttvar$  均用于计算 RTO 且随时间变化。增量  $g$  为新 RTT 样本  $M$  占  $srtt$  估计值的权重, 取为  $1/8$ 。增量  $h$  为新平均偏差样本 (新样本  $M$  与当前平均值  $srtt$  之间的绝对误差) 占偏差估计值  $rttvar$  的权重, 取为  $1/4$ 。当 RTT 变化时, 偏差的增量越大, RTO 增长越快。 $8$  和  $4$  的值取为  $2$  的 (负的) 多少次方, 使得整个计算过程较为简单, 对计算机来说只要采用定点整型数的移位和加法操作即可, 而无须复杂的乘除法运算。

标准差公式:

$$\sigma = \sqrt{\frac{\sum (X - \mu)^2}{N}} \quad (14.10)$$

比较经典方法与 Jacobson 的计算方法, 平均 RTT 的计算过程类似 ( $a$  等于  $1$  减增量  $g$ ), 只是采用的增量不同。另外, Jacobson 同时基于平滑 RTT 和平滑偏差计算 RTO, 而经典方法简单采用平滑 RTT 的倍数。这是迄今为止许多 TCP 实现计算 RTO 的方法, 并且由于其作为 [RFC6298] 的基础, 我们称其为标准方法, 尽管在 [RFC6298] 中有一些改进。下面我们就讨论这个问题。

### 时钟粒度与 RTO 边界

在测量 RTT 的过程中, TCP 时钟始终处于运转状态。对初始序列号来说, 实际 TCP 连接的时钟并非从零开始计时, 也没有绝对精确的精度。相反地, TCP 时钟通常某个变量, 该变量值随着系统时钟而做出更新, 但并非一对一地同步更新。TCP 时钟一个“滴答”的时间长度称为粒度。通常, 该值相对较大 (约  $500\text{ms}$ ), 但近期实现的时钟使用更细的粒度 (如 Linux 采用  $1\text{ms}$ )。

粒度会影响 RTT 的测量以及 RTO 的设置。在 [RFC6298] 中, 粒度用于优化 RTO 的更新情况, 并给 RTO 设置了一个下界。计算公式如下!

这里的  $G$  为计时器粒度,  $1000\text{ms}$  为整个 RTO 的下界值 ([RFC6298] 的规则 (2.4) 建议值)。因此, RTO 至少为  $1\text{s}$ , 同时提供了可选上界值, 假设为  $60\text{s}$ 。

### 初始值

我们已经看到估计器怎样随时间进行更新, 但同时也需要了解怎样设置初始值。在首个 SYN 交换前, TCP 无法设置 RTO 初始值。除非系统提供 (有些系统在转发表中缓存了该信息, 见 14.9 节), 否则也无法设置估计器的初始值。根据 [RFC6298], RTO 的初始值为  $1\text{s}$ , 而初始 SYN 报文段采用的超时间隔为  $3\text{s}$ 。当接收到首个 RTT 测量结果  $M$ , 估计器按如下方法进行初始化:

$$srtt + -Mrttvar \leftarrow M/2 \quad (14.11)$$

我们已经了解了估计器的初始化和运行过程。RTO 的设置看似取决于得到的 RTT 采样值，下面我们将看到一些例外情况。

### 重传二义性与 Karn 算法

在测量 RTT 样本的过程中若出现重传，就可能导致某些问题。假这一个包的传输出现超时，该数据包会被重传，接着收到一个确认信息。那么该信息是对第一次还是第二次传输的确认就存在二义性。这就是重传二义性的一个例子。

[KP87]指出，当出现超时重传时，接收到重传数据的确认信息时不能更新 RTT 估计值。这是 Kar 算法的“第一部分”。它通过排除二义性数据来解决 RTT 估算中出现的二义性问题。[RFC6298]做出了相关要求。

假如我们在设置 RTO 过程中简单地将重传问题完全忽略，就可能将网络提供的一些有用信息也同时忽略（即网络中可能出现某些因素影响传输速度）。这种情况下，在网络不再出现丢包前降低重传率有助于减轻网络负担。这也是下面指数退避行为的理论基础，见图 14-1。

TCP 在计算 RTO 过程中采用一个退避系数 (backoff factor)，每当重传计时器出现超时，退避系数加倍，该过程一直持续至接收到非重传数据。此时，退避系数重新设为 1（即二进制指数退避取消），重传计时器返回正常值。对重传过程退避系数加倍，这是 Kar 算法的“第二部分”。注意若 TCP 超时，同时会引发拥塞控制机制，以此改变发送速率（拥塞控制将在第 16 章详细讨论）。因此，Kam 算法实际上由两部分组成，如 [KP89] 所述：

当接收到重复传输（即至少重传一次）数据的确认信息时，不进行该数据包的 RTT 测量，可以避免重传二义性问题。另外，对该数据之后的包采取退避策略。仅当接收到未经重传的数据时，该 SRTT 才用于计算 RTO。

Kam 算法一直作为 TCP 实现中的必要方法（自『RFC1122』起），然而也有例外情况。在使用 TCP 时间戳选项（见第 13 章）的情况下，可以避免二义性问题，因此 Karn 算法的第二部分不适用。

### 带时间戳选项的 RTT 测量

TCP 时间戳选项 (TSOPT) 作 PAWS 算法的基础（第 13 章中已经提过），还可用作 RTT 测量 (RTTM) [RFC1323]。TSOPT 的基本格式第 13 章中已经介绍过。它允许发送者在返回的对应确认信息中携带一个 32 比特的数。

时间戳值 (TSV) 携带于初始 SYN 的 TSOPT 中，并在 SYN+ACK 的 TSOPT 的 TSER 部分返回，以此设定 *srtt*、*rttvar* 与 RTO 的初始值。由于初始 SYN 可看作数据（即同样采取丢失重传策略且占用一个序列号），应测量其 RTT 值。其他报文段中也包含 TSOPT，因此可结合

其他样本值估算该连接的 RTT。该过程看似简单但实际存在很多不确定因素，因为 TCP 并非对其接收到的每个报文段都返回 ACK。例如，当传输大批量数据时，TCP 通常采取每两个报文段返回一个 ACK 的方法（见第 15 章）。另外，当数据出现丢失、失序或重传成功时，TCP 的累积确认机制表明报文段与其 ACK 之间并非严格的一一对应关系。为解决这些问题，使用时间戳选项的 TCP（大部分的 Linux 和 Windows 版本都包含）采用如下算法来测量 RTT 样本值：

1. TCP 发送端在其发送的每个报文段的 TSOPT 的 TSV 部分携带一个 32 比特的时间戳值。该值包含数据发送时刻的 TCP 时钟值。
2. 接收端记录接收到的 TSV 值（名为 TsRecent 的变量）并在对应的 ACK 中返回，并且记录其上一个发送的 ACK 号（名 LastACK 的变量）。回忆一下，ACK 号代表接收端（即 ACK 的发送方）期望接收的下一个有序序号。
3. 当一个新的报文段到达时，如果其序列号与 LastACK 的值吻合（即为下一个期望接收的报文段），则将其 TSV 值存入 TsRecent。
4. 接收端发送的任何一个 ACK 都包含 TSOPT, TsRecent 变量包含的时间戳值被写入其 TSER 部分。
5. 发送端接收到 ACK 后，将当前 TCP 时钟减去 TSER 值，得到的差即为新的 RTT 样本估计值。

FreeBSD、Linux 以及近期的 Windows 版本都默认启用时间戳选项。在 Linux 中，系统配置变量 `net.ipv4.tcp_timestamps` 控制是否使用该选项（0 代表禁用，1 代表使用）。在 Windows 中，通过前面提到的注册表区域的 `Tcp1323opts` 值来控制其使用。若值 0，时间戳被禁用；若值为 2，则启用。该键值没有设默认值（它并非默认存在于注册表中）。但若在连接初始化过程中，TCP 通信的另一方使用时间戳，则默认启用。

### 14.3.3 Linux 采用的方法

Linux 的 RTT 测量过程与标准方法有所差别。它采用的时钟粒度为 1ms，与其他实现方法相比，其粒度更细，TSOPT 也是如此。采用更频繁的 RTT 测量与更细的时钟粒度，RTT 测量也更为精确，但也易于导致 `rttvar` 值随时间减为最小 [LS00]。这是由于当累积了大量的平均偏差样本时，这些样本之间易产生相互抵消的效果。这是其 RTO 设置区别于标准方法的一个原因。另外，当某个 RTT 样本显著低于现有的 RTT 估计值 `srtt` 时，标准方法会增大 `rttvar`。

为更好地理解第二个问题，首先回顾一下 RTO 通常设置为 `srtt + 4 (rttvar)`。因此，无论最大 RTT 样本值是大于还是小于 `srtt`，`rttvar` 的任何大的变动都会导致 RTO 增大。这与直觉相反——若实际 RTT 大幅降低，RTO 并不会因此增大。Linux 通过减小 RTT 样本值大幅下降对

rttvar 的影响来解决这一问题。下面我们详细讨论 Linux 设置 RTO 的方法，该方法可以同时解决上述两个问题。

与标准方法一样，Linux 也记录变量 srt 与 rttvar 值，但同时还记录两个新的变量，即 mdev 和 mdev\_max。mdev 为采用标准方法的瞬时平均偏差估计值，即前面方法的 rttvar。mdev\_max 则记录在测量 RTT 样本过程中的最大 mdev，其最小值不小于 50ms。另外，rttvar 需定期更新以保证其不小于 mdev\_max。因此 RTO 不会小于 200ms。

注意最小 RTO 可更改，这可通过在重新编译加载内核前改变内核配置常量 TCP\_RTO\_MIN 的值来实现。有些 Linux 版本也允许通过 ip route 命令改变该值。若在数据中心网络中使用 TCP，这些环境中 RTT 可能只有几微秒。当本地交换出现丢包时，若 RTO 的最小值力 200ms，就会严重影响网络性能。这就是所谓的 TCP“添头”问题。针对这一问题提出了很多解决方法，包括调整 TCP 时钟粒度；或将最小 RTO 设内几微秒 [IVO9]，但不推荐在全球因特网中使用该方法。

Linux 根据 mdev\_max 的值来更新 rttvar。RTO 总是等于 srt 与 4 (rttvar) 之和，以此确保 RTO 不超过 TCP\_RTO\_MAX (默认值为 120s)。详见 [SK02]。图 14-2 详细描述了这一过程，从中也可看到时间戳选项是怎样工作的。

TCP 时间戳选项携带了发送端 TCP 时钟的副本。接着 ACK 将该值返回至接收端，通过计算两者之差（当前时钟减去返回的时间戳）来更新其 srt 与 rttvar 估计值。为看得更清晰，图中只描述了一部分时间戳。本 Linux 系统中，rttvar 值限制为至少 50ms, RTO 下界值为 200ms

从图 14-2 中可以看到，该 TCP 连接采用时间戳选项。发送端为 Linux 2.6 系统，接收端为 FreeBSD 5.4 系统。为简单起见，序列号和时间戳取相对值，且只显示了发送端的时间戳。为使数据简单可读，本图并未严格按照时间尺度。基于本例中得到的初始 RTT 测量值，Linux 采用如下算法进行更新：

- $srtt = 16ms$
- $mdev = (16/2)ms = 8ms$
- $rttvar = mdev\_max = \max(mdev, TCP\_RTO\_MIN) = \max(8, 50) = 50ms$
- $RTO = srtt + 4 (rttvar) = 16 + 4 (50) = 216ms$

在初始 SYN 交换后，发送端对接收端的 SYN 返回一个 ACK，接收端则进行了一次相应的窗口更新。由于这些包都未包含实际数据（SYN 或 FIN 位字段，但都被算作数据），并没有记录对应的时间，且发送端收到窗口更新时也没有进行 RTT 更新。TCP 对不含数据的报文段不提供可靠传输，意味着若出现丢包不会重传，因此无须设定重传计时器。

值得注意的是，TCP 选项本身并不进行重传或可靠传输。仅当数据段（包括 SYN 和 FIN 报文段）中明确设定，才会丢失重传，但也仅作副作用。



当应用首次执行写操作，发送端 TCP 发送两个报文段，每个报文段包含一个值为 127 的 TSV。由于两次发送间隔小于 1ms（发送端 TCP 时钟粒度），因此这两个值相等。当发送端以这种方式接连发送多个报文段时，很容易看到时钟没有前进或小幅前进的情况。

接收端变量 LastACK 记录其上一个发送 ACK 的序列号。在本例中，上一个发送的 ACK 为连接建立阶段的 SYN+ACK 包，因此 LastACK 从 1 开始。当首个全长（full-size）报文段到达，其序列号与 LastACK 吻合，则将 TsRecent 变量更新为新接收分组的 TSV，即 127。第二个报文段的到达并没有更新 TsRecent，因其序列号字段与 LastACK 中的值并不匹配。接收端返回对应分组的 ACK 时，需在其 TSER 部分包含 TsRecent，同时接收端还要更新 LastACK 变量的 ACK 号为 2801。

当该 ACK 到达时，TCP 就可以进行第二个 RTT 样本的测量。首先获得当前 TCP 时钟值，减去已接收 ACK 包含的 TSER，即样本值  $m=223-127=96$ 。根据该测量值，Linux TCP 按如下步骤更新连接变量：

- $mdev = mdev \cdot (3/4) + m - srtt \cdot (1/4) = 8(3/4) + 80(1/4) = 26ms$
- $mdev\_max = \max(mdev\_max, mdev) = \max(50, 26) = 50ms$
- $srtt = srtt \cdot (7/8) + m \cdot (1/8) = 16(7/8) + 96(1/8) = 14+12=26ms$
- $rttvar = mdev\_max = 50ms$
- $RTO = srtt + 4(rttvar) = 26 + 4(50) = 226ms$

如前所述，Linux TCP 针对经典 RTT 估算方法做了几处改进。在经典算法提出之时，TCP 时钟粒度普遍为 500ms，且时间戳选项也没有得到广泛应用。通常，每个窗口只测量一个 RTT 样本，并据此进行估计器的更新。在不使用时间戳的情况下，依然采用这种方法。

若每个窗口只测量一个 RTT 样本，rttvar 相对变动则较小。利用时间戳和对每个包的测量，就可以得到更多的样本值。因为对同一个窗口的数据而言，每个包对应的 RTT 样本通常存在一定的差异，短时间内得到的大量样本值（如窗口较大）可能导致平均偏差变小（接近 0，基于大数定律 [F68]）。为解决上述问题，Linux 维护瞬时平均偏差估计值 mdev，但设置 RTO 时则基于 rttvar（在一个窗口数据期间记录的最大 mdev，且最小值 50ms）。仅当进入下一个窗口时，rttvar 才可能减小。

标准方法中 rttvar 所占权重较大（系数为 4），因此即使当 RTT 减小时，也会导致 RTO 增长。在时钟粒度较粗时（如 500ms），这种情况不会有很大影响，因为 RTO 可用值很少。然而，若时钟粒度较细，如 Linux 的 1ms，就可能出现这个问题。针对 RTT 减小的情况，若新样本值小于 RTT 估计范围的下界（ $srtt - mdev$ ），则减小新样本的权重。完整的关系式如下：



```

1 if (m < (stt - mdev))
2     mdev = (31/32) * mdev + (1/32) * |srtt - m|
3 else
4     mdev = (3/4) * mdev + (1/4) * |srtt - m|

```

该条件语句只在新 RTT 样本值小于期望的 RTT 测量范围下界的前提下成立。若该条件成立，则表明该连接的 RTT 正处于急剧减小的状态。为避免该情况下的 mdev 增大（以及由此导致的 rttvar 和 RTO 增大），新的平均偏差样本  $|srtt - m|$ ，将其权重减小为原来的 1/8。整体来看，该结果可以避免 RTT 减小导致的 RTO 增大问题。对该问题的进一步讨论，请参见 [LS00] 及 [SK02]。在 [RKS07] 中，作者在 280 万个 TCP 流的多个系统上运行了 RTT 估算算法，运行结果表明 Linux 估计器性能最优，这很大程度上是由于其相对快速收敛，但也可能是减小了 RTT 变动对 RTO 的影响。

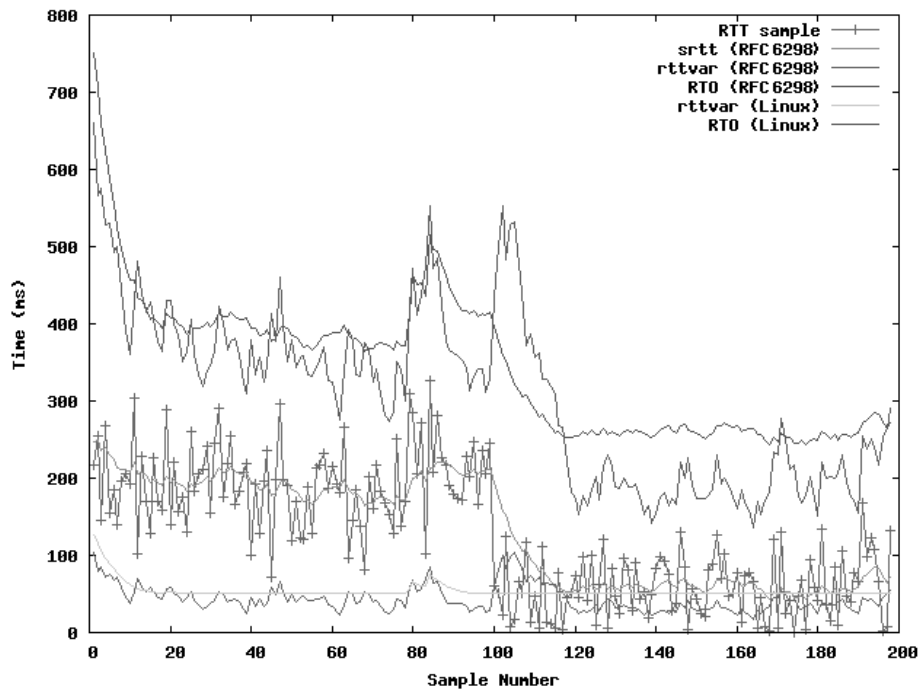


图 14.2: 对于 200 个伪随机的样本点采用 Linux 方法和标准方法来设置 RTO 和估算 RTT。前 100 个点是基于分布  $N(200, 50)$ ，后 100 个则基于  $N(50, 50)$ ，且对负值进行了符号变换。Linux 将最小 RTO 设为 200ms，而标准方法在样本 120 之后则变得更为密集。Linux 避免 RTO 设置过小就是防止这种情况的发生。标准方法在样本 78 和 191 出现潜在问题

现在回到图 14-2，当接收端生成 ACK 7001 时，我们看到其 TSV 包含了一个 TSV 副本，该值并非来自最新到达的报文段，而是最早的一个未经确认的报文段。当该 ACK 返回至发送端，通过计算得到的 RTT 样本是基于第一个报文段，而非第二个。这说明了时间戳算法在延时或不稳定的 ACK 下的工作情况。若计算最早的包对应的 RTT，得到的样本值为发送端期望收

到 ACK 需经过的时间，而非实际网络 RTT。这点很重要，因为发送端需根据其 ACK 接收率来设置 RTO，接收率可能小于包的发送率。

### 14.3.4 RTT 估计器行为

我们已经看到，设置 RTO 与估算 RTT 有大量的设计和改进方法。图 14-3 显示了其中主要的估算方法，即基于标准方法和 Linux 算法得到的综合数据集。图中 [RFC6298] 推荐的标准算法最小 RTO 值 1s 已被移除。目前的大多数 TCP 实现方法都不再采用该值 [RKS07]。

图中显示了两个在高斯概率分布  $N(200,50)$  和  $N(50,50)$  上的 200 个值对应的时间序列图。第一个分布对应前 100 个点，第二个对应后 100 个点。负的样本值通过符号变换转化为正值（只针对第二个分布）。每个加号（+）表示一个具体的样本值。很明显可以看到，在第 100 个样本值之后出现了巨幅下降，另外 Linux 方法在第 100 个样本值之后 RTO 立即减小，而标准方法则在 120 个样本值后才开始减小。

观察 Linux 的 `rttvar` 线，可以看到其基本保持恒定。这是由于 `mdev_max` 的最小值力 50ms（因此 `rtvar` 也是如此），使得 Linux 的 RTO 始终保持在 200ms 以上，并且避免了所有不必要的重传（尽管可能由于 RTO 较大，计时器未超时，导致丢包时性能降低）。标准方法在样本 78 和 191 可能出现潜在问题，即伪重传的发生。这个问题留到后面再讨论。

### 14.3.5 RTTM 对丢包和失序的鲁棒性

当没有丢包情况时，不论接收端是否延迟发送 ACK，TSOPT 可以很好地工作。该算法在以下几种情况下都能正确运行：

- 失序报文段：当接收端收到失序报文段时，通常是由于在此之前出现了丢包，应当立即返回 ACK 以启动快速重传算法（见 14.5 节）。该 ACK 的 TSER 部分包含的 TSV 值为接收端收到最近的有序报文段的时刻（即最新的使窗口前进的报文段，通常不会是失序报文段）。这会使得发送端 RTT 样本增大，由此导致相应的 RTO 增大。这在一定程度上是有利的，即当包失序时，发送端有更多的时间去发现是出现了失序而非丢包，由此可避免不必要的重传。
- 成功重传：当收到接收端缓存中缺失的报文段时（如成功接收重传报文段），窗口通常会前移。此时对应 ACK 中的 TSV 值来自最新到达的报文段，这是比较有利的。若采用原来报文段中的 TSV，可能对应的是前一个 RTO，导致发送端 RTT 估算的偏离。

图 14-4 的例于描述了这些点。假设三个报文段，每个包含 1024 字节，接收顺序如下：报文段 1 包含 1 1024 字节，报文段 3 包含 2049 3027 字节，接着是报文段 2 包含 1025 2048 字节。

当报文段失序，返回的时间戳为最新的使窗口前移的报文段（而非到达接收端的最大的时间戳）。这将使得发送端 RTO 在包失序期间过高估计 RTT，并降低其重传积极性

图 14-4 中发回的 ACK 1025 包含了报文段 1 的时间戳（正常的数据确认），以及另一个包含报文段 1 时间戳的 ACK 1025（对应于在窗口中但失序的重复 ACK），接着是 ACK 3037 包含了报文段 2 的时间戳（而非报文段 3 的时间戳）。当分组失序（或丢失）时，RTT 会被过高估算。较大的 RTT 估计值使得 RTO 也更大，由此发送端也不会急于重传。在失序情况下这是很有利的，因为过分积极的重传可能导致伪重传。

我们已经看到，时间戳选项使得发送端即使在丢包、延时、失序的情况下也能测量 RTT。发送端在测量 RTT 的过程中，可以在其选项中包含任意值，但其单位必须至少和实际时间成比例，且粒度合理，并与 TCP 序列号兼容，连接速率可信（详见 [RFC1323]）。特别是，为了对发送端更有利，对任何可信的 RTT，TCP 时钟必须至少“滴答”一次。另外，其每次变化不能快于 59ns。若小于，在 IP 层允许单个包存在的最大时间（255s）内，记录 TCP 时钟的 32 位的 TSV 值能够环绕 [ID1323b]。满足上述所有条件后，RTO 值就可以用来触发重传。

## 14.4 基于计时器的重传

一旦 TCP 发送端得到了基于时间变化的 RTT 测量值，就能据此设置 RTO，发送报文段时应确保重传计时器设置合理。在设定计时器前，需记录被计时的报文段序列号，若及时收到了该报文段的 ACK，那么计时器被取消。之后发送端发送一个新的数据包时，需设定一个新的计时器，并记录新的序列号。因此每一个 TCP 连接的发送端不断地设定和取消一个重传计时器；如果没有数据丢失，则不会出现计时器超时。

该过程对主机操作系统设计者来说可能难以理解。对典型的操作系统来说，计时器用于标记大量事件，计时器的实现也仅限于有效地设定和触发超时（需要调用系统函数）。然而对 TCP 来说，计时器需要有效地实现被设置、重新设置或取消的功能；若 TCP 正常工作，则计时器不会出现超时的情况。

若在连接设定的 RTO 内，TCP 没有收到被计时报文段的 ACK，将会触发超时重传。我们已经在图 14-1 中看到这一过程。TCP 将超时重传视为相当重要的事件，当发生这种情况时，它通过降低当前数据发送率来对此进行快速响应。实现它有两种方法：第一种方法是基于拥塞控制机制减小发送窗口大小（见第 16 章）；另一种方法为每当一个重传报文段被再次重传时，则增大 RTO 的退避因子，即前面提到的 Kar 算法的“第二部分”。特别是当同一报文段出现多次重传时，RTO 值（暂时性地）乘上值  $y$  来形成新的超时退避值：

$$RTO = yRTO \quad (14.12)$$

在通常环境下,  $\alpha$  值为 1。随着多次重传,  $\alpha$  呈加倍增长: 2, 4, 8, 等等。通常  $\alpha$  不能超过最大退避因子 (Linux 确保其 RTO 设置不能超过 `TCP_RTO_MAX`, 其默认值为 120s)。一旦接收到相应的 ACK,  $\alpha$  会重置为 1。

### 14.4.1 例子

我们通过建立一个与图 14-1 和图 14-2 相似的连接来观察重传计时器的行为。这里故意两次将序列号为 1401 的报文段丢弃 (见图 14-5)。

在本例中, 我们可以利用一个特殊函数将某个序列号的报文段多次丢弃。这与图 14-2 相比, 将会使 RTT 引入一定的延时。连接建立之初与之前类似, 仅当发送序列号为 1 和 1401 的报文段时, 后一个包才被丢弃。当一个报文段到达接收端时, 接收端并没有立即给出响应而是延迟发送 ACK。在 219ms 内都没有得到回应, 发送端的计时器超时, 导致序列号为 1 的包被重传 (此时的 TSV 值为 577)。随即该包的到达使得接收端返回一个 ACK。由于该 ACK 确认了数据被成功接收, 并使得窗口前移, 其 TSER 值被用于更新 srtt 和 RTO 分别为 34 和 234。

接着返回了三个 ACK, 带星号 (\*) 的 ACK 为重复 ACK, 但含了 SACK 信息。我们将在 14.5 节和 14.6 节讨论重复 ACK 和 SACK。现在, 由于这些 ACK 并没有使发送窗口前移, 这些 TSER 值不会被采用。

随着最后一次重传以及报文段 1401 到达 (在 TCP 时钟为 911 的时刻), 修复阶段完成, 接收端返回序列号为 7001 的 ACK, 表明所有数据已成功接收。

当网络无法正常传输数据时, 重传计时器为 TCP 连接提供了“最后一招的重新启动”。在大多数情况下, 计时器超时并触发重传是不必要的 (也不是期望的), 因为 RTO 的取路通常大于 RTT (约 2 倍或更大), 因此基于计时器的重传会导致网络利用率的下降。幸运的是, TCP 有另一种方法来检测和修复丢包, 它比超时重传更为高效。由于它并不需要计时器超时来触发, 因此称为快速重传。

## 14.5 快速重传

快速重传机制 [RFC5681] 基于接收端的反馈信息来引发重传, 而非重传计时器的超时。因此与超时重传相比, 快速重传能更加及时有效地修复丢包情况。典型的 TCP 同时实现了两者。在详细讨论快速重传前, 首先需要了解当接收到失序报文段时, TCP 需要立即生成确认信息 (重复 ACK), 并且失序情况表明在后续数据到达前出现了丢包, 即接收端缓存出现了空缺。发送端的工作即为尽快地、高效地填补该空缺。

当失序数据到达时, 重复 ACK 应立即返回, 不能延时发送。原因在于使发送端尽早得知有失序报文段, 并告诉其空缺在哪。当采用 SACK 时, 重复 ACK 通常也包含 SACK 信息, 利用该信息可以获知多个空缺。

重复 ACK（不论是否包含 SACK 信息）到达发送端表明先前发送的某个分组已丢失。在 14.8 节中我们会更详细地讨论到，重复 ACK 也可能在另一种情况下出现，即当网络中出现失序分组时——若接收端收到当前期盼序列号的后续分组时，当前期盼的包可能丢失，也可能仅为延迟到达。通常我们无法得知是哪种情况，因此 TCP 等待一定数目的重复 ACK（称为重复 ACK 网位或 dupthresh），来决定数据是否丢失并触发快速重传。通常，dupthresh 为常量（值为 3），但一些非标准化的实现方法（包括 Linux）可基于当前的失序程度动态调节该值（见 14.8 节）。

快速重传算法可以概括如下：TCP 发送端在观测到至少 dupthresh 个重复 ACK 后，即重传可能丢失的数据分组，而不必等到重传计时器超时。当然也可以同时发送新的数据。根据重复 ACK 推断的丢包通常与网络拥塞有关，因此伴随快速重传应触发拥塞控制机制（详见第 16 章）。不采用 SACK 时，在接收到有效 ACK 前至多只能重传一个报文段。采用 SACK，ACK 可包含额外信息，使得发送端在每个 RTT 时间内可以填补多个空映。在描述一个基本快速重传算法的例子之后，我们将讨论快速重传中 SACK 的用法。

### 14.5.1 例子

在下面的例子中，我们建立一个与图 14-4 类似的连接，但这次丢本报文段 23801 和 26601，并且禁用 SACK。我们将看到 TCP 怎样利用基本的快速重传算法来填补空缺。发送端为 Linux 2.6 系统，接收端为 FreeBSD 5.4 系统。图 14-6 可通过 Wireshark 的“统计 ITCP 流图 | 时间序列图”（Statistics | TCP Stream Graph | Time-Sequence Graph）功能（toptrace）得到，该图显示了快速重传行为。

该图 y 轴表示相对发送序列号，x 轴表示时间。黑色的 I 形线段表示传输报文段的序列号范围。Wireshark 中的蓝色（图 14-6 中的浅灰色）线段为返回的 ACK 号。约 1.0s 时刻，序列号 23801 发生了快速重传（初始传输不可见，因为被发送端 TCP 下层丢弃）。第三个重复 ACK 的到达触发了快速重传，图中表现为重叠的浅灰色线段。通过 Wireshark 的基本分析窗口也可以观察到重传过程（见图 14-7）。

图 14-7 的第一行（40 号）为 ACK 23801 首次到达。Wireshark 标示出了（红色，在图 14-7 中看来是黑色）其他“有趣的”TCP 包。这些包与其他没有丢失或异常的包不同。我们可以看到窗口更新、重复 ACK 和重传。0.853s 时刻的窗口更新为带重复序列号的 ACK（因为没有携带数据），但包含了 TCP 流控窗口的变动。窗口由 231616 字节变为 233016 字节。因此，它并没有等到三个重复 ACK 来触发快速重传。窗口更新仅是提供了窗口通告的一个副本。我们将在第 15 章中详细讨论。

0.8908s、0.9268s 以及 0.964s 时刻到达的均为序列号为 23801 的重复 ACK。第三个重复 ACK 的到达触发了报文段 23801 的快速重传，时间为 0.993s。该过程也可通过 Wireshark 的“统计 | 流图”（Statistics | Flow Graph）功能来观测（见图 14-8）。

现在我们换个角度来看 0.993s 时刻的快速重传过程，也可以看 1.326s 时刻发生的第二次快

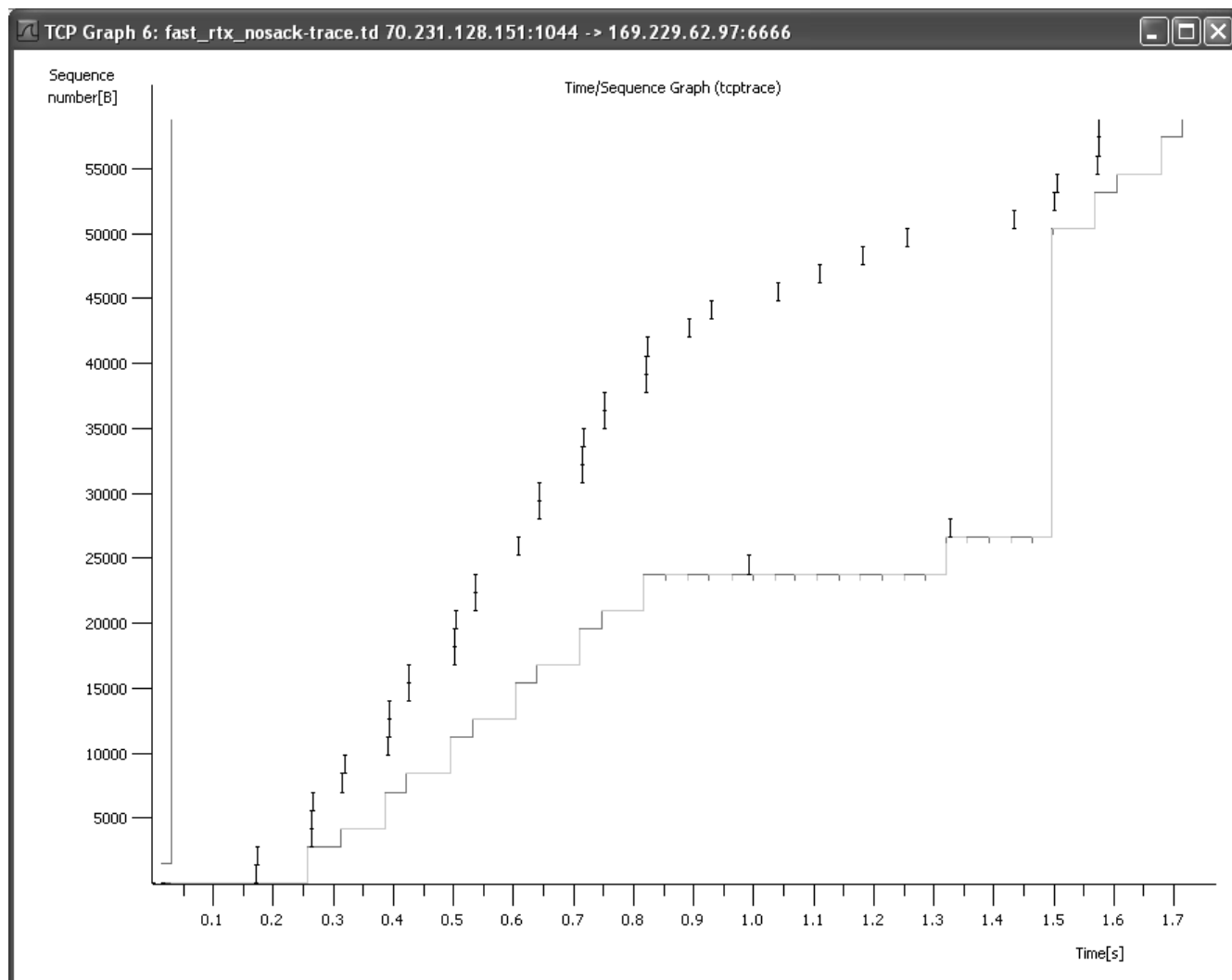


图 14.3: 本图中，轴为 TCP 序列号， $x$  轴为时间。发出的报文段用较深的黑色线段标出，收到的 ACK 号用浅灰线。在  $0.993s$  到达的第三个重复 ACK 触发快速重传。该连接未采用 SACK，所以每个 RTT 内至多只能填补一个空缺。之后到达的重复 ACK 使得发送端发送新报文段（非重传报文段）。在  $1.32s$  时刻到达的“部分 ACK”再次触发了重传

速重传，该重传是由  $1.322s$  时刻达到的 ACK 触发的。

第二次重传与第一次有所不同。当第一次重传发生时，发送端在执行重传前已发送的最大序列号为  $(43401 + 1400 = 44801)$ ，称恢复点 (recovery point)。TCP 在接收到序列号等于或大于恢复点的 ACK 时，才会被认为从重传中恢复。本例中， $1.322s$  和  $1.321s$  时刻的 ACK 并不是 44801，而是 26601。该序列号大于之前接收到的最大 ACK 值 (23801)，但不足以到达恢复点 (44801)。因此这种类型的 ACK 称为部分 ACK (partial ACK)。当部分 ACK 到达时，TCP 发送端立即发送可能丢失的报文段 (这里是 26601)，并且维持这一过程直到到达或超过恢复点。如果拥塞控制机制允许 (见第 16 章)，也可以同时发送新的数据。

| No. | Time     | Source         | Destination  | Protocol | Info  |
|-----|----------|----------------|--------------|----------|---|
| 40  | 0.815379 | 169.229.62.97  | 70.231.128.1 | TCP      | 6666 > 1044 [ACK] Seq=1 Ack=23801 win=231616 Len=0 TSV=488;   |
| 41  | 0.820951 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=37801 Ack=1 win=5808 Len=1400 TSV=29;   |
| 42  | 0.821692 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=39201 Ack=1 win=5808 Len=1400 TSV=29;   |
| 43  | 0.822282 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=40601 Ack=1 win=5808 Len=1400 TSV=29;   |
| 44  | 0.853283 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP window Update] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;  |
| 45  | 0.890255 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#1] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 46  | 0.893441 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=42001 Ack=1 win=5808 Len=1400 TSV=29;   |
| 47  | 0.925942 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#2] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 48  | 0.929728 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=43401 Ack=1 win=5808 Len=1400 TSV=29;   |
| 49  | 0.963646 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#3] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 50  | 0.992938 | 70.231.128.151 | 169.229.62.9 | TCP      | [TCP Retransmission] 1044 > 6666 [ACK] Seq=23801 Ack=1 win=2; |
| 51  | 0.998380 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#4] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 52  | 1.036083 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#5] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 53  | 1.040409 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=44801 Ack=1 win=5808 Len=1400 TSV=29;   |
| 54  | 1.069333 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#6] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 55  | 1.106530 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#7] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 56  | 1.110553 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=46201 Ack=1 win=5808 Len=1400 TSV=29;   |
| 57  | 1.142507 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#8] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 58  | 1.177764 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#9] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;   |
| 59  | 1.182439 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=47601 Ack=1 win=5808 Len=1400 TSV=29;   |
| 60  | 1.212966 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#10] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;  |
| 61  | 1.250471 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#11] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;  |
| 62  | 1.254697 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=49001 Ack=1 win=5808 Len=1400 TSV=29;   |
| 63  | 1.286622 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 44#12] 6666 > 1044 [ACK] Seq=1 Ack=23801 win=2;  |
| 64  | 1.321104 | 169.229.62.97  | 70.231.128.1 | TCP      | 6666 > 1044 [ACK] Seq=1 Ack=26601 win=230216 Len=0 TSV=488;   |
| 65  | 1.321602 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP window Update] 6666 > 1044 [ACK] Seq=1 Ack=26601 win=2;  |
| 66  | 1.326378 | 70.231.128.151 | 169.229.62.9 | TCP      | [TCP Retransmission] 1044 > 6666 [ACK] Seq=26601 Ack=1 win=2; |
| 67  | 1.356099 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 65#1] 6666 > 1044 [ACK] Seq=1 Ack=26601 win=2;   |
| 68  | 1.392063 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 65#2] 6666 > 1044 [ACK] Seq=1 Ack=26601 win=2;   |
| 69  | 1.430013 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 65#3] 6666 > 1044 [ACK] Seq=1 Ack=26601 win=2;   |
| 70  | 1.434094 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=50401 Ack=1 win=5808 Len=1400 TSV=29;   |
| 71  | 1.463026 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP Dup ACK 65#4] 6666 > 1044 [ACK] Seq=1 Ack=26601 win=2;   |
| 72  | 1.497273 | 169.229.62.97  | 70.231.128.1 | TCP      | 6666 > 1044 [ACK] Seq=1 Ack=50401 win=209216 Len=0 TSV=488;   |
| 73  | 1.497989 | 169.229.62.97  | 70.231.128.1 | TCP      | [TCP window Update] 6666 > 1044 [ACK] Seq=1 Ack=50401 win=2;  |
| 74  | 1.501522 | 70.231.128.151 | 169.229.62.9 | TCP      | 1044 > 6666 [ACK] Seq=51801 Ack=1 win=5808 Len=1400 TSV=29;   |

图 14.4: TCP 交换的相对序列号。包 50 和 66 为重传。包 50 是由三次重复 ACK 引发的快速重传。由于没有重传计时器超时，所以恢复过程相对较快

这里的例子并没有采用 SACK，不论是快速重传，还是基于“NewReno”算法 [RFC3782] 恢复阶段执行的其他重传。由于没有 SACK，通过观察返回的 ACK 号的增长情况，发送端在每个 RTT 内只能获知至多一个空缺。

在恢复阶段的具体行为根据 TCP 发送端和接收端的类型和配置差异有所不同。这里描述的是无 SACK 发送端采用 NewReno 算法的例子，这种配置比较常见。根据 NewReno 算法，部分 ACK 只能使发送端继续处于恢复状态。对较旧的 TCP 版本（单纯的 Reno 算法）来说，没有部分 ACK 这个概念，任何一个可接受的 ACK（序列号大于之前接收到的所有 ACK）都能使发送端结束恢复阶段。这种方法可能会使 TCP 出现一些性能问题，我们在第 16 章会详细讨论。下面讨论 NewReno 和 SACK，它们有时也被称为“高级丢失恢复”技术，以此来区别旧的方法。

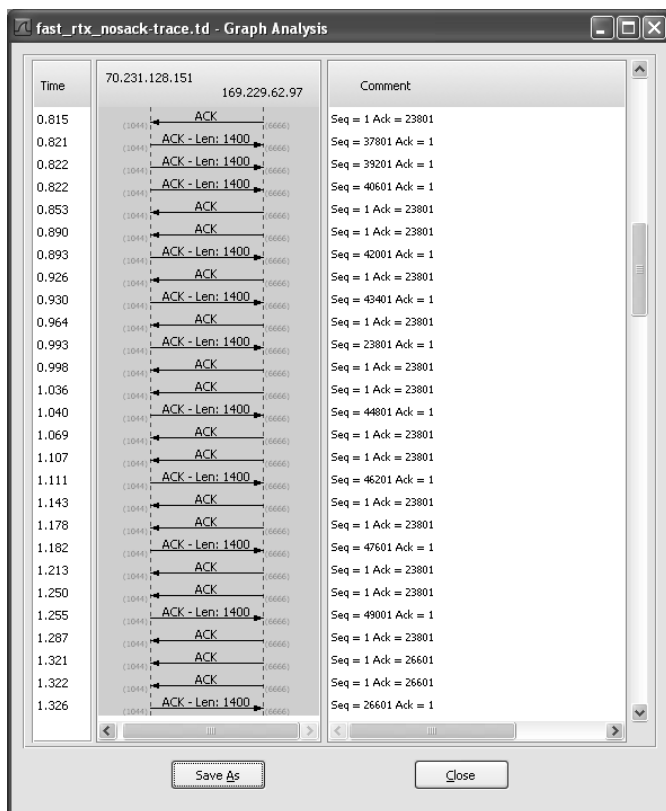


图 14.5: 0.890s、0.926s 以及 0.964s 时刻到达的三次重复 ACK 触发了 0.993s 时刻的快速重传。0.853s 时刻的 ACK 并不算作重复 ACK，因为它包含了一个窗口更新

## 14.6 带选择确认的重传

随着选择确认选项的标准化 [RFC2018]，TCP 接收端可提供 SACK 功能，通过 TCP 头部的累积 ACK 号字段来描述其接收到的数据。之前提到过，ACK 号与接收端缓存中的其他数据之间的间隔称为空缺。序列号高于空缺的数据称为失序数据，因为这些数据和之前接收的序列号不连续。

TCP 发送端的任务是通过重传丢失的数据来填补接收端缓存中的空缺，但同时也要尽可能保证不重传已正确接收到的数据。在很多环境下，合理采用 SACK 信息能更快地实现空缺填补，且能减少不必要的重传，原因在于其在一个 RTT 内能获知多个空缺。当采用 SACK 选项时，一个 ACK 可包含三四个告知失序数据的 SACK 信息。每个 SACK 信息包含 32 位的序列号，代表接收端存储的失序数据的起始至最后一个序列号（加 1）。

SACK 选项指定  $n$  个块的长度为  $8n + 2$  字节，因此 40 字节可包含最多 4 个块。通常 SACK 会与 TSOPT 一同使用，因此需要额外的 10 个字节（外加 2 字节的填充数据），这意味着 SACK 在每个 ACK 中只能包含 3 个块。

3 个块表明可向发送端报告 3 个空缺。若不受拥塞控制（见第 16 章）限制，利用 SACK 选项



可在一个 RTT 时间填补 3 个空映。包含一个或多个 SACK 块的 ACK 有时也简单称为“SACK”。

### 14.6.1 SACK 接收端行为

接收端在 TCP 连接建立期间（见第 13 章）收到 SACK 许可选项即可生成 SACK。通常来说，每当缓存中存在失序数据时，接收端就可生成 SACK。导致数据失序的原因可能是由于传输过程中丢失，也可能是新数据先于旧数据到达。这里只讨论第一种情况，后一种留待以后再讨论。

第一个 SACK 块内包含的是最近接收到的（most recently received）报文段的序列号范围。由于 SACK 选项的空间有限，应尽可能确保向 TCP 发送端提供最新信息。其余的 SACK 块包含的内容也按照接收的先后依次排列。也就是说，最新一个块中包含的内容除了包含最近接收的序列号信息，还需重复之前的 SACK 块（在其他报文段中）。

在一个 SACK 选项中包含多个 SACK 块，并且在多个 SACK 中重复这些块信息的目的在于，为防止 SACK 丢失提供一些备份。若 SACK 不会丢失，[RFC2018] 指出每个 SACK 中包含一个 SACK 块即可实现 SACK 的全部功能。不幸的是，SACK 和普通的 ACK 有时会丢失，并且若其中不包含数据（SYN 或 FIN 控制位字段不被置位）就不会被重传。

### 14.6.2 SACK 发送端行为

尽管一个支持 SACK 的接收端可通过生成合适的 SACK 信息来充分利用 SACK，但还不足以使该 TCP 连接充分利用 SACK 功能。在发送端也应提供 SACK 功能，并且合理地利用接收到的 SACK 块来进行丢失重传，该过程也称为选择性重传（selective retransmission）或选择性重发（selective repeat）。SACK 发送端记录接收到的累积 ACK 信息（像大多数 TCP 发送端一样），还需记录接收到的 SACK 信息，并利用该信息来避免重传正确接收的数据。一种方法是当接收到相应序列号范围的 ACK 时，则在其重传缓存中标记该报文段的选择重传成功。

当 SACK 发送端执行重传时，通常是由于其收到了 SACK 或重复 ACK，它可以选择发送新数据或重传旧数据。SACK 信息提供接收端数据的序列号范围，因此发送端可据此推断需要重传的空缺数据。最简单的方法是使发送端首先填补接收端的空缺，然后再继续发送新数据 [RFC3517]（若拥塞控制机制允许）。这也是最常用的方法。

该行为有一个例外。在 [RFC2018] 中，SACK 选项和 SACK 块的当前规范是建议性的（advisory）。这意味着接收端可能提供一个 SACK 告诉发送端已成功接收一定序列号范围的数据，而之后做出变更（“食言”）。由于这个原因，SACK 发送端不能在收到一个 SACK 后立即清空其重传缓存中的数据；只有当接收端的普通 TCP ACK 号大于其最大序列号值时才可清除。这一规则同样影响重传计时器超时的行为。当 TCP 发送端启动基于计时器的重传时，应忽略 SACK 显示的任何关于接收端数据失序的信息。如果接收端仍存在失序数据，那么重传报文段的 ACK 中就包含附加的 SACK 块，以便发送者使用。幸运的是，食言情况很少出现，也应尽量避免出现。

### 14.6.3 例子

为理解 SACK 怎样影响发送端和接收端的行为，我们重复前面的快速重传实验，参数设置也如前（丢掉序列号 23601 与 28801），但这次发送端和接收端都采用 SACK。为准确观测到实验过程，我们仍采用 Wireshark 的 TCP 序列号（tcptrace）图功能（见图 14-9）。

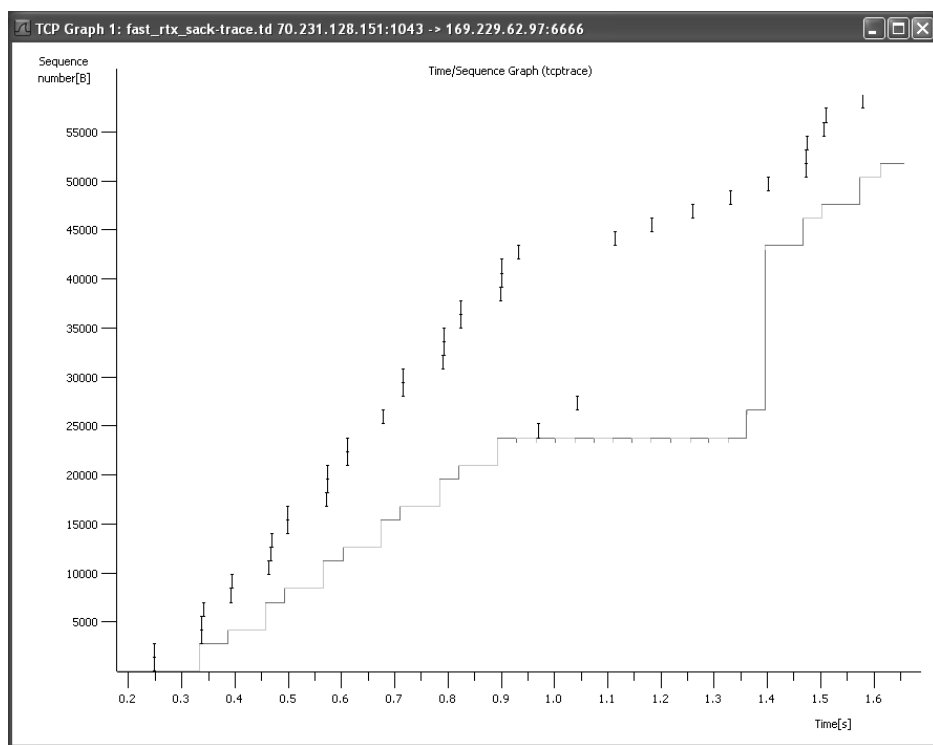


图 14.6: 第一个包含 SACK 信息的重复 ACK 触发了快速重传。后一个 ACK 的到达使得发送端了解到第二个丢失的报文段，并在同一个 RTT 内重传了该报文段

图 14-9 与图 14-6 类似，但利用 SACK 信息，发送端在重传完报文段 23601 后，不必等待一个 RTT 再重传丢失报文段 28801。后面将仔细讨论这些内容，现在我们首先在连接建立过程中验证 SACK 允许（SACK-Permitted）选项的存在，见图 14-10。

与预计的一样，接收端通过 SACK 允许选项来使用 SACK。发送端的 SYN 包，即记录的第一个包，也包含了该选项。这些选项只在连接建立阶段才能看到，因此只出现在 SYN 置位的报文段中。

一旦连接被允许使用 SACK，发生丢包即会使得接收端开始生成 SACK。Wireshark 显示了第一个 SACK 选项的内容（见图 14-11）。

图 14-11 显示了首个 SACK 被接收后的一系列事件。Wireshark 通过 SACK 范围的左右边界来表示 SACK 信息。这里我们看到 23801 的 ACK 包含了一个 SACK 块 [25201,26601]，指明接收端的空缺。接收端敏失的序列号范围为 [23801,25200]，相当于一个从序列号 23801 开始的

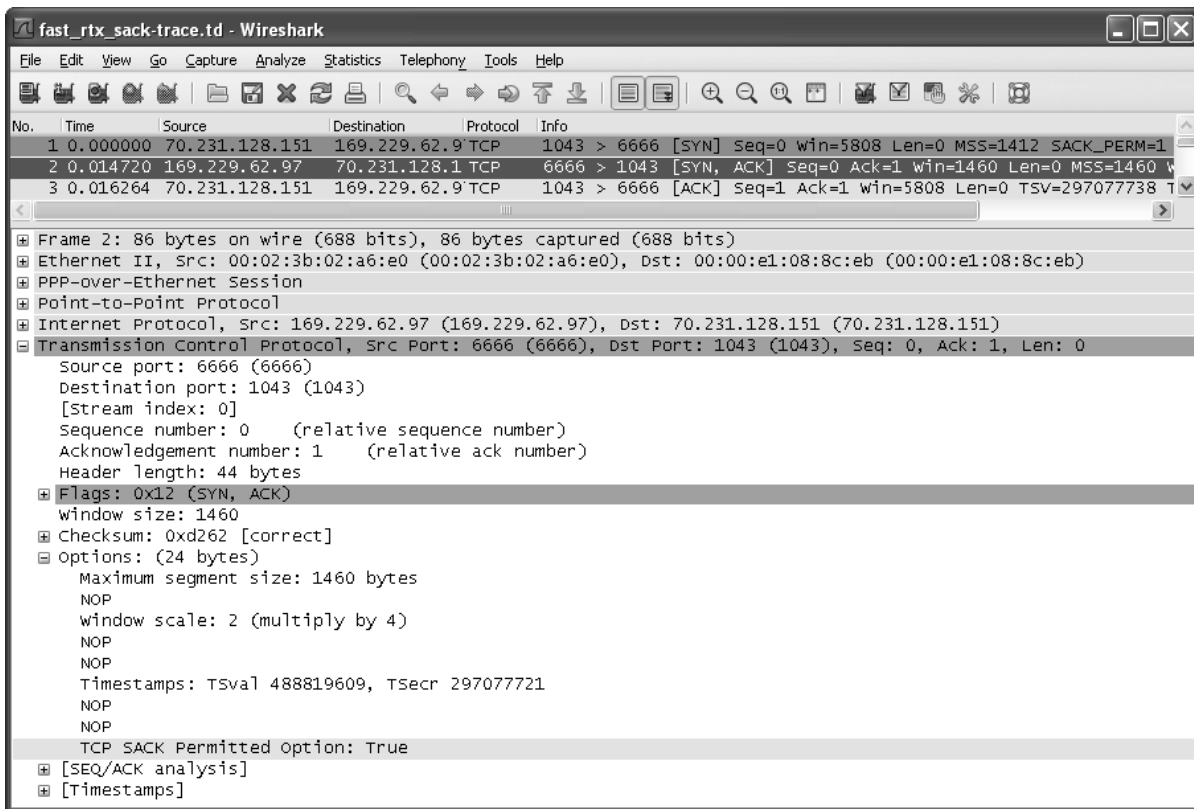


图 14.7: SYN 报文段中 SACK 允许选项表明可生成和发送 SACK 信息。现在的大部分 TCP 版本在连接建立阶段都支持 MSS, 时间戳, 窗口扩大以及 SACK 允许选项

1400 字节的包。注意到该 SACK 为一次窗口更新, 并不能算作重复 ACK, 之前也提到过, 因此不能触发快速重传。

0.967S 时刻到达的 SACK 包含两个块: [28001, 29401] 和 [25201, 26601]。回忆一下前面提过的, 为提高对 ACK 丢失的鲁棒性, 前面 SACK 的第一个块需要重复出现在后续 SACK 的靠后的位置。该 SACK 为序列号 23801 的重复 ACK, 表明接收端现在需要从序列号 23801 至 26601 的两个全长报文段。发送端立即响应, 启动快速重传, 但由于拥塞控制机制 (见第 16 章), 发送端只重传了一个报文段 23801。随着另外的两个 ACK 的到达, 发送端被允许发送第二个重传报文段 26601。

TCP SACK 发送端借鉴了 NewReno 算法中的恢复点的思想。本例中, 在重传前发送的最大序列号为 43400, 低于图 14-5 所示的 NewReno 算法的例子。这里的 SACK 快速重传实现中, 不需要三次重复 ACK; TCP 更早地启动了重传, 但恢复的出口本质上是一致的。一旦接收到序列号 43401 的 ACK, 即 1.3958s 时刻, 恢复阶段即完成。

值得注意的是, 发送端采用 SACK 并不能百分百地提高整体传输性能。我们来看之前讨论过的这两个例子, NewReno 发送端 (非 SACK) 完成 131 074 字节的数据传输用时 3.529s, 而 SACK 发送端则用了 3.674s。尽管这两个值不能这样直接比较, 因为两者所处的网络环境并非绝

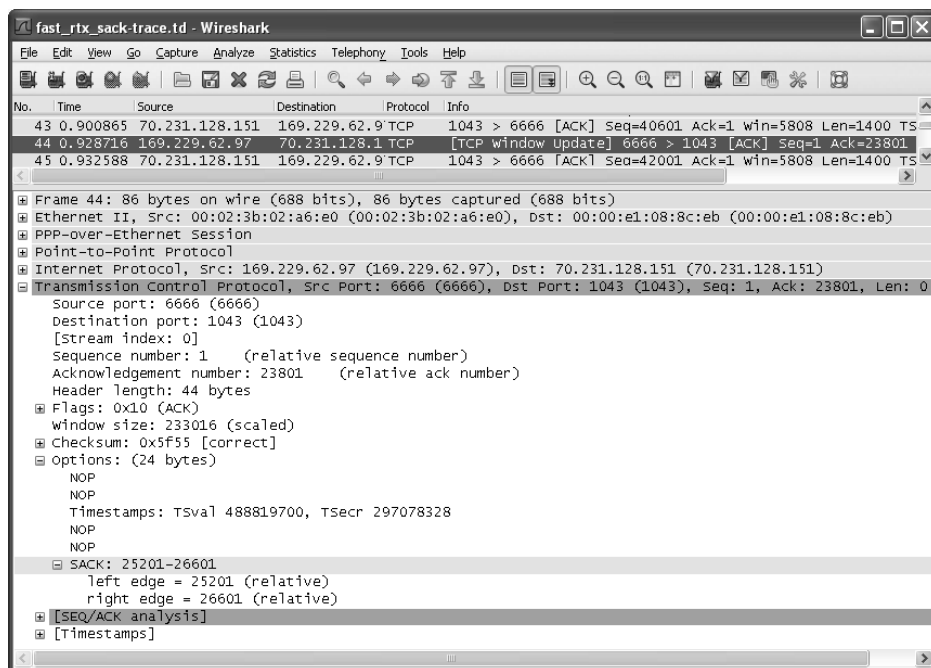


图 14.8: 第一个 ACK 包含 SACK 信息表明失序数据的序列号范围为 25201 至 26601

对相同（这不是仿真实验，而是在真实环境中的测试），但极为近似。在 RTT 较大，丢包严重的情况下，SACK 的优势就能很好地体现出来，因为在这样的环境下，一个 RTT 内能填补多个空缺显得尤为重要。

## 14.7 伪超时与重传

在很多情况下，即使没有出现数据丢失也可能引发重传。这种不必要的重传称为伪重传 (spurious retransmission)，其主要造成原因是伪超时 (spurious timeout)，即过早判定超时，其他因素如包失序、包重复，或 ACK 丢失也可能导致该现象。在实际 RTT 显著增长，超过当前 RTO 时，可能出现伪超时。在下层协议性能变化较大的环境中（如无线环境），这种情况出现得比较多，[KP87] 中也提到。这里我们仅关注由伪超时导致的伪重传。失序与重复的影响在下面的章节中再讨论。

为处理伪超时问题提出了许多方法。这些方法通常包含检测 (detection) 算法与响应 (response) 算法。检测算法用于判断某个超时或基于计时器的重传是否真实，一旦认定出现伪超时则执行响应算法，用于撤销或减轻该超时带来的影响。本章中我们只讨论报文段重传行为。典型的响应算法也涉及拥塞控制变化，会在第 16 章讨论。

图 14-12 描述了一个简化的 TCP 交换过程。在报文段 8 发送完成后 ACK 链路上出现了延迟高峰导致了一次伪重传。在报文段 5 超时重传后，原始传输的报文段 5 8 的 ACK 仍然处于在传状态。本图中为简便起见，序列号和 ACK 号都基于包而非字节来表示，并且 ACK 号表示已

接收到的包，而非期望接收的下一个包。当这些 ACK 到达时，发送端继续重传早已接收的其他报文段，从已确认的报文段之后开始。这导致 TCP 出现了“回退 N”(go-back-N) 的行为模式，并产生了更多的重复 ACK 返回发送端，这时就可能会触发快速重传。针对这一问题，提出了一些方法来减轻不良影响。下面我们讨论其中比较常用的几种方法。

### 14.7.1 重复 SACK (DSACK) 扩展

在非 SACK 的 TCP 中，ACK 只能向发送端告知最大的有序报文段。采用 SACK 则可告知其他的（失序）报文段。基本的 SACK 机制对接收端收到重复数据段时怎样运作没有规定。这些重复数据可能是伪重传、网络中的重复或其他原因造成的。

在传输完报文段 8 后出现了一次延迟高峰，导致了报文段 5 的伪超时和重传。重传完成后，首次传输的报文段 5 对应的 ACK 到达。报文段 5 的重传使得接收端收到了重复报文段，紧接着又重传了报文段 6、7、8，尽管在接收端已存在这些报文段，整个连接还是执行了“回退 N”行为。

在 SACK 接收端采用 DSACK（或称作 D-SACK），即重复 SACK [RFC2883]，并结合通常的 SACK 发送端，可在第一个 SACK 块中告知接收端收到的重复报文段序列号。DSACK 的主要目的是判断何时的重传是不必要的，并了解网络中的其他事项。因此发送端至少可以推断是否发生了包失序、ACK 丢失、包重复或伪重传。

DSACK 相比于传统 SACK 并不需要额外的协商过程。为使其正常工作，接收端返回的 SACK 的内容会有所改变，对应的发送端的响应也会随之变化。如果一个非 DSACK 与 DSACK 的 TCP 共用一个连接，它们会交互操作，但非 DSACK 不能使用 DSACK 的功能。

SACK 接收端的变化在于，允许包含序列号小于（或等于）累积 ACK 号字段的 SACK 块。这并非 SACK 的本意，但这样做能很好地配合该目的。（在 DSACK 信息高于累积 ACK 号字段的情况下，即出现重复的失序报文段时，它也能很好地工作。）DSACK 信息只包含在单个 ACK 中，该 ACK 称为 DSACK。与通常的 SACK 信息不同，DSACK 信息不会在多个 SACK 中重复。因此，DSACK 较通常的 SACK 鲁棒性低。

[RFC2883] 没有具体规定发送端对 DSACK 怎样处理。[RFC3708] 给出了一种实验算法，利用 DSACK 来检测伪重传，但它并没有提供响应算法。它提到可采用 Eifel 响应算法，我们在 14.7.4 节中会讨论，在此之前我们先介绍一些其他的检测算法。

### 14.7.2 Eifel 检测算法

本章开头，我们讨论了重传二义性问题。实验性的 Eifel 检测算法 [RFC3522] 利用了 TCP 的 ISOPT 来检测伪重传。在发生超时重传后，Eifel 算法等待接收下一个 ACK，若为针对第一次传输（即原始传输）的确认，则判定该重传是伪重传。

利用 Eifel 检测算法能比仅来用 DSACK 更早检测到伪重传行为，因为它判断伪重传的 ACK 是在启动丢失恢复之前生成的。相反，DSACK 只有在重复报文段到达接收端后才能发送，并且

在 DSACK 返回至发送端后才能有所响应。及早检测伪重传更为有利，它能使发送端有效避免前面提到的“回退 N”行为。

Eifel 检测算法的机制很简单。它需要使用 TCP 的 TSOPT。当发送一个重传（不论是基于计时器的重传还是快速重传）后，保存其 TSV 值。当接收到相应分组的 ACK 后，检查该 ACK 的 TSER 部分。若 TSER 值小于之前存储的 TSV 值，则可判定该 ACK 对应的是原始传输分组，即该重传是伪重传。这种方法针对 ACK 丢失也有很好的鲁棒性。如果一个 ACK 丢失，后续 ACK 的 TSV 值仍比存储的重传分组的 TSV 小。该窗口内的任一 ACK 的到达都能判断是否出现伪重传，因此单个 ACK 的丢失不会造成太大问题。

Eifel 检测算法可与 DSACK 结合使用，这样可以解决整个窗口的 ACK 信息均丢失，但原始传输和重传分组都成功到达接收端的情况。在这种特殊情况下，重传分组的到达会生成一个 DSACK。Eifel 算法会理所当然地认定出现了伪重传。然而，在出现了如此之多的 ACK 丢失的情况下，使得 TCP 相信该重传不是伪重传是有用的（例如，使其减慢发送速率—采用拥塞控制的后果，第 16 章会讨论）。因此，DSACK 的到达会使得 Eifel 算法认定相应的重传不是伪重传。

### 14.7.3 前移 RTO 恢复 (F-RTO)

前移 RTO 恢复 (Forward-RTO Recovery, F-RTO) [RFC5682] 是检测伪重传的标准算法。它不需要任何 TCP 选项，因此只要在发送端实现该方法后，即使针对不支持 TSOPT 的接收端也能有效地工作。该算法只检测由重传计时器超时引发的伪重传；对之前提到的其他原因引起的伪重传则无法判断。

在一次基于计时器的重传之后，F-RTO 会对 TCP 的常用行为做出一定修改。由于这类重传针对的是没有收到 ACK 信息的最小序列号，通常情况下，TCP 会继续按序发送相邻的分组，这就是前面描述的“回退 N”行为。

F-RTO 会修改 TCP 的行，在超时重传后收到第一个 ACK 时，TCP 会发送新（非重传）数据，之后再响应后一个到达的 ACK。如果其中有一个为重复 ACK，则认为此次重传没问题。如果这两个都不是重复 ACK，则表示该重传是伪重传。这种方法比较直观。如果新数据的传输得到了相应的 ACK，就使得接收端窗口前移。如果新数据的发送导致了重复 ACK，那么接收端至少有一个或更多的空缺。这两种情况下，接收新数据都不会影响整体数据的传输性能（假设接收端有足够的存储空间）。

### 14.7.4 Eifel 响应算法

一旦判断出现伪重传，则会引发一套标准操作，即 Eifel 响应算法 [RFC4015]。由于响应算法逻辑上与 Eifel 检测算法分离，所以它可与我们前面讨论的任一种检测方法结合使用。原则上超时重传和快速重传都可使用 Eifel 响应算法，但目前只针对超时重传做了相关规定。

尽管 Eifel 响应算法可结合其他检测算法使用，但根据是否能尽早（如 Eifel 检测算法或 F-RTO）或较迟（如 DSACK）检测出伪超时的不同而有所区别。前者称为伪超时，通过检查 ACK 或原始传输来实现。后者称为迟伪超时（late spurious timeout），基于由（伪）超时而引发的重传所返回的 ACK 来判定。

响应算法只针对第一种重传事件。若在恢复阶段完成之前再次发生超时，则不会执行响应算法。在重传计时器超时后，它会查看 `srtt` 和 `rttvar` 的值，并按如下方式记录新的变量 `srtt_prev` 和 `rttvar_prev`：

```
srtt_prev = srtt + 2(G)
rttvar_prev = rttvar
```

在任何一次计时器超时后，都会指定这两个变量，但只有在判定出现伪超时才会使用它们，用于设定新的 RTO。在上式中，`G` 代表 TCP 时钟粒度。`srtt_prev` 设为 `srtt` 加上两倍的时钟粒度是由于：`srtt` 的值过小，可能会出现伪超时。如果 `srtt` 稍大，就可能不会发生超时。`srtt` 加上 `2G` 得到 `srtt_prev`，之后都使用 `srtt_prev` 来设置 RTO。

完成 `srtt_prev` 和 `rttvar_prev` 的存储后，就要触发某种检测算法。运行检测算法后可得到一个特殊的值，称为伪恢复（SpuriousRecovery）。如果检测到一次伪超时，则将伪恢复置为 `SPUR_TO`。如果检测到迟伪超时，则将其置为 `LATB_SPUR_TO`。否则，该次超时为正常超时，TCP 继续执行正常的响应行为。

若伪恢复为 `SPUR_TO`，TCP 可在恢复阶段完成之前进行操作。通过将下一个要发送报文段（称为 `SND.NXT`）的序列号修改为最新的未发送过的报文段（称为 `SND.MAX`）。这样就可避免不必要的“回退 N”行为。如果检测到一次迟伪超时，此时已生成对首次重传的 ACK，则 `SND.NXT` 不改变。在以上两种情况中，都要重新设置拥塞控制状态（见第 16 章）。并且一旦接收到重传计时器超时后发送的报文段的 ACK，就要按如下方式更新 `srtt`、`rttvar` 和 RTO 的值：

这里，`m` 是一个 RTT 样本值，它是基于超时后首个发送数据收到的 ACK 而计算得到的。进行这些变量更新的目的在于，实际的 RTT 值可能发生了很大变化，RTT 的当前估计值已经不适合用于设置 RTO。若路径上的实际 RTT 突然增大（例如由于无线切换至一个新的基站），当前的 `srtt` 和 `rttvar` 就显得过小，应重新设置。而另一方面，RTT 的增大可能只是暂时的，这时重设 `srtt` 和 `rttvar` 的值就不那么明智了，因为它们原先的值可能更为精确。

在新 RTT 样本值较大的情况下，上述等式尽力获得上述两种情况的平衡。这样做可以有效地抛弃之前的 RTT 历史值（和 RTT 的历史变化情况）。只有在响应算法中 `srtt` 和 `rttvar` 的值才会增大。若 RTT 不会增大，则维持估计值不变，这本质上是忽略超时情况的发生。两种情况中，RTO 还是按正常方式进行更新，并针对此次超时设置新的重传计时器值。

## 14.8 包失序与包重复

前面讨论的都是 TCP 如何处理包丢失的问题。这是普遍讨论的问题，而且针对包丢失的鲁棒性问题已经做了很多工作。在最后一个章节中可以看到，其他的包传输异常现象，如重复和失序问题也会影响 TCP 操作。在这两种情况中，我们希望 TCP 能区分是出现了失序或重复还是丢失。我们接下来会看到，正确区分这些情况并非易事。

### 14.8.1

在 IP 网络中出现包失序的原因在于 IP 层不能保证包传输是有序进行的。这一方面是有利的（至少对 IP 层来说），因为 IP 可以选择另一条传输链路（例如传输速度更快的路径），而不用担心新发送的分组会先于旧数据到达，这就导致数据的接收顺序与发送顺序不一致。还有其他的原因也会导致包失序。例如，在硬件方面一些高性能路由器会采用多个并行数据链路 [BPS99]，不同的处理延时也会导致包的离开顺序与到达顺序不匹配。

失序问题可能存在于 TCP 连接的正向或反向链路中（也可能两者同时存在）。数据段的失序与 ACK 包的失序对 TCP 的影响有一定差别。回忆一下，由于非对称路由，经常会出现 ACK 经不同于数据包的链路（和不同路由器）传输。

当传输出现失序时，TCP 可能会在某些方面受影响。如果失序发生在反向（ACK）链路，就会使得 TCP 发送端窗口快速前移，接着又可能收到一些显然重复而应被丢弃的 ACK。由于 TCP 的拥塞控制行为（见第 16 章），这种情况会导致发送端出现不必要的流量突发（即瞬时的高速发送）行为，影响可用网络带宽。

如果失序发生在正向链路，TCP 可能无法正确识别失序和丢包。数据的丢失和失序都会导致接收端收到无序的包，造成数据之间的空缺。当失序程度不是很大（如两个相邻的包交换顺序），这种情况可以迅速得到处理。反之，当出现严重失序时，TCP 会误认为数据已经丢失。这就会导致伪重传，主要来自快速重传算法。

回忆一下之前的讨论，快速重传是根据收到重复 ACK 来推断出现丢包并启动重传，而不必等待重传计时器超时。由于 TCP 接收端会对接收到的失序数据立即返回 ACK，以此来帮助触发快速重传，因此网络中任何一个失序的数据包都会生成重复 ACK。由于网络中少量失序情况是常见的，假设一旦收到重复 ACK，发送端即启动快速重传，那么就会导致大量不必要的重传发生。为解决这一问题，快速重传仅在达到重复阈值（dupthresh）后才会被触发。

图 14-13 描述了上述情况。图中左半部分表示在轻微失序的情况下 TCP 的操作，这里的 dupthresh 设为 3。单个重复 ACK 不会影响 TCP 的行。右半部分表示出现严重失序时的情况。由于出现了 3 次失序，对应生成了 3 次重复 ACK，因此触发了快速重传，使得接收端收到了一个重复报文段。

轻微失序（左）中，可忽略少量的重复 ACK。当失序状况较为严重（右）时，这里 4 个包



中有三个出现失序，就会触发伪快速重传

区分丢包与失序不是一个很重要的问题。要解决它需要判断发送端是否已经等待了足够长的时间来填补接收端的空缺。幸运的是，互联网中严重的失序并不常见 [J03]，因此将 dupthresh 设为相对较小值（如缺省值为 3）就能处理绝大部分情况。即便如此，还是有很多研究致力于调整 TCP 行为来应对严重失序 [LLY07]。有些方法可动态调整 dupthresh，如 Linux 的 TCP 实现。

### 14.8.2 重复

尽管出现得比较少，但 IP 协议也可能出现将单个包传输多次的情况。例如，当链路层网络协议执行一次重传，并生成同一个包的两个副本。当重复生成时，TCP 可能出现混淆。考虑图 14-14 中的情况，包 3 重复生成三个副本。

从图中可看到，包 3 的多次重复会使得接收端生成一系列的重复 ACK，这足以触发伪快速重传，使得非 SACK 发送端误认为包 5 与包 6 更早到达。利用 SACK（特别是 DSACK），就可以简单地忽略这个问题。采用 DSACK，每个 A3 的重复 ACK 都包含报文段 3 已成功接收的信息，并且没有包含失序数据信息，意味着到达的包（或 ACK）一定是重复数据。TCP 通常都可防止此类伪重传。

## 14.9 目的度量

从前面的讨论中我们看到，TCP 能不断“学习”发送端与接收端之间的链路特征。学习的结果显示为发送端记录一些状态变量，如 srtt 和 rttvar。一些 TCP 实现也记录一段时间内已出现的失序包的估计值。一般来说，一旦该连接关闭，这些学习结果也会丢失，即与同一个接收端建立一个新的 TCP 连接后，它必须从头开始获得状态变量值。

较新的 TCP 实现维护了这些度量值，即使在连接断开后，也能保存之前存在的路由或转发表项，或其他的一些系统数据结构。当创立一个新的连接时，首先查看数据结构中是否存在与该目的端的先前通信信息。如果存在，则选择较近的信息，据此为 srtt、rttvar 以及其他变量设初值。在 TCP 连接关闭前，可更新统计数据，通过替换现存数据或其他方式的更新来实现。在 Linux 2.6 中，变量值更新为现存值中的最大值和最近测量的数据。可通过 iproute2 [IPR2] 的相关工具来查看这些变量值：

```
Linux# ip route show cache 132.239.50.184
132.239.50.184 from 10.0.0.9 tos 0x10 via 10.0.0.1 dev
mtu 1500 rtt 29ms rttvar 29ms cwnd 2 advmss 1460 hoplimit 64
```

该命令利用特的 DSCP 值 (16, 表示 CS2, 但值为 0x10 代表采用较旧日的“Tos”) 显示了之前连接存储的信息, 本地系统和 132.239.50.184 之间采用 IPv4, 下一跳为 10.0.0.1 网络设备为 ehto。我们可以看到包大小信息 (由 PMTUD 得到路径 MTU, 由远端告知 MSS)、最大跳步数 (针对 IPv6, 这里没有用到)、stt 和 rttvar 的值, 以及第 16 章中会讨论的拥塞控制信息如 cwnd。

## 14.10 重新组包

当 TCP 超时重传, 它并不需要完全重传相同的报文段。TCP 允许执行重新组包 (repacketing), 发送一个更大的报文段来提高性能。(通常该更大的报文段不能超过接收端通告的 MSS, 也不能大于路径 MTU。)允许这样做的原因在于, TCP 是通过字节号来识别发送和接收的数据, 而非报文段 (或包) 号。

TCP 能重传一个与原报文段不同大小的报文段, 这从一定意义上解决了重传二义性问题。STODER [T2205] 就是基于该思想, 采用重新组包的方法来检测伪超时。

我们可以很容易地观察到重新组包的过程。我们采用 sock 程序作为服务器, 并用 Telnet 连接它。首次我们输入一行信息“hello there”。这就生成了一个 13 字节的数据段, 包括回车换行在内。接着断开网络连接, 输入“line number 2”(14 字节, 包括换行)。然后在等待约 45 秒后, 输入“and 3”, 之后关闭连接:

在分析结果中, 省略了初始 SYN 交换过程。前两个报文段包含数据字符串“hellothere”及其确认信息。紧接着的包并非有序: 它从序列号 29 开始, 包含字符串“and 3”(7 个字节)。它返回的 ACK 包含 ACK 号 14, 但 SACK 块的相对序列号为 (29, 36}。中间的数据已丢失。TCP 采用一个更大的包来重传, 包含序列号 14 36。因此, 我们可以看到序列号 14 数据的重传导致了一次重新组包, 形成了 22 字节的较大包来传输。有趣的是, 包中重复包含了 SACK 块中的数据, 同时也将 FIN 位字段置位, 表明这是连接关闭前最后传输的数据。

## 14.11 与 TCP 重传相关的攻击

有一类 DoS 攻击称为低速率 DoS 攻击 [KK03]。在这类攻击中, 攻击者向网关或主机发送大量数据, 使得受害系统持续处于重传超时的状态。由于攻击者可预知受害 TCP 何时启动重传, 并在每次重传时生成并发送大量数据。因此, 受害 TCP 总能感知到拥塞的存在, 根据 Kam 算法不断减小发送速率并退避发送, 导致无法正常使用网络带宽。针对此类攻击的预防方法是随机选择 RTO, 使得攻击者无法预知确切的重传时间。

与 Dos 相关但不同的一种攻击为减慢受害 TCP 的发送, 使得 RTT 估计值过大。这样受害者在丢包后不会立即重传。相反的攻击也是有可能的: 攻击者在数据发送完成但还未到达接收端时伪造 ACK。这样攻击者就能使受害 TCP 认为连接的 RTT 远小于实际值, 导致过分发送, 造

成大量的无效传输。

## 14.12 总结

本章详细讨论了 TCP 超时和重传策略。第一个例子描述了当 TCP 需要发送一个数据包时，简单地暂时断开网络，导致重传计时器超时触发了一次超时重传。每个后续重传与前一次传输都间隔两倍时长，形成二进制指数退避，即 Kar 算法的第二部分。

TCP 测量 RTT 并用这些测量值记录平滑的 RTT 与均值偏差估计值，用这两个估计值计算新的重传超时值。在不采用时间戳选项的情况下，TCP 在每个数据窗口只能测量一个 RTT。Kam 算法通过不测量重传报文段的 RTT 样本值来避免重传二义性问题。现在的大部分 TCP 版本都使用时间戳选项，使得每个报文段都能单独测量。时间戳选项即使在包失序或包重复的情况下也能很好地工作。

我们还讨论了快速重传算法，它在计时器没有超时的情况下就能被触发。该算法可有效地（并最常用）填补由丢包引起的空缺。结合选择确认可更好地提高算法性能。选择确认在 ACK 中携带其他的信息，允许发送端在每个 RTT 内修补多个空触，在某些环境下可有效提高传输性能。

如果 RTT 测量值小于连接的实际值，就可能发生伪重传。在这种情况下，若 TCP 的等待时间稍长，（不必要的）重传就可能不会发生。针对伪超时问题提出了很多算法。DSACK 需要等到接收到重复报文段的 ACK。Eifel 检测方法依据 TCP 时间戳，但它的响应速度能比 DSACK 更快，这是因为它是根据超时前所发送报文段返回的 ACK 来检测伪超时的。F-RTO 与 Eifel 算法类似，但不需要时间戳。它使得发送端在判断出现伪超时后发送新数据。以上这些检测算法都需要结合使用响应算法，我们讨论到的响应算法主要是 Eifel 响应算法。它在延迟大幅增长的情况下（否则对超时不做任何响应），会重新设置 RTT 和 RTT 变化估计值。

我们也讨论了 TCP 怎样存储连接状态，怎样重新组包，以及相关攻击，包括使得 TCP 过分被动或过分积极。在第 16 章讨论 TCP 拥塞控制时，我们会看到更多的此类攻击及其造成的影响。



# TCP 数据流和与窗口管理

---

## 15.1 引言

第 13 章介绍了 TCP 连接的建立和终止，第 14 章则讨论了 TCP 怎样利用丢失数据的重传来保证传输可靠性。下面我们探讨 TCP 的动态数据传输，首先关注交互式连接，接着介绍流量控制以及窗口管理规程。批量数据传输中的拥塞控制策略（参见第 16 章）也包含了相应的窗口管理机制。

“交互式”TCP 连接是指，该连接需要在客户端和服务端之间传输用户输入信息，如按键操作、短消息、操作杆或鼠标的动作等。如果采用较小的报文段来承载这些用户信息，那么传输协议需要耗费很高的代价，因为每个交换分组中包含的有效负载字节较少。反之，报文段较大则会引入更大的延时，对延迟敏感类应用（如在线游戏、协同工具等）造成负面影响。因此我们需要权衡相关因素，找到折中方法。

在讨论交互式通信的相关问题后，会介绍 TCP 流量控制机制。它通过动态调节窗口大小来控制发送端的操作，确保接收端不会溢出。这个方法主要用于批量数据传输（即非交互式通信），但对交互式应用也同样有效。在第 16 章我们会看到，流量控制的思想也可以扩展应用于其他问题，不仅可以保护接收端免于溢出，还可处理中间传输网络的拥塞问题。

## 15.2 交互式通信

在一定时间内，互联网的不同部分传输的网络流量（通常以字节或包来计算）也存在相当大的差异。例如，局域网与广域网以及不同网站之间的流量都会有所不同。TCP 流量研究表明，通常 90% 或者更多的 TCP 报文段都包含大批量数据（如 Web、文件共享、电子邮件、备份），其余部分则包含交互式数据（如远程登录、网络游戏）。批量数据段通常较大（1500 字节或更大），

而交互式数据段则会比较小（几十字节的用户数据）。

对于使用相同协议以及封包格式的数据，TCP 都会处理，但执行的算法有所不同。在本节中我们会讨论 TCP 如何传输交互式数据，以 ssh（安全外壳）应用为例。安全外壳协议 [RFC4251] 是具备较强安全性（基于密码学的加密和认证）的远程登录协议。它已经基本取代了早期的 UNIX `rlogin` 和 `Telnet`，因为这些远程登录服务都存在安全隐患。

通过对 ssh 的探讨，我们会了解延时确认是怎样工作的，以及 Nagle 算法怎样实现减少广域网中较小包的数目。同样的算法也可以用于其他远程登录应用，如 `Telnet`、`rlogin` 和微软终端服务。

对一个 ssh 连接，观察当我们输入一个交互命令后的数据流。客户端获取用户输入信息，然后将其传给服务器端。服务器对命令进行解释并生成响应返回给客户端。客户端对其传输数据加密，意味着用户输入的信息在通过连接传送前已经进行了加密（参见第 18 章）。即使传输数据被截获，窃听者也很难获得用户输入信息的明文。客户端支持多种加密算法和认证方法。它也支持一些新的特性，如隧道技术实现对其他协议的封装（参见第 3 章及 [RFC4254]）。

许多 TCP/IP 的初学者会惊奇地发现，每个交互按键通常都会生成一个单独的数据包。也就是说，每个按键是独立传输的（每次一个字符而非每次一行）。另外，ssh 会在远程系统（服务器端）调用一个 shell（命令解释器），对客户端的输入字符做出回显。因此，每个输入的字符会生成 4 个 TCP 数据段：客户端的交互按键输入、服务器端对按键的确认、服务器端生成的回显、客户端对该回显的确认（参见图 15-1a）。

通常，第 2 和第 3 段可以合并，如图 15-1b 所示，可将对按键的确认与回显一并传送。下一节会介绍这种方法（称为捎带延时确认）。

- a) 对一次交互按键的远程回显，一种可行的方法是将对按键的确认与回显包各自独立发送。
- b) 典型 TCP 则将两者结合传输

我们以 ssh 为例的原因在于，对从客户端到服务器键入的每个字符都会生成一个独立的包。然而，若用户的输入速度较快，每个包可能包含多个字符。图 15-2 显示了在 ssh 连接至 Linux 服务器中输入 `date` 命令，利用 Wireshark 获得的数据流。

如图 15-2 所示，包 1 包含了客户端到服务器端的命令字符 `d`。包 2 为对字符 `d` 的确认和回显（如图 15-1 中将两段结合传送）。包 3 为对回显字符 `d` 的确认。同理，包 4~6 对应字符 `i`，包 7 对应字符 `l，包 10 12 对应字符 e。包 13 15 则对应回车键。在包 3+6 7、9 10 和 12 13 之间的时间差为人工输入每个字符的延迟，这里特意设置得较长（约 1.5 秒）。`

注意到包 16 19 与前面的包稍有差异，包长度从 48 字节变为 64 字节。包 16 包含了服务器端对 `date` 命令的输出。这 64 字节的数据是对下面 28 个明文（未加密）字符的加密结果：

```
Wed Dec 28 22:47:16 PST 2005
```

加上最后的回车和换行符。下一个从服务器端发送至客户端的包（包 18）包含了服务器主机对客户的命令提示符：`Linux%`。包 19 为对该数据的确认。

The image shows a Wireshark capture of an SSH session. The packet list on the left shows 19 packets. The packet details pane on the right shows the structure of the selected packet (No. 19), which is a TCP segment. The packet is an ACK with sequence number 241, acknowledgment number 369, and window size 4220. The packet length is 64 bytes.

| No. | Time     | Source        | Destination   | Protocol | Info  |
|-----|----------|---------------|---------------|----------|---|
| 1   | 0.000000 | 70.231.141.59 | 169.229.62.97 | SSH      | Encrypted request packet len=48   |
| 2   | 0.014508 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=48  |
| 3   | 0.014769 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=49 Ack=49 win=4220 Len=0 TSV=913185368 TSER=114503261   |
| 4   | 1.736761 | 70.231.141.59 | 169.229.62.97 | SSH      | Encrypted request packet len=48   |
| 5   | 1.751620 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=48  |
| 6   | 1.751840 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=97 Ack=97 win=4220 Len=0 TSV=913187106 TSER=114503435   |
| 7   | 3.284481 | 70.231.141.59 | 169.229.62.97 | SSH      | Encrypted request packet len=48   |
| 8   | 3.299718 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=48  |
| 9   | 3.299937 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=145 Ack=145 win=4220 Len=0 TSV=913188654 TSER=114503590 |
| 10  | 4.982810 | 70.231.141.59 | 169.229.62.97 | SSH      | Encrypted request packet len=48   |
| 11  | 4.997635 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=48  |
| 12  | 4.997858 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=193 Ack=193 win=4220 Len=0 TSV=913190352 TSER=114503759 |
| 13  | 6.626947 | 70.231.141.59 | 169.229.62.97 | SSH      | Encrypted request packet len=48   |
| 14  | 6.642338 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=48  |
| 15  | 6.642557 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=241 win=4220 Len=0 TSV=913191997 TSER=114503924 |
| 16  | 6.644846 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=64  |
| 17  | 6.645054 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=305 win=4220 Len=0 TSV=913192000 TSER=114503924 |
| 18  | 6.646053 | 169.229.62.97 | 70.231.141.59 | SSH      | Encrypted response packet len=64  |
| 19  | 6.646251 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=369 win=4220 Len=0 TSV=913192001 TSER=114503924 |

图 15.1: 变得更为密集。Linux 避免 RTO 设置过小就是防止这种情况的发生。标准方法在样本 78 和 191 出现潜在问题

图 15-3 描述与图 15-2 相同的传输情况，只是细化了 TCP 层的信息，可以更清晰地看到 TCP 怎样进行确认以及 ssh 使用的包大小。包 1（包含字符 d）的相对序列号从 0 开始。包 2 是对图中包 1 的确认，ACK 号设为 48，为上次成功接收字节的序列号加 1。包 2 也包含了服务器至客户端的对 d 字符的回显，字节序列号为 0。包 3 为客户端对该回显的确认，ACK 号设为 48。可以看到，该连接包含了两个序列号流个是从客户端至服务器，另一个为相反方向。在介绍窗口通告时将会详细讨论这一问题。

The image shows a Wireshark capture of an SSH session with protocol decoding disabled. The packet list on the left shows 19 packets. The packet details pane on the right shows the structure of the selected packet (No. 19), which is a TCP segment. The packet is an ACK with sequence number 241, acknowledgment number 369, and window size 4220. The packet length is 64 bytes.

| No. | Time     | Source        | Destination   | Protocol | Info   |
|-----|----------|---------------|---------------|----------|--|
| 1   | 0.000000 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [PSH, ACK] Seq=1 Ack=1 win=4220 Len=48 TSV=913185354 TSER=114501133      |
| 2   | 0.014508 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=1 Ack=49 win=32900 Len=48 TSV=114503261 TSER=913185354    |
| 3   | 0.014769 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=49 Ack=49 win=4220 Len=0 TSV=913185368 TSER=114503261          |
| 4   | 1.736761 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [PSH, ACK] Seq=49 Ack=49 win=4220 Len=48 TSV=913187091 TSER=114503261    |
| 5   | 1.751620 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=49 Ack=97 win=32900 Len=48 TSV=114503435 TSER=913187091   |
| 6   | 1.751840 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=97 Ack=97 win=4220 Len=0 TSV=913187106 TSER=114503435          |
| 7   | 3.284481 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [PSH, ACK] Seq=97 Ack=97 win=4220 Len=48 TSV=913188639 TSER=114503435    |
| 8   | 3.299718 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=97 Ack=145 win=32900 Len=48 TSV=114503590 TSER=913188639  |
| 9   | 3.299937 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=145 Ack=145 win=4220 Len=0 TSV=913188654 TSER=114503590        |
| 10  | 4.982810 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [PSH, ACK] Seq=145 Ack=145 win=4220 Len=48 TSV=913190337 TSER=114503590  |
| 11  | 4.997635 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=145 Ack=193 win=32900 Len=48 TSV=114503759 TSER=913190337 |
| 12  | 4.997858 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=193 Ack=193 win=4220 Len=0 TSV=913190352 TSER=114503759        |
| 13  | 6.626947 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [PSH, ACK] Seq=193 Ack=193 win=4220 Len=48 TSV=913191982 TSER=114503759  |
| 14  | 6.642338 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=193 Ack=241 win=32900 Len=48 TSV=114503924 TSER=913191982 |
| 15  | 6.642557 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=241 win=4220 Len=0 TSV=913191997 TSER=114503924        |
| 16  | 6.644846 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=241 Ack=241 win=32900 Len=64 TSV=114503924 TSER=913191982 |
| 17  | 6.645054 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=305 win=4220 Len=0 TSV=913192000 TSER=114503924        |
| 18  | 6.646053 | 169.229.62.97 | 70.231.141.59 | TCP      | 22 > 1058 [PSH, ACK] Seq=305 Ack=241 win=32900 Len=64 TSV=114503924 TSER=913191982 |
| 19  | 6.646251 | 70.231.141.59 | 169.229.62.97 | TCP      | 1058 > 22 [ACK] Seq=241 Ack=369 win=4220 Len=0 TSV=913192001 TSER=114503924        |

图 15.2: 与图 15-2 相同，只是这里禁用了 ssh 的协议解码，因此可以看到 TCP 序列号信息。注意到除了最后两个包外，其他包都为 48 字节，该长度与 ssh 使用的加密算法有关（参见第 18 章）

另外我们也发现，每个带数据（长度不为 0）的包都将 PSH 置位。之前提到过，该标志位通常表示发送端缓存为空。也就是说，当 PSH 置位的数据包发送完成后，发送端没有其他数据包需要传输。

## 15.3 延时确认

在许多情况下, TCP 并不对每个到来的数据包都返回 ACK, 利用 TCP 的累积 ACK 字段 (参见第 12 章) 就能实现该功能。累积确认可以允许 TCP 延迟一段时间发送 ACK, 以便将 ACK 和相同方向上需要传的数据结合发送。这种捎带传输的方法经常用于批量数据传输。显然, TCP 不能任意时长地延迟 ACK; 否则对方会误认为数据丢失而出现不必要的重传。

主机需求 RFC [RFC1122] 指出, TCP 实现 ACK 延迟的时延应小于 500ms。实践中时延最大取 200ms。

采用延时 ACK 的方法会减少 ACK 传输数目, 可以一定程度地减轻网络负载。对于批量数据传输通常为 2:1 的比例。基于不同的主机操作系统, 延迟发送 ACK 的最大时延可以动态配置。Linux 使用了一种动态调节算法, 可以在每个报文段返回一个 ACK (称次“快速确认”模式) 与传统延时 ACK 模式间相互切换。Mac OS X 中, 可以改变系统变量 `net.inet.top.delayed_ack` 值来设置延时 ACK。可选值如下: 禁用延时 (设为 0), 始终延时 (设为 1), 每隔一个包回复一个 ACK (设为 2), 自动检测确认时间 (设次 3)。默认值为 3。最新的 Windows 版本中, 注册表项

`HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces\IG`

中, 每个接口的全局唯一标识 (GUID) 都不同 (IG 表示被引用的特定网络接口的 GUID)。TepAckFrequency 值 (需要被添加) 可以设为 0 255, 默认为 2。它代表延时 ACK 计时器超时前在传的 ACK 数目。将其设为 1 表明对每个收到的报文段都生成相应的 ACK。ACK 计时器值可以通过 TopDelAckTicks 注册表项控制。该值可设为 2 6, 默认为 2。它以百毫秒为单位, 表明在发送延时 ACK 前要等待百毫秒数。

之前提到过, 通常 TCP 在某些情况下使用延时 ACK 的方法, 但时延不会很长。在第 16 章中大量采用了延时 ACK 的方法, 我们将会看到 TCP 怎样在处理批量数据的大数据包传输中实现拥塞控制。在小数据包传输中, 如交互式应用, 需要采用另外的算法。将该算法与延时 ACK 结合使用, 如果处理不好, 反而会导致性能降低。下面我们详细讨论该算法。

## 15.4 Nagle 算法

从前面的小节中可以知道, 在 ssh 连接中, 通常单次击键就会引发数据流的传输。如果使用 IPv4, 一次按键会生成约 88 字节大小的 TCP/IPv4 包 (使用加密和认证): 20 字节的『头部, 20 字节的 TCP 头部 (假设没有选项), 数据部分为 48 字节。这些小包 (称为微型报 (tinygram)) 会造成相当高的网络传输代价。也就是说, 与包的其他部分相比, 有效的应用数据所占比例甚微。



该问题对于局域网不会有很大影响，因为大部分局域网不存在拥塞，而且这些包无须传输很远。然而对于广域网来说则会加重拥塞，严重影响网络性能。John Nagle 在 [RFC0896] 中提出了一种简单有效的解决方法，现在称其为 Nagle 算法。下面首先介绍该算法是怎样运行的，接着我们会讨论结合延时 ACK 方法使用时可能出现的一些缺陷和问题。

Nagle 算法要求，当一个 TCP 连接中有在传数据（即那些已发送但还未经确认的数据），小的报文段（长度小于 SMSS）就不能被发送，直到所有的在传数据都收到 ACK。并且，在收到 ACK 后，TCP 需要收集这些小数据，将其整合到一个报文段中发送。这种方法迫使 TCP 遵循停等（stop-and-wait）规程—只有等接收到所有在传数据的 ACK 后才能继续发送。该算法的微妙之处在于它实现了自时钟（self-clocking）控制：ACK 返回越快，数据传输也越快。在相对高延迟的广域网中，更需要减少微型报的数目，该算法使得单位时间内发送的报文段数目更少。也就是说，RTT 控制着发包速率。

从图 15-3 中可以看到，单个字节的发送、确认以及回显的 RTT 较小（15ms 以下）。为更快地生成数据，我们需要每秒输入 60 个字符以上。这意味着，当两台主机之间以很小的 RTT 传输数据时，例如在同一个局域网中，我们将很难看到该算法的显著效果。

为了显示 Nagle 算法的效果，我们比较分析某个 TCP 应用使用和禁用该算法的行为。我们对一个 ssh 版本的客户端做了一定的修改。利用一个 RTT 相对较大（约 190ms）的连接，就可以看出区别。首先观察禁用 Nagle 算法（ssh 默认）的情况，如图 15-4 所示。

| No. | Time     | Source         | Destination    | Protocol | Info  |
|-----|----------|----------------|----------------|----------|---|
| 1   | 0.000000 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [PSH, ACK] Seq=1 Ack=1 win=8320 Len=48 TSV=134534 |
| 2   | 0.069366 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [PSH, ACK] Seq=49 Ack=1 win=8320 Len=48 TSV=13453 |
| 3   | 0.189457 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=1 Ack=49 win=10880 Len=48 TSV=2133 |
| 4   | 0.189706 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=97 Ack=49 win=8320 Len=0 TSV=134535023  |
| 5   | 0.202758 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [PSH, ACK] Seq=97 Ack=49 win=8320 Len=48 TSV=134  |
| 6   | 0.232567 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [PSH, ACK] Seq=145 Ack=49 win=8320 Len=48 TSV=134 |
| 7   | 0.260124 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=49 Ack=97 win=10880 Len=48 TSV=213 |
| 8   | 0.300289 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=193 Ack=97 win=8320 Len=0 TSV=134535134 |
| 9   | 0.377729 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [PSH, ACK] Seq=193 Ack=97 win=8320 Len=48 TSV=134 |
| 10  | 0.393425 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=97 Ack=145 win=10880 Len=48 TSV=2  |
| 11  | 0.393647 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=241 Ack=145 win=8320 Len=0 TSV=1345352  |
| 12  | 0.421981 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=145 Ack=193 win=10880 Len=48 TSV=  |
| 13  | 0.422435 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=241 Ack=193 win=8320 Len=0 TSV=1345352  |
| 14  | 0.567368 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=193 Ack=241 win=10880 Len=48 TSV=  |
| 15  | 0.567784 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=241 Ack=241 win=8320 Len=0 TSV=1345354  |
| 16  | 0.572460 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=241 Ack=241 win=10880 Len=64 TSV=  |
| 17  | 0.572797 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=241 Ack=305 win=8320 Len=0 TSV=1345354  |
| 18  | 0.581490 | 193.10.133.128 | 70.231.143.234 | TCP      | 22 > 1055 [PSH, ACK] Seq=305 Ack=241 win=10880 Len=112 TSV= |
| 19  | 0.581905 | 70.231.143.234 | 193.10.133.128 | TCP      | 1055 > 22 [ACK] Seq=241 Ack=417 win=8320 Len=0 TSV=1345354  |

图 15.3: ssh 分析文件显示该 TCP 连接 RTT 约为 190ms。Nagle 算法被禁用。数据和 ACK 结合传输，19 个包传输持续了 0.58s。许多包相对较小（48 字节的用户数据）。纯 ACK（不包含数据的报文段）表明服务器端的输出命令已被客户端接收处理

图 15-4 中显示的传输是在初始的认证完成以后、登录会话开始时记录的。这时输入 date 命令，我们看到共捕获到了 19 个包，整个传输过程持续了 0.58S。共有 5 个 ssh 请求包，7 个 ssh 应答包，以及 7 个 TCP 层的纯 ACK 包（不包含数据）。下面我们将在使用 Nagle 算法的情况下重复探测这一过程（即在相似的网络环境下），可以得到图 15-5。

RTT 为 190ms 并启用 Nagle 算法的 TCP 连接的 ssh 传输情况。请求和响应传输规律，紧密一致。整个传输过程持续了 0.80s，共有 11 个包

可以看到图 15-5 中的包数目要少于图 15-4（少了 8 个）。另外一个明显的差异是，请求和响应包随时间分布呈一定的规律性。回想一下 Nagle 算法的原理，它迫使 TCP 遵循停等行为模式，因此 TCP 发送端只有在接收到全部 ACK 后才能继续发送。观察每组请求/响应的传输时刻—0.0、0.19、0.38 以及 0.57，我们可以发现它们遵循一定的模式：每两个间隔为 190ms，恰为连接的 RTT。每发送一组请求和响应包需要等待一个 RTT，这就加长了整个传输过程（需要 0.80s 而非前面的 0.58s）。Nagle 算法做出了一种折中：传输的包数目更少而长度更大，但同时传输时延也 longer。从图 15-6 中可以更清晰地看出差别。

图 15-6 显示了 Nagle 算法的停等行为。左侧显示双向传输，而右侧使用 Nagle 算法，使得在任一给定时刻，只有一个方向保持传输状态。

### 15.4.1 延时 ACK 与 Nagle 算法结合

若将延时 ACK 与 Nagle 算法直接结合使用，得到的效果可能不尽如人意。考虑如下情形，客户端使用延时 ACK 方法发送一个对服务器的请求，而服务器端的响应数据并不适合在同一个包中传输（参见图 15-7）。

从图中可以看到，在接收到来自服务器端的两个包以后，客户端并不立即发送 ACK，而是处于等待状态，希望有数据一同捎带发送。通常情况下，TCP 在接收到两个全长的数据包后就应返回一个 ACK，但这里并非如此。在服务器端，由于使用了 Nagle 算法，直到收到 ACK 前都不能发送新数据，因为任一时刻只允许至多一个包在传。因此延时 ACK 与 Nagle 算法的结合导致了某种程度的死锁（两端互相等待对方做出行动）[MMSV99] [MMO1]。幸运的是，这种死锁并不是永久的，在延时 ACK 计时器超时后死锁会解除。客户端即使仍然没有要发送的数据也无需再等待，而可以只发送 ACK 给服务器。然而，在死锁期间整个传输连接处于空闲状态，使性能变差。在某些情况下，如这里的 ssh 传输，可以禁用 Nagle 算法。

### 15.4.2 禁用 Nagle 算法

从前面的例子可以看到，在有些情况下并不适用 Nagle 算法。典型的包括那些要求时延尽量小的应用，如远程控制中鼠标或按键操作需要及时送达以得到快捷的反馈。另一个例子是多人网络游戏，人物的动作需要及时地传送以确保不影响游戏进程（也不致影响其他玩家的动作）。

禁用 Nagle 算法有多种方式，主机需求 RFC [RFC1122] 列出了相关方法。若某个应用使用 Berkeley 套接字 API，可以设置 `TCP_NODELAY` 选项。另外，也可以在整个系统中禁用该算法。在 Windows 系统中，使用如下的注册表项：

```
HKLM\ SOFTWARE \Microsoft \MSMQ \Parameters\ TCPNoDelay
```

这个双字节类型的值必须由用户添加，应将其设为 1。为使更改生效，消息队列也需要重新设置。

## 15.5 流量控制与窗口管理

回顾一下第 12 章中提到过，可以采用可变滑动窗口来实现流量控制。如图 15-8 所示，TCP 客户端和服务端交互作用，互相提供数据流的相关信息，包括报文段序列号、ACK 号和窗口大小（即接收端的可用空间）。

每个 TCP 连接都是双向的。数据传输方向的另一端会返回 ACK 及其窗口通告信息。反向亦然。

图 15-8 中两个大的箭头表示数据流方向（TCP 报文段的传输方向）。每个 TCP 都是双向连接，这里用两个箭头表示，一个是客户端至服务器方向（C—S），另一个为服务器至客户端方向（S—C）。每个报文段包含 ACK 和窗口信息，可能还有用户数据。根据数据流传输方向的不同，将 TCP 头部中的字段标记上阴影。例如，在 C—S 方向的数据流为下方箭头的报文段，但是对该数据的 ACK 和窗口信息却在上方箭头指示的报文段中。每个 TCP 报文段（除了连接建立之初的包交换）都包含一个有效的序列号字段、一个 ACK 号或确认字段，以及一个窗口大小字段（包含窗口通告信息）。

在前面的 ssh 示例中，我们看到的窗口通告都是固定的，有 8320 字节、4220 字节，还有 32900 字节。这些数值表示发送该窗口信息的通信方为即将到来的新数据预留的存储空间。当 TCP 应用程序空闲时，就会排队处理这些数据，致使窗口大小字段保持不变。当系统处理速度较慢，或者程序忙于执行其他操作，到来的数据返回 ACK 后，就需要排队等待被读取或“消耗”。若这种排队状况持续，新数据的可用存储空间就会减小，窗口大小值也随之减小。最终，若应用程序一直不处理这些数据，TCP 必须采取策略使得发送端完全停止新数据的发送，因为可能没有空间来存储新数据。此时就可以将窗口通告设为 0（没有空间）。

每个 TCP 头部的窗口大小字段表明接收端可用缓存空间的大小，以字节为单位。该字段长度为 16 位，但窗口缩放选项可用大于 65 535 的值（参见第 13 章）。报文段发送方在相反方向上可接受的最大序列号值为 TCP 头部中 ACK 号和窗口大小字段之和（保持单位一致）。

### 15.5.1 滑动窗口

TCP 连接的每一端都可收发数据。连接的收发数据量是通过一组窗口结构（window structure）来维护的。每个 TCP 活动连接的两端都维护一个发送窗口结构（send window structure）和接收窗口结构（receive window structure）。这些结构与第 12 章中描述的概念窗口结构类似，这里我们将详细讨论。图 15-9 显示了一个假设的 TCP 发送窗口结构。

TCP 发送端滑动窗口结构记录了已确认、在传以及还未传的数据的序列号。提供窗口的大小是由接收端返回的 ACK 中的窗口大小字段控制的。

TCP 以字节（而非包）为单位维护其窗口结构。在图 15-9 中，我们已标号为 211 字节。由接收端通告的窗口称为提供窗口（offered Window），包含 49 字节。接收端已成功确认包括第 3 字节在内的之前的数据，并通告了一个 6 字节大小的窗口。回顾第 12 章，窗口大小字段相对 ACK 号有一个字节的偏移量。发送端计算其可用窗口，即它可以立即发送的数据量。可用窗口计算值为提供窗口大小减去在传（已发送但未得到确认）的数据值。变量 `SND.UNA` 和 `SND.WND` 分别记录窗口左边界和提供窗口值。`SND.NXT` 则记录下次发送的数据序列号，因此可用窗口值等于  $(\text{SND.UNA} + \text{SND.WND} - \text{SND.NXT})$ 。

随着时间的推移，当接收端确认数据，滑动窗口也随之右移。窗口两端的相对运动使得窗口增大或减小。可用三个术语来描述窗口左右边界的运动：

1. 关闭（close），即窗口左边界右移。当已发送数据得到 ACK 确认时，窗口会减小。
2. 打开（open），即窗口右边界右移，使得可发送数据量增大。当已确认数据得到处理，接收端可用缓存变大，窗口也随之变大。
3. 收缩（shrink），即窗口右边界左移。主机需求 RFC [RFC1122] 并不支持这一做法，但 TCP 必须能处理这一问题。15.5.3 节的糊涂窗口综合征中举了一个例子，一端试图将右边界左移使窗口收缩，但没有成功。

每个 TCP 报文段都包含 ACK 号和窗口通告信息，TCP 发送端可以据此调节窗口结构。窗口左边界不能左移，因为它控制的是已确认的 ACK 号，具有累积性，不能返回。当得到的 ACK 号增大而窗口大小保持不变时（通常如此），我们就说窗口向前“滑动”。若随着 ACK 号增大窗口却减小，则左右边界距离减小。当左右边界相等时，称之为零窗口。此时发送端不能再发送新数据。这种情况下，TCP 发送端开始探测（probe）对方窗口（参见 15.5.2 节），伺机增大提供窗口。

接收端也维护一个窗口结构，但比发送端窗口简单。该窗口结构记录了已接收并确认的数据，以及它能够接收的最大序列号。该窗口可以保证其接收数据的正确性。特别是，接收端希望避免存储重复的已接收和确认的数据，以及避免存储不应接收的数据（超过发送方右窗口边界的数据）。图 15-10 描述了接收窗口结构。

TCP 接收端滑动窗口结构帮助了解其下次应接收的数据序列号。若接收到的数据序列号在窗口内，则可以存储，否则丢弃

与发送端窗口一样，该窗口结构也包含一个左边界和右边界，但窗口内的字节（图中的 49 字节）并没有区分。对接收端来说，到达序列号小于左窗口边界（称 `RCV.NXT`），被认为是重复数据而丢弃，超过右边界（`RCV.WND + RCV.NXT`）的则超出处理范围，也被丢弃。注意到由于 TCP 的累积 ACK 结构，只有当到达数据序列号等于左边界时，数据才不会被丢弃，窗口才能向前滑动。对选择确认 TCP 来说，使用 SACK 选项，窗口内的其他报文段也可以被接收确认，但只有在接收到等于左边界的序列号数据时，窗口才能前移（SACK 的更多细节可以参见第 14 章）。

### 15.5.2 零窗口与 TCP 持续计时器

我们了解到，TCP 是通过接收端的通告窗口来实现流量控制的。通告窗口指示了接收端可接收的数据量。当窗口值变为 0 时，可以有效阻止发送端继续发送，直到窗口大小恢复为非零值。当接收端重新获得可用空间时，会给发送端传输一个窗口更新（window update），告知其可继续发送数据。这样的窗口更新通常都不包含数据（为“纯 ACK”），不能保证其传输的可靠性。因此 TCP 必须有相应措施能处理这类丢包。

如果一个包含窗口更新的 ACK 丢失，通信双方就会一直处于等待状态：接收方等待接收数据（已将窗口设为非零值），发送方等待收到窗口更新告知其可继续发送。为防止这种死锁的发生，发送端会采用一个持续计时器间歇性地查询接收端，看其窗口是否已增长。持续计时器会触发窗口探测（window probe）的传输，强制要求接收端返回 ACK（其中包含了窗口大小字段）。主机需求 RFC [RFC1122] 建议在一个 RTO 之后发送第一个窗口探测，随后以指数时间间隔发送（与第 14 章讨论过的 Kam 算法中的“第二部分”类似）。

窗口探测包含一个字节的的数据，采用 TCP 可靠传输（丢失重传），因此可以避免由窗口更新丢失导致的死锁。当 TCP 持续计时器超时，就会触发窗口探测的发送。其中包含的一个字节的数据是否能被接收，取决于接收端的可用缓存空间大小。与 TCP 重传计时器（参见第 14 章）类似，可以采用指数时间退避来计算持续计时器的超时。而不同之处在于，通常 TCP 不会停止发送窗口探测，由此可能会放弃执行重传操作。这种情况可能导致某种程度的资源耗尽，我们将在 15.7 节中讨论这一问题。

#### 例子

为了说明 TCP 的动态窗口调节和流量控制机制，我们建立了一个 TCP 连接，并使其在处理接收到的数据之前暂停接收。本实验采用了 Mac OS X 10.6 发送端和 Windows 7 接收端。在接收端运行带-P 选项的 sock 程序：

```
C: \> sock -l -s -P 20 6666
```

该命令使得接收端在处理接收到的数据前暂停 20s。这样就导致接收端的通告窗口在 125 号包处开始关闭，如图 15-11 所示。从图 15-11 中可以看到，在接收了 100 多个包后，窗口大小仍然维持在 64KB。这是由于自动窗口调节算法（参见 15.5.4 节）默认分配了 TCP 接收端的缓存。然而，随着可用缓存的减少，可以看到在 125 号包之后，窗口开始减小。随着大量的 ACK 到达，窗口进一步减小，每个到达的 ACK 号都增大 2896 字节。这表明接收端在存储这些数据，但应用程序并没有处理。如果我们进一步观察，会发现最终接收端已经没有更多空间来存储到达的数据（见图 15-12）。

从图 15-12 中可以看到，151 号包耗尽了 327 字节大小的窗口，Wireshark 显示“TCP 窗口满”（TCP Window Full）。约 200ms 后，在 4.979s 时刻，零窗口通告产生，表明无法接收新的数

| No. | Time     | Source    | Destination | Protocol | Info   |
|-----|----------|-----------|-------------|----------|--|
| 2   | 0.003022 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [SYN, ACK] Seq=0 Ack=1 win=65535  |
| 15  | 0.010130 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=2473 win=65535 L  |
| 19  | 0.010421 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=4521 win=65535 L  |
| 23  | 0.010640 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=6569 win=65535 L  |
| 27  | 0.011881 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=8617 win=65535 L  |
| 31  | 0.012137 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=10665 win=65535 L |
| 34  | 0.012929 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=12713 win=65535 L |
| 38  | 0.015507 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=15609 win=65535 L |
| 43  | 0.017863 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=18505 win=65535 L |
| 48  | 0.022081 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=21401 win=65535 L |
| 53  | 0.025970 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=24297 win=65535 L |
| 58  | 0.026315 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=27193 win=65535 L |
| 63  | 0.034158 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=30089 win=65535 L |
| 68  | 0.049115 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=32985 win=65535 L |
| 73  | 0.056894 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=35881 win=65535 L |
| 78  | 0.058797 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=38777 win=65535 L |
| 83  | 0.066892 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=41673 win=65535 L |
| 88  | 0.069709 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=44569 win=65535 L |
| 93  | 0.074032 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=47465 win=65535 L |
| 98  | 0.075499 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=50361 win=65535 L |
| 103 | 0.080786 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=53257 win=65535 L |
| 108 | 0.088841 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=56153 win=65535 L |
| 113 | 0.091330 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=59049 win=65535 L |
| 117 | 0.094739 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=61945 win=65535 L |
| 121 | 0.097035 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=64841 win=65535 L |
| 125 | 0.098536 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=67737 win=64087   |
| 127 | 0.100571 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=70633 win=61191   |
| 128 | 0.102534 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=73529 win=58295   |
| 129 | 0.107267 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=76425 win=55399   |
| 130 | 0.107578 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=79321 win=52503   |
| 131 | 0.107778 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=82217 win=49607   |
| 132 | 0.108664 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=85113 win=46711   |
| 133 | 0.109498 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=88009 win=43815   |
| 134 | 0.114609 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=90905 win=40919   |
| 135 | 0.115865 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=93801 win=38023   |
| 136 | 0.118856 | 10.0.1.37 | 10.0.1.33   | TCP      | 6666 > 53005 [ACK] Seq=1 Ack=96697 win=35127   |

图 15.4: 比较相似环境下使用 Nagle 算法与否的延

据。窗口最后的可用空间已满，接收端应用程序暂停处理数据，直到 20.143s 时刻。

收到零窗口通告后，发送端每隔 5s 共发送了三次窗口探测以查看窗口是否打开。在 20s 时刻，接收端开始处理 TCP 队列中的数据。因此有两个窗口更新传送至发送端，表明可以继续传输数据（64KB）。窗口更新并不是对新数据的确认，而只是将的日右边界有弦。这时，发送端可以恢复正常的数据传输。

随着接收端可用缓存的逐渐减小，在一段时间后，窗口开始减小。如果接收端应用程序一直不处理任何数据，且发送端持续发送，窗口最终会缩减为 0

从图 15-11 和图 15-12 中可以总结出以下几点：

1. 发送端不必传输整个窗口大小的数据。
2. 接收到返回的 ACK 的同时可将窗口右移。这是由于通告窗口是和该报文段中的 ACK 号相关的。
3. 窗口大小可能减小，如图 15-11 所示，但窗口右边界不会左移，以此避免窗口收缩。
4. 接收端不必等到窗口满才发送 ACK。

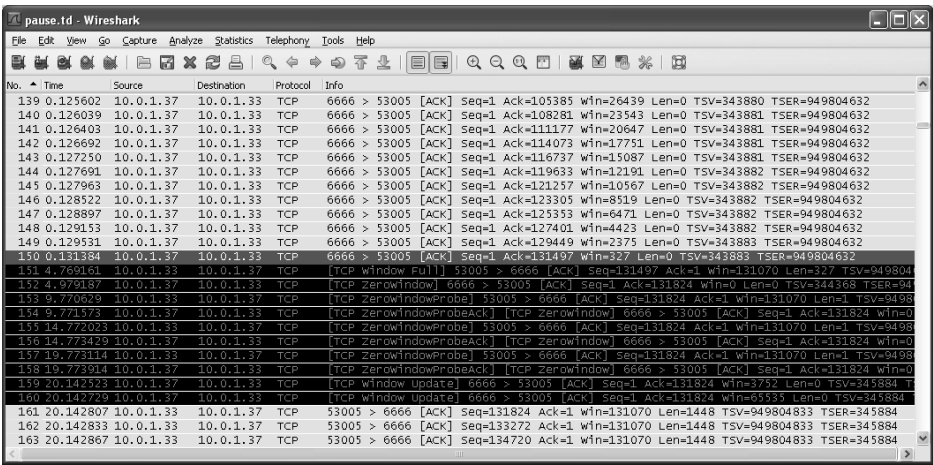


图 15.5: 接收端缓存已满。当接收应用程序再次开始处理数据时，窗口更新会通知发送端可继续发送

此外，还可通过观察吞吐量随时间的变化函数得到一些启发。使用 Wireshark 的“统计 | TCP 流图 | 吞吐量图” (Statistics | TCP Stream Graph | Throughput Graph) 功能，可以得到图 15-13 所示的时间序列。

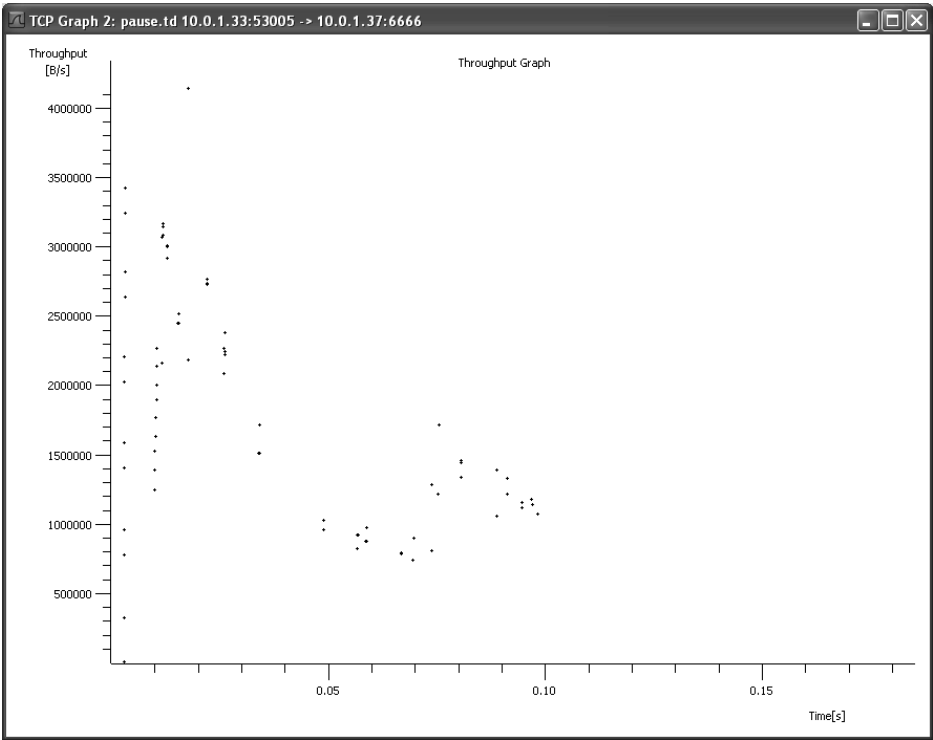


图 15.6: 使用相对较大的接收缓存，即使在接收端应用处理数据前也能传输大量的数据

这里我们看到一个有趣的现象。即使在接收端处理任何数据前，连接依然能达到约 1.3MB/s 的吞吐量。这种状况一直持续到约 0.10s 时刻。之后，直到接收端开始处理数据前（在 20s 时刻后），吞吐量基本上都为 0。

### 15.5.3 糊涂窗口综合征

基于窗口的流量控制机制，尤其是不使用大小固定的报文段的情况（如 TCP），可能会出现称为糊涂窗口综合征（Silly Window Syndrome,sws）的缺陷。当出现该问题时，交换数据段大小不是全长的而是一些较小的数据段 [RFC0813]。由于每个报文段中有效数据相对于头部信息的比例较小，因此耗费的资源也更多，相应的传输效率也更低。

TCP 连接的两端都可能导致 SWS 的出现：接收端的通告窗口较小（没有等到窗口变大才通告），或者发送端发送的数据段较小（没有等待将其他数据组合成一个更大的报文段）。要避免 SWS 问题，必须在发送端或接收端实现相关规则。TCP 无法提前预知某一端的行为。需要遵循以下规则：

- 对于接收端来说，不应通告小的窗口值。[RFC1122] 描述的接收算法中，在窗口可增至一个全长的报文段（即接收端 MSS）或者接收端缓存空间的一半（取两者中较小者）之前，不能通告比当前窗口（可能为 0）更大的窗口值。注意到可能有两种情况会用到该规则：当应用程序处理接收到的数据后使得可用缓存增大，以及 TCP 接收端需要强制返回对窗口探测的响应。
- 对于发送端来说，不应发送小的报文段，而且需由 Nagle 算法控制何时发送。为避免 SWS 问题，只有至少满足以下条件之一时才能传输报文段：
  - 全长（发送 MSS 字节）的报文段可以发送。
  - 数据段长度 接收端通告过的最大窗口值的一半的，可以发送。
  - 满足以下任一条件的都可以发送：(i) 某一 ACK 不是目前期盼的（即没有未经确认的在传数据）；(ii) 该连接禁用 Nagle 算法。

条件（a）最直接地避免了高耗费的报文段传输问题。条件（b）针对通告窗口值较小，可能小于要传输的报文段的情况。条件（c）防止 TCP 在数据需要被确认以及 Nagle 算法启用的情况下发送小报文段。若发送端应用在执行某些较小的写操作（如小于报文段大小），条件（c）可以有效避免 SWS。

上述三个条件也让我们回答了以下问题：当有未经确认的在传数据时，若使用 Nagle 算法阻止发送小的报文段，究竟多小才算小？从条件（a）可以看出，“小”意味着字节数要小于 MSS（即不超过 PMTU 或接收端 MSS 的最大包大小）。条件（b）只用于比较旧的原始主机，或者因接收端缓存有限而使用较小通告窗口时。

条件（b）要求发送端记录接收端通告窗口的最大值。发送端以此猜测接收端缓存大小。尽管当连接建立时缓存大小可能减小，但实际这种情况很少见。另外，前面也提到过，TCP 需要避免窗口收缩。



## 例子

下面我们通过一个具体的例子来观察 SWS 避免的行；本例也包含持久计时器。这里使用我们的 sock 程序，发送端主机为 Windows XP 系统，接收端为 FreeBSD，执行三次 2048 字节的写操作传输。发送端命令如下：

```
C: \> sock -l -n 3 -W 2048 10.0.0.8 6666
```

接收端相应的命令为：

```
FreeBSD& sock -i -s -P 15 -p 2  
-x 256  
-R 3000 6666
```

该命令将接收端缓存设为 3000 字节，在首次读数据前有 15s 的初始延时，之后每次读都会引入 2s 的延时，每次读的数据量为 256 字节。设置初始延时是为使接收端缓存占满，最终迫使传输停止。这时通过使接收端执行小的读操作，我们期望看到它执行 SWS 避免。利用 Wireshark 可以得到如图 15-14 所示的记录。

整个连接的传输内容如图所示。包长度是根据每个报文段中携带的 TCP 有效载荷数据描述的。在连接建立过程中，接收端通告窗口为 3000 字节，MSS 为 1460 字节。发送端在 0.052s 时刻发送了一个 1460 字节的包（包 4），在 0.053s 时刻发送了 588 字节的包（包 5）。两者总和为 2048 字节，为应用写操作的大小。包 6 是对这两个包的确认，并提供了一个 952 字节的窗口通告（ $3000-1460-588=952$ ）。

952 字节的窗口（包 6）并没有一个 MSS 大，所以 Nagle 算法阻止了发送端的立即发送。相反，发送端等待了 5s，直到持续计时器超时，才发送了一个窗口探测。考虑到无论如何都要发送一个包，因此发送端发送了允许的 952 字节数据填满了可用窗口，因此包 8 返回了零窗口通告。

下一个事件发生在 6.970s 时刻，TCP 发送了一个窗口探测，即在接收到首个零窗口通告约 25 后。探测包本身包含一个字节的的数据，图中 Wireshark 显示为“TCP Zero WindowProbe”，但对该探测包的 ACK 号却没有增大（Wireshark 将其标记为“TCP Zero WindowProbeAck”），因此这一个字节的的数据并没有被接收端保存。在 10.7828 时刻又产生了一个探测包（约 45 后），接着 18.408s 时刻又产生一个（约 8s 以后），表明其发送间隔随时间呈指数增长。注意到最后一次窗口探测中包含的一个字节的数据已被接收端确认。

在 25.061s 时刻，在上层应用执行了 6 次 256 字节的读数据操作后（每次同隔 25），窗口更新表明现在接收端缓存中有 1535 字节（ACK 号加 1）的可用空间。根据接收端 SWS 避免规则，该数位已“足够大”。发送端开始继续传送数据。在 25:0645 时刻发送了 1400 字节的包，在 25.161s 时刻得到了对 4462 字节数据的 ACK，这时通告窗口大小只有 75 字节（包 17）。该通告似乎违背了我们之前的规则，即窗口值应至少为一个 MSS（对 FreeBSD 来说）或总级存空间的

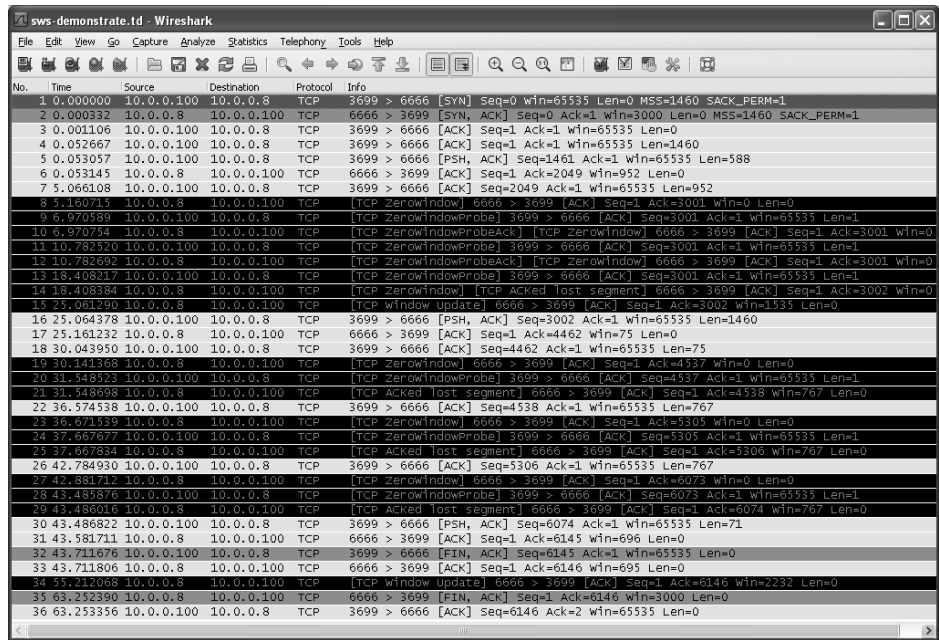


图 15.7: SWS 避免的行为分析。由于在 0.0538 时刻执行 SWS 避免，发送端没有使用通告窗口传输数据。相反，一直等到 5.066s 时刻，同时也有效地执行了窗口探测。通过 14 号包可以看到接收端 SWS 避免，即使已经处理了部分数据，接收端依然通告零窗口

四分之一。出现这种情况的原因在于避免窗口收缩。最后一个窗口更新中（包 15），接收端通告窗口右边界为  $(3002 + 1535) - 4537$ 。如果当前 ACK（包 17）通告的窗口小于 75 字节，像接收端 SWS 避免要求的那样，窗口右边界就会左移，TCP 是不允许出现这种情况的。因此这 75 字节的通告窗口代表一种更高的优先级：避免窗口收缩优先于避免 SWS。

通过包 17 和包 18 之间的 5s 的延时，我们再次看到发送端的 SWS 避免。发送端被强制要求发送一个 75 字节的包，接收端返回一个零窗口通告响应。在 1s 之后的包 20 是再次的窗口探测，得到了 767 字节的可用窗口。又一轮的发送端 SWS 避免导致了 Ss 的延时；发送端填满窗口后，再次返回零窗口通告；这种状况一直重复。最终发送端没有新的数据发送而终止。包 30 代表发送的最后一个包，在 20s 后连接终止（由于接收端应用每次读数据的间隔为 2s）。

为了理解上层应用行为、通告窗口和 SWS 避免之间的关系，我们将连接的动态传输以表格的形式展现出来。表 15-1 给出了发送端和接收端的行为，以及接收端应用执行读操作的估计时间。

SWS 避免的通告窗口及应用层动态变化情况

在表 15-1 中，第 1 列表示图中出现的每个传输行为的相对时刻，带三位小数的数值是 Wireshark 显示的时间值（参见图 15-14）。而不带小数的则是接收端主机行为的估计时刻，图中并没有显示。

接收端缓存中的数据（表中标记为“已存数据”）随着新数据的到达而增加，随着上层应用的

读取而减少。我们想了解的是接收端返回给发送端的窗口通告中包含的内容。这样就能知道接收端是怎样避免 SWS 的。

如前所述，第一次 SWS 避免是包 6 和包 7 之间的 5s 延时，由于窗口大小只有 952 字节，发送端一直避免传输直到被强制要求发送数据。传输完成后，接收端缓存满，之后产生了一系列的零窗口通告和窗口探测交换。我们可以看到持续计时器指示的时间间隔呈指数增长：探测包的发送时刻为 6.970s, 10.782s 和 18.408s。这些时刻与发送端首次接收到零窗口通告的时刻 5.160s 的间隔约为 2s、4s、8s。

尽管上层应用在 15s 和 17s 时刻读数据，但至 18.408s 时刻为止只读了 512 字节。根据接收端 SWS 避免规则要求，由于 512 字节的可用缓存既小于总缓存空间（3000 字节）的一半，也没有达到一个 MSS（1460 字节），因此不能提供窗口更新。发送端在 18.408s 时刻发送了一个窗口探测（报文段 13）。该探测包被接收，由于缓存有一定的可用空间，因此其中包含的一个字节数据也被保存，报文段 12 和 14 之间的 ACK 号的增长验证了这一点。

尽管有 511 字节的可用空间，但接收端再次实施了 SWS 避免。接收端 FreeBSD 在实现 sWS 避免时区分了何时发送窗口更新与怎样响应窗口探测。它遵循 [RFC1122] 中的规则，只在通告窗口至少为总接收缓存的一半（或一个 MSS）时才发送窗口更新，并且只有当窗口至少为一个 MSS 或超过总接收缓存的四分之一才响应窗口探测。但在这里，511 字节小于一个 MSS 且不到  $3000/4=750$  字节，因此接收端只好对报文段 13 的 ACK 中包含的通告窗口设为 0。

直到 25 时刻为止，上层应用完成了 6 次读操作，接收端缓存有 1535 字节空闲（大于总的 3000 字节的一半），因此发送了一个窗口更新（报文段 15）。发送的数据为全长报文段（报文段 16），接收到的 ACK 中包含的通告窗口仅为 75 字节。在接下来的 5s 内，两端都执行 SWS 避免。发送端需要等待一个更大的通告窗口，上层应用在 27s 时刻和 29s 时刻执行读操作，但只有 587 字节的空间，不足以发送窗口更新。因此，发送端持续等待了 5s 并最终发送了剩余的 75 字节，迫使接收端再次进入 SWS 避免状态。

接收端没有提供窗口更新，直到 31.548s 时刻发送端的持续计时器超时，发送了一个窗口探测。接收端响应了一个非零窗口，为 767 字节（大于总接收缓存的四分之一）。该窗口值对发送端并非足够大，因此继续执行发送端的 SWS 避免。发送端等待了 5s，之后一直重复上述过程。最终，在 43.486s 时刻，最后的 71 字节发送完并得到确认。该 ACK 中包含 696 字节的窗口通告。尽管小于总接收缓存的四分之一，为了避免窗口收缩，通告窗口并没有设为 0。

从报文段 32 开始，不再包含数据，连接开始关闭。随即得到的确认中窗口大小为 695 字节（接收端的 FIN 消耗了一个序列号）。在上层应用再次完成 6 次读操作后，接收端提供了一个窗口更新，但发送端已经完成所有数据的发送，并保持空闲状态。上层应用又执行了 4 次读操作，其中 3 次返回 256 字节，最后 1 次没有返回，表明已经无数据到达。此时，接收端关闭连接并发送 FIN。发送端返回了最后一个 ACK，双向连接结束。

由于发送端应用在执行 3 次 2048 字节的写操作后开始关闭连接，在发送完报文段 32 后，

发送端从 ESTABLISHED 状态变为 FIN\_WAIT\_1 状态（参见第 13 章）。接着在接收到报文段 33 后，进入 FIN\_WAIT\_2 状态。尽管在这时接收到了窗口更新，但发送端没有任何动作，因为它已经发送了 FIN 并经确认（这一阶段没有计时器）。相反，在接收到对方的 FIN 前，它只是静静等待。这就是我们没有看到更多的传输直至接收到 FIN 的原因（报文段 35）。

#### 15.5.4 大容量缓存与自动调优

从前面的章节可以看到，在相似的环境下，使用较小接收缓存的 TCP 应用的吞吐性能更差。即使接收端指定一个足够大的缓存，发送端也可能指定一个很小的缓存，最终导致性能变差。这个问题非常严重，因此很多 TCP 协议栈中上层应用不能指定接收缓存大小。在多数情况下，上层应用指定的缓存会被忽视，而由操作系统来指定一个较大的固定值或者动态变化的计算值。

在较新的 Windows 版本（Vista/7）和 Linux 中，支持接收窗口自动调优 [S98]。有了自动调优，该连接的在传数据值（连接的带宽延时积——一个重要概念，将在第 16 章讨论）需要不断被估算，通告窗口值不能小于这个值（假如剩余缓存空间足够）。这种方法使得 TCP 达到其最大可用吞吐率（受限于网络可用容量），而不必提前在发送端或接收端设置过大的缓存。在 Windows 系统中，默认自动设置接收端缓存大小。然而，也可以通过 netsh 命令更改默认值：

```
C: \> netsh interface tcp
set heuristics disabled
C:\> netsh interface tcp set global autotuninglevel=x
```

这里 X 可设置为 disabled、highlyrestricted、restricted、normal 或 experimental。不同的设置值会影响接收端通告窗口的自动选择。在 disabled 状态下，禁用自动调优，窗口大小使用默认值。restricted 模式限制窗口增长，normal 允许其相对快速增长。而 experimental 模式允许窗口积极增长，但通常并不推荐 normal 模式，因为许多因特网站点及某些防火墙会干扰，或没有很好地实现 TCP 窗口缩放（Window Scale）选项。

对于 Linux 2.4 及以后的版本，支持发送端自动调优。2.6.7 及之后的版本，两端都支持该功能。然而，自动调优受制于缓存大小。下面的 Linux sysctl 变量控制发送端和接收端的最大缓存。等号之后的值为默认值（根据不同的 Linux 版本可能会有不同），如果系统用于高带宽延时积的环境下，上述值需要增大。

```
net.core.rmem_max = 131071
net.core.wmem_max = 131071
net.core.rmem_default = 110592
net.core.wmem_default = 110592
```

另外，通过下面的变量设定自动调优参数：

```
net.ipv4.tcp_rmem = 4096 87380 174760
net.ipv4.tcp_wmem = 4096 16384
131072
```

每个变量包含三个值：自动调优使用的缓存的最小值、默认值和最大值。

## 例子

为演示接收端自动调优行为，这里采用 Windows XP 发送端（设为使用大容量窗口和窗口缩放）和 Linux 2.6.11 接收端（支持自动调优）。在发送端运行如下命令：

```
C: \> gock -n 512 -l 10.0.0.1 6666
```

对接收端，我们不对接收缓存做任何设置，但在上层应用读取数据前设置 20s 的初始延时：

```
Linux8 sock -l -g -v - 20 6666
```

为描述接收端通告窗口的增长，可以利用 Wireshark 来显示包传输情况，并根据接收端地址来进行分类（参见图 15-15）。在连接建立阶段，接收端初始窗口值为 1460 字节，初始 MSS 为 1412 字节。由于其采用了窗口缩放，会有 2 倍的变动（图中没有显示），使得最大可用窗口为 256KB。可以看到在完成第一个包传输后，窗口有了增长，相应地发送端也提高了发送速率。在第 16 章中我们将会讨论 TCP 拥塞控制的发送速率。现在，我们只需要知道发送端何时开始发送，通常情况下其首先发送一个包，接着每收到一个 ACK，发包数就增加一个 MSS。因此，每接收一个 ACK，就会发送两个包（每个包长度为一个 MSS）。

观察窗口通告值 10712、13536、16360、19184 ... 可以发现，每接收一个 ACK，窗口就增长两个 MSS，这与发送端拥塞控制操作相一致（第 16 章会讨论）。假设接收端存储空间足够大，根据拥塞控制局限性，通告窗口总是大于允许发送的数据量。这种方式是最优的——在保持发送端最大发送速率的情况下，接收端通告和使用的缓存空间最小。

当接收端缓存资源耗尽时，自动调优也会受影响。在本例中，0.678s 时刻窗口达到最大值 33304 字节，接着开始减小。这是由于上层应用停止读取数据，导致缓存被占满。当 20s 后上层应用继续读操作时，窗口再次增大，并超过了之前的最大值（参见图 15-16）。

零窗口通告（包 117）使得发送端执行了一系列的窗口探测，但返回的仍是一系列的零窗口。在 20.043s 时刻恢复读数据操作时，发送端接收到了窗口更新。每接收到一个 ACK，窗口就增大两个 MSS。随着数据的发送、接收和处理，通告窗口达到了最大值 67808。该版本的 Linux 也测量相邻两次读操作完成的时间，并与估计 RTT 值相比较。如果 RTT 估计值增大，那么缓存也将增大（但不会因 RTT 的减小而减小缓存）。这样即使在连接的带宽延时积增大的情况下，自动调优也保持接收端通告窗口优先于发送端窗口。

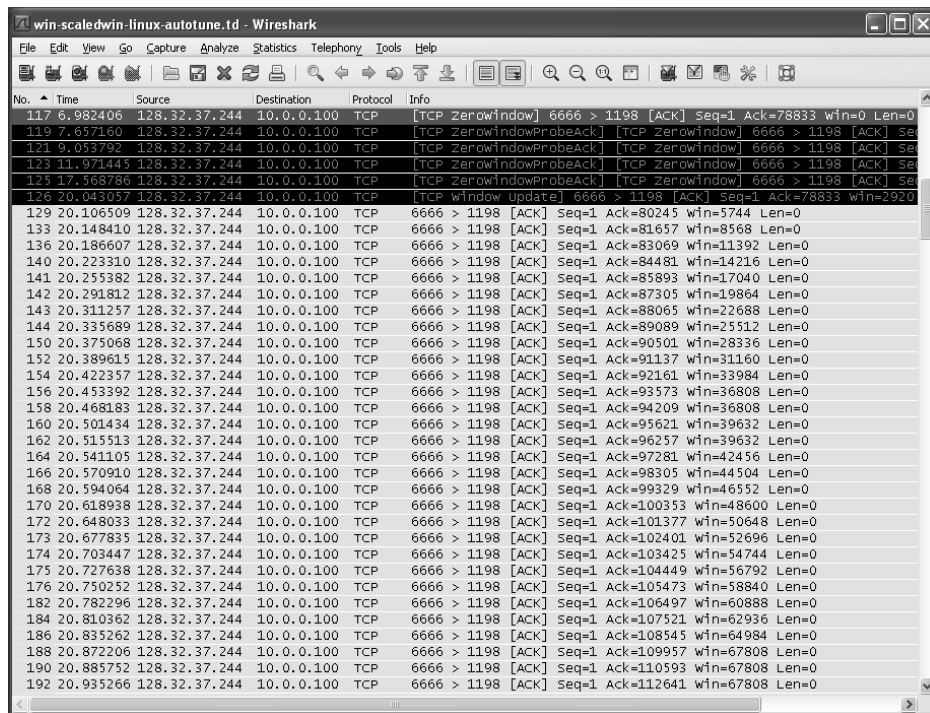
| No. | Time      | Source        | Destination | Protocol | Info   |
|-----|-----------|---------------|-------------|----------|--|
| 2   | 0.014114  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [SYN, ACK] Seq=0 Ack=1 win=1460 Len=0 MSS=1412 SACK_PERM |
| 5   | 0.078100  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=1025 win=7888 Len=0                      |
| 8   | 0.136991  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=2437 win=10712 Len=0                     |
| 11  | 0.172750  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=3849 win=13536 Len=0                     |
| 14  | 0.214482  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=5261 win=16360 Len=0                     |
| 17  | 0.249142  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=6673 win=19184 Len=0                     |
| 20  | 0.284069  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=8085 win=22008 Len=0                     |
| 22  | 0.325937  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=9497 win=24832 Len=0                     |
| 26  | 0.356503  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=10909 win=27656 Len=0                    |
| 27  | 0.366867  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=11265 win=27656 Len=0                    |
| 29  | 0.399342  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=12677 win=30480 Len=0                    |
| 31  | 0.439019  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=14337 win=33304 Len=0                    |
| 35  | 0.498645  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=16773 win=33304 Len=0                    |
| 38  | 0.543485  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=18433 win=33304 Len=0                    |
| 41  | 0.599412  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=20481 win=33304 Len=0                    |
| 44  | 0.678770  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=23941 win=29844 Len=0                    |
| 47  | 0.732456  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=25989 win=27796 Len=0                    |
| 51  | 0.784443  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=28037 win=25748 Len=0                    |
| 54  | 0.842815  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=30085 win=23700 Len=0                    |
| 57  | 0.900957  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=31745 win=22040 Len=0                    |
| 60  | 0.982327  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=33841 win=17944 Len=0                    |
| 65  | 1.070205  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=39301 win=14484 Len=0                    |
| 68  | 1.124690  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=41349 win=12436 Len=0                    |
| 72  | 1.140705  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=41985 win=11800 Len=0                    |
| 73  | 1.179343  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=43397 win=10388 Len=0                    |
| 75  | 1.198797  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=44033 win=11392 Len=0                    |
| 77  | 1.239476  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=45445 win=14216 Len=0                    |
| 79  | 1.253527  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=46081 win=17040 Len=0                    |
| 81  | 1.275912  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=47105 win=19864 Len=0                    |
| 83  | 1.359920  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=50177 win=16792 Len=0                    |
| 87  | 1.444195  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=53249 win=13720 Len=0                    |
| 91  | 1.521311  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=57345 win=9624 Len=0                     |
| 97  | 1.597698  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=60417 win=6552 Len=0                     |
| 99  | 1.622798  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=61441 win=5744 Len=0                     |
| 100 | 1.659010  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=62853 win=8568 Len=0                     |
| 103 | 1.674557  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=63489 win=11340 Len=0                    |
| 105 | 1.762732  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=66949 win=7880 Len=0                     |
| 109 | 1.860567  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=70657 win=4172 Len=0                     |
| 111 | 1.916495  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=73093 win=5740 Len=0                     |
| 114 | 1.989704  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=74753 win=4080 Len=0                     |
| 115 | 2.085549  | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=77577 win=1256 Len=0                     |
| 117 | 6.982406  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP ZeroWindow] 6666 > 1198 [ACK] Seq=1 Ack=78833 win=0 Len=0       |
| 119 | 7.657160  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP ZeroWindowProbeAck] [TCP ZeroWindow] 6666 > 1198 [ACK] Seq=1 A  |
| 121 | 9.053792  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP ZeroWindowProbeAck] [TCP ZeroWindow] 6666 > 1198 [ACK] Seq=1 A  |
| 123 | 11.971445 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP ZeroWindowProbeAck] [TCP ZeroWindow] 6666 > 1198 [ACK] Seq=1 A  |
| 125 | 17.568786 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP ZeroWindowProbeAck] [TCP ZeroWindow] 6666 > 1198 [ACK] Seq=1 A  |
| 126 | 20.043057 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP window Update] 6666 > 1198 [ACK] Seq=1 Ack=78833 win=2920 Len=0 |

图 15.8: Linux 接收端执行自动调优, 窗口值随着接收数据的增多而增大。由于上层应用在 20s 之内都没有读取数据, 最终窗口将关闭

随着广域网络连接速度的增长, TCP 应用使用的缓存太小已成为严重局限。在美国, 全国范围内的 RTT 约为 100ms, 在一个 1Gb/s 的网络中使用 64KB 的窗口将 TCP 吞吐量限制在约 640KB/s, 而计算的最大值可以达到约 130MB/s (99% 的带宽都浪费了)。实际上, 如果在相同网络环境下使用较大容量的缓存, 吞吐性能将提升 100 倍。Web100 工程 [W100] 应得到更多关注和信心。它开发了一系列工具和改进软件, 致力于使应用从众多的 TCP 实现中获得最优的吞吐性能。

## 15.6 紧急机制

我们在第 12 章中已经提到, TCP 头部有一个位字段 URG 用来指示“紧急数据”。应用在执行写操作时, 可通过设置 Berkeley 套接字 API (MSG\_OOB) 的特殊选项将数据标记为紧急, 但 [RFC6093] 不再推荐设置紧急数据。当发送端 TCP 收到这类写操作要求时, 会进入称为紧急模式 (urgent mode) 的特殊状态。它记录紧急数据的最后一个字节, 用于设置紧急指针 (Urgent Pointer) 字段, 随后发送端生成的每个 TCP 头部都包含该字段, 直到应用停止紧急数据写操



The image shows a Wireshark packet capture window titled "win-scaledwin-linux-autotune.td - Wireshark". The packet list pane displays a series of TCP packets. Packets 117 through 126 show a sequence of "TCP zerowindow" and "TCP zerowindowProbsAck" events, indicating a zero window state. Packet 127 is a "TCP Window Update" from 128.32.37.244 to 10.0.0.100, with Seq=1, Ack=78833, and Win=2920. Subsequent packets (129-192) show a normal flow of data with "Seq=1" and increasing "Ack" values, indicating successful data reception and acknowledgment.

| No. | Time      | Source        | Destination | Protocol | Info   |
|-----|-----------|---------------|-------------|----------|--|
| 117 | 6.982406  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP zerowindow] 6666 > 1198 [ACK] Seq=1 Ack=78833 win=0 Len=0 |
| 119 | 7.657160  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP zerowindowProbsAck] 6666 > 1198 [ACK] Seq=1               |
| 121 | 9.053792  | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP zerowindowProbsAck] 6666 > 1198 [ACK] Seq=1               |
| 123 | 11.971445 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP zerowindowProbsAck] 6666 > 1198 [ACK] Seq=1               |
| 125 | 17.568786 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP zerowindowProbsAck] 6666 > 1198 [ACK] Seq=1               |
| 126 | 20.043057 | 128.32.37.244 | 10.0.0.100  | TCP      | [TCP Window Update] 6666 > 1198 [ACK] Seq=1 Ack=78833 win=2920 |
| 129 | 20.106509 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=80245 win=5744 Len=0               |
| 133 | 20.148410 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=81657 win=8568 Len=0               |
| 136 | 20.186607 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=83069 win=11392 Len=0              |
| 140 | 20.223310 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=84481 win=14216 Len=0              |
| 141 | 20.255382 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=85893 win=17040 Len=0              |
| 142 | 20.291812 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=87305 win=19864 Len=0              |
| 143 | 20.311257 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=88065 win=22688 Len=0              |
| 144 | 20.335689 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=89089 win=25512 Len=0              |
| 150 | 20.375068 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=90501 win=28336 Len=0              |
| 152 | 20.389615 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=91137 win=31160 Len=0              |
| 154 | 20.422357 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=92161 win=33984 Len=0              |
| 156 | 20.453392 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=93573 win=36808 Len=0              |
| 158 | 20.468183 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=94209 win=36808 Len=0              |
| 160 | 20.501434 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=95621 win=39632 Len=0              |
| 162 | 20.515113 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=96257 win=39632 Len=0              |
| 164 | 20.541105 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=97281 win=42456 Len=0              |
| 166 | 20.570910 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=98305 win=44504 Len=0              |
| 168 | 20.594064 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=99329 win=46552 Len=0              |
| 170 | 20.618938 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=100353 win=48600 Len=0             |
| 172 | 20.648033 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=101377 win=50648 Len=0             |
| 173 | 20.677835 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=102401 win=52696 Len=0             |
| 174 | 20.703447 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=103425 win=54744 Len=0             |
| 175 | 20.727638 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=104449 win=56792 Len=0             |
| 176 | 20.750252 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=105473 win=58840 Len=0             |
| 182 | 20.782296 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=106497 win=60888 Len=0             |
| 184 | 20.810362 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=107521 win=62936 Len=0             |
| 186 | 20.835262 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=108545 win=64984 Len=0             |
| 188 | 20.872206 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=109957 win=67808 Len=0             |
| 190 | 20.885752 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=110593 win=67808 Len=0             |
| 192 | 20.935266 | 128.32.37.244 | 10.0.0.100  | TCP      | 6666 > 1198 [ACK] Seq=1 Ack=112641 win=67808 Len=0             |

图 15.9: 当上层应用暂停读取数据时, 接收端缓冲区开始被占满, 自动调优也相应停止。随着读操作的继续执行, 通告窗口也逐渐增大, 并超过了之前的最大值

作, 并且所有序列号在紧急指针之前的数据都经接收端确认。根据 [RFC6093], 紧急指针指示的是紧急数据之后的一个字节。大量的 RFC 文档中对紧急指针的阐述都存在语义上的模糊和二义性。对于使用 IPv6 的超长数据报而言, 紧急指针值需设为 65535, 用于指示紧急数据的末端位于 TCP 数据域的最后 [RFC2675], 如果使用传统的 16 位紧急指针字段就不能表示 64KB 的偏移。

当收到 URG 置位的报文段时, TCP 接收端就会进入紧急模式。接收端应用可以调用标准套接字 API (select O) 来判断是否进入紧急模式。紧急机制会带来操作上的混淆, 因为 Berkeley 套接字 API 和文档中用到了术语: 带外 (Out-Of-Band, OOB) 数据。而实际上 TCP 并没有实现任何 OOB 功能。相反, 差不多所有 TCP 实现在将紧急数据的最后一个字节传输给上层应用时, 在接收端使用了一个截然不同的 API 参数。接收端必须要设置 MSG\_OOB 选项检索该字节, 或者设置 MSG\_OOBINLINE 使该字节保持在正常数据流传输 (在使用紧急机制情况下, 需要用到该方法)。

### 15.6.1 例子

为更好地理解紧急机制, 我们通过一个例子来具体观察紧急模式的行为, 包括在零窗口事件期间发生的状况。这里使用 Mac OS X 发送端和 Linux 接收端。为获得零窗口, 我们首先在接

收端限制接收窗口自动调优:

```
Linux# sysctl -w net.ipv4.tcp_rmem='4096 4096 174760'
```

```
Linux& sock -1 -v - -D 1 -P 10 5555
```

第一个命令确保接收窗口的自动调整幅度不超过 4KB，这样就可以清楚地看到窗口关闭时发生的情形。第二个命令使服务器在读数据前等待 10s，并在每次读操作间等待 1s。在客户端我们执行如下命令：

```
Mac& sock -1 -n 7 -O 7 -D1 -8 8192 10.0.1.1
SO_SNDBUF
5555
- 8192
connected on 10.0.1.33.51101 to 10.0.1.1.5555
TCP_MAXSEG = 1448
wrote 1024 bytes wrote 1024 bytes wrote 1024 bytes wrote 1024 bytes wrote 1024 bytes wrote
wrote 1 byte of urgent data
wrote 1024 bytes
```

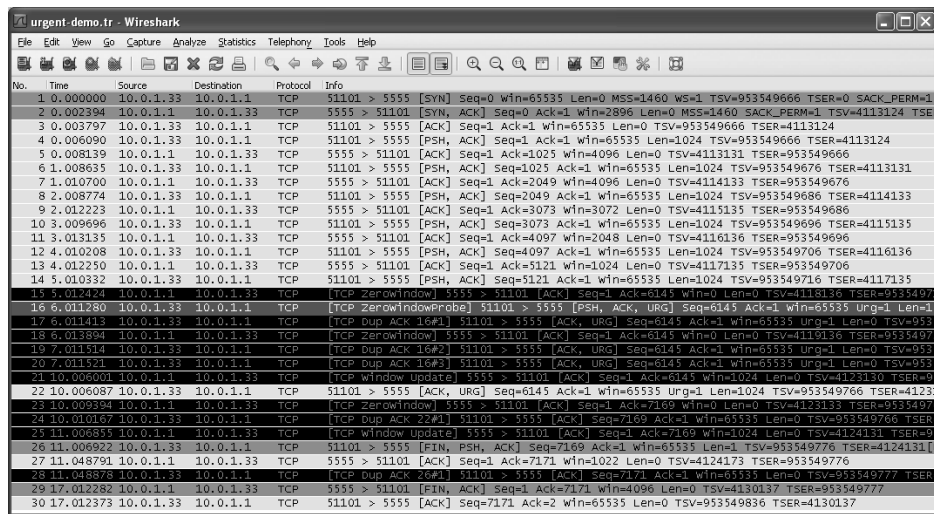
该命令使得客户端每隔 1s 执行一次写操作，共 7 次，每次 1024 字节，且在最后一次写之前写了 1 个字节的紧急数据。客户端缓存设置为 8192 字节，由于在 TCP 发送数据前所有数据都暂时存储在发送端，因此缓存已足够大，该应用可立即得到执行。

如图 15-17 所示，接收端初始通告窗口右边界为 2800，并很快增至 5121。在 1.0s 时刻，应用执行了一次写操作，窗口有边界前进至 6145。之后由于自动调优在高于 4192 字节时被禁用且接收应用没有执行读操作，窗口没有继续增长。直到 10.0s 时刻，发送端执行了窗口探测，但依旧没有获得窗口增长。最终在 10.0s 时刻之后，接收端开始继续读取数据，窗口打开，发送端继续发送直至完成传输。包交换情况如图 15-18 所示。

紧急模式的“出口点”定义为 TCP 报文段中序列号字段与紧急指针字段之和。每个 TCP 连接只维护一个紧急“点”（序列号偏秘），因此紧急指针字段为空的包会导致前面的紧急指针包含的信息丢失。报文段 16 为第一个包含有效紧急指针的报文段，使得序列号 6146 到达出口点。注意到该序列号可能并不在指示的报文段中，而可能在之后的报文段中。例如报文段 17 就是这种情况，它没有包含任何数据，只有紧急指针（值为 1）。

如前所述，有个问题一直存在争议，即出口点指示的是紧急数据的最后一个字节还是非紧急数据的第一个字节。TRFC122] 认为指针指示的是紧急数据的最后一个字节。然而，基本上所有的 TCP 实现都没有遵循该规定，因此 [RFC6093] 认识到了这一问题，并修改规范让指针指向非紧急数据的第一个字节。在本例中，序列号为 6145 的字节包含由 sock 客户端产生的 1 字节紧急数据，但在所有的报文段中，紧急指针的值为 1，序列号为 6145。因此，可以看出该 TCP





| No. | Time      | Source    | Destination | Protocol | Info   |
|-----|-----------|-----------|-------------|----------|--|
| 1   | 0.000000  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [SYN, Seq=0 Win=65535 Len=0 MSS=1460 WS=1 TSV=953549666 TSER=0 SACK_PERM=1    |
| 2   | 0.002394  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSV=4113124 TSER= |
| 3   | 0.003797  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [ACK] Seq=1 Ack=1 Win=65535 Len=0 TSV=953549666 TSER=4113124                  |
| 4   | 0.006090  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=1 Ack=1 Win=65535 Len=1024 TSV=953549666 TSER=4113124          |
| 5   | 0.008139  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=1025 Win=4096 Len=0 TSV=4113131 TSER=953549666                |
| 6   | 1.008835  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=1025 Ack=1 Win=65535 Len=1024 TSV=953549676 TSER=4113131       |
| 7   | 1.010700  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=2049 Win=4096 Len=0 TSV=4114133 TSER=953549676                |
| 8   | 2.008774  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=2049 Ack=1 Win=65535 Len=1024 TSV=953549686 TSER=4114133       |
| 9   | 2.012223  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=3073 Win=3072 Len=0 TSV=4115135 TSER=953549686                |
| 10  | 3.009696  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=3073 Ack=1 Win=65535 Len=1024 TSV=953549696 TSER=4115135       |
| 11  | 3.013135  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=4097 Win=2048 Len=0 TSV=4116136 TSER=953549696                |
| 12  | 4.010208  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=4097 Ack=1 Win=65535 Len=1024 TSV=953549706 TSER=4116136       |
| 13  | 4.012250  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=5121 Win=1024 Len=0 TSV=4117135 TSER=953549706                |
| 14  | 5.010532  | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [PSH, ACK] Seq=5121 Ack=1 Win=65535 Len=1024 TSV=953549716 TSER=4117135       |
| 15  | 6.012424  | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=6145 Win=65535 Len=0 TSV=4118136 TSER=95354977                |
| 16  | 6.011280  | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP ZeroWindowProbe] 51101 > 5555 [PSH, ACK, URG] Seq=6145 Ack=1 Win=65535 Urg=1 Len=1    |
| 17  | 6.011413  | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP Dup ACK 16#1] 51101 > 5555 [ACK, URG] Seq=6145 Ack=1 Win=65535 Urg=1 Len=0 TSV=953    |
| 18  | 6.013894  | 10.0.1.1  | 10.0.1.33   | TCP      | [TCP ZeroWindow] 5555 > 51101 [ACK] Seq=1 Ack=6145 Win=0 Len=0 TSV=4119136 TSER=95354972   |
| 19  | 7.011514  | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP Dup ACK 16#2] 51101 > 5555 [ACK, URG] Seq=6145 Ack=1 Win=65535 Urg=1 Len=0 TSV=953    |
| 20  | 7.011521  | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP Dup ACK 16#3] 51101 > 5555 [ACK, URG] Seq=6145 Ack=1 Win=65535 Urg=1 Len=0 TSV=953    |
| 21  | 10.006001 | 10.0.1.1  | 10.0.1.33   | TCP      | [TCP window update] 5555 > 51101 [ACK] Seq=1 Ack=6145 Win=1024 Len=0 TSV=4121310 TSER=95   |
| 22  | 10.006087 | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [ACK, URG] Seq=6145 Ack=1 Win=65535 Urg=1 Len=1024 TSV=953549766 TSER=4123    |
| 23  | 10.009394 | 10.0.1.1  | 10.0.1.33   | TCP      | [TCP ZeroWindow] 5555 > 51101 [ACK] Seq=1 Ack=7169 Win=0 Len=0 TSV=4123135 TSER=95354976   |
| 24  | 10.010157 | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP Dup ACK 24#1] 51101 > 5555 [ACK] Seq=7169 Ack=1 Win=65535 Len=0 TSV=95354976 TSER=    |
| 25  | 11.000855 | 10.0.1.1  | 10.0.1.33   | TCP      | [TCP window update] 5555 > 51101 [ACK] Seq=1 Ack=7169 Win=1024 Len=0 TSV=4124131 TSER=95   |
| 26  | 11.006922 | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [FIN, PSH, ACK] Seq=7169 Ack=1 Win=65535 Len=1 TSV=953549776 TSER=4124131 [   |
| 27  | 11.048791 | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [ACK] Seq=1 Ack=7171 Win=1022 Len=0 TSV=4124173 TSER=953549776                |
| 28  | 11.048878 | 10.0.1.33 | 10.0.1.1    | TCP      | [TCP Dup ACK 26#1] 51101 > 5555 [ACK] Seq=7171 Ack=1 Win=65535 Len=0 TSV=953549777 TSER=   |
| 29  | 17.012282 | 10.0.1.1  | 10.0.1.33   | TCP      | 5555 > 51101 [FIN, ACK] Seq=1 Ack=7171 Win=4096 Len=0 TSV=4130137 TSER=953549777           |
| 30  | 17.012373 | 10.0.1.33 | 10.0.1.1    | TCP      | 51101 > 5555 [ACK] Seq=7171 Ack=2 Win=65535 Len=0 TSV=953549836 TSER=4130137               |

图 15.10: 整个数据传输在 5.0125 时刻出现了一次零窗口通告。当应用执行下一次写操作时, 发送端 TCP 进入紧急模式, 6.0113s 时刻发送的窗口探测报文段中的 URG 置位。在第 7 秒执行最后一次写操作并关闭, 产生了两个空的报文段。10.006s 时刻的窗口更新重新启动数据传输。10.009s 时刻的零窗口通告使得传输再次停止, 同时由于紧急指针已被确认, 因此可以退出紧急模式。11.007s 时刻的 FIN 包含了数据的最后一个字节

实现中 (大多数 TCP 实现都如此), 出口点为非紧急数据的第一个字节的序列号。从这个例子可以看到, TCP 将紧急数据携带在数据流中传输 (而非“带外传输”)。如果某个应用确实需要独立的信号通道, 可以简单采用另一个 TCP 连接。(某些传输层协议确实提供大多数人认为的 OOB 数据, 即像通常数据链路那样使用同一个连接, 但有独立的逻辑数据路径。TCP 并不提供该功能。)

## 15.7 与窗口管理相关的攻击

TCP 窗口管理可能受到多种攻击, 主要形式为资源耗尽。通告窗口较小会使得 TCP 传输减慢, 因此会更长时间地占用资源, 如存储空间。这一点已被用于针对传输性能较差的网络攻击 (即蠕虫)。例如, LaBrea tarpit 程序 [LO1] 在完成 TCP 三次握手后, 要么不做任何行为, 要么只产生一些最小的应答, 使得发送速率不断减慢。通过此法来保持发送端忙碌, 本质上是减慢蠕虫的传播速度。因此 tarpit 程序是针对流量攻击的一类攻击。

比较新的攻击发布于 2009 年 [109], 它基于已知的持续计时器的缺陷, 采用客户端多“SYN cookies”技术 (参见第 13 章)。所有必要的连接状态都可以下载到受害主机进行, 从而使得攻击方主机消耗最少的资源。这种攻击本身类似于 LaBrea 思想, 只是其针对持续计时器。同一台服务器可能受到多个此类攻击, 最终导致资源耗尽 (如系统内存耗尽)。[C723308] 提出了一种解决方法, 即当推断出现资源耗尽时, 允许其他进程关闭 TCP 连接。

## 15.8 总结

交互式数据传输的报文段通常小于 SMSS。接收方收到这些分组时可能会采取延时确认的方法，希望能将这些 ACK 与需要发送给对方的数据一起捎带传输。这种方法可以减少传输报文段的数目，特别是在交互式流量传输中，服务器需要对客户端的每个按键都返回响应。然而，延时确认也会引入额外的延时。

对于 RTT 相对较大的连接，如 WAN，通常使用 Nagle 算法来减少较小报文段数目。该算法限制发送端在任意时刻发送单个小数据包。这样会减少较小数据包在网络连接中的数目，从而减小传输资源开销，但同时也可能引入上层应用无法接受的延时。另外，延时 ACK 与 Nagle 算法的互相作用可能导致短暂的死锁。基于上述原因，有的应用可能禁用 Nagle 算法，而大多数交互式应用都使用该功能。

TCP 通过在其发送的每个 ACK 中包含一个窗口通告来实现流量控制。该窗口告诉对方自己还有多少缓存空间。当没有使用 TCP 窗口缩放选项时，最大通告窗口为 65535 字节。否则最大窗口值可以更大（约 1GB）。

通告窗口值可能为 0，表明接收端缓存已满。这时发送端停止发送，并以一定间隔不断地发送窗口探测，发送间隔类似于超时重传（参见第 14 章），直到收到 ACK 表明窗口变大，或收到接收端主动发送的窗口通告（窗口更新）表明有可用缓存空间。这种以一定间隔连续发送的行可能被用于资源耗尽攻击。

随着 TCP 的不断发展，出现了一种奇怪的现象。当通告窗口较小时，发送端会立即发送数据填满该窗口，这样在连接中就会出现大量高耗费的小数据包。这种现象被称为“糊涂窗口综合征”。针对这一问题，在 TCP 发送端和接收端都有相应的策略。对发送端来说，若通告窗口较小则避免发送小数据包；接收端则尽量避免通告小窗口。

接收端窗口大小受限于其缓存大小。一般来说，如果上层应用没有设置其接收缓存大小，就会为其分配一个相对较小的空间，这样即使在高带宽的传输路径上，传输延时仍旧很大，导致网络吞吐性能变差。在较新的操作系统中不会出现上述问题，采用自动调优的方法可以高效地自动分配缓存大小。

# TCP 拥塞控制

---

## 16.1 引言

本章将探讨 TCP 实现拥塞控制的方法，这也是批量数据传输中最重要的。拥塞控制是 TCP 通信的每一方需要执行的一系列行为。这些行为由特定算法规定，用于防止网络因为大规模的通信负载而瘫痪。其基本方法是当有理由认为网络即将进入拥塞状态（或者已经由于拥塞而出现路由器丢包情况）时减缓 TCP 传输。TCP 拥塞控制的难点在于怎样准确地判断何时需要减缓且如何减缓 TCP 传输，以及何时恢复其原有的速度。

TCP 是提供系统间数据可靠传输服务的协议。第 15 章已经提到，当 TCP 通信的接收方的接收速度无法匹配发送速度时，发送方会降低发送速度。TCP 的流量控制机制完成了对发送速率的调节，它是基于 ACK 数据包中的通告窗口大小字段来实现的。这种方式提供了明确的接收方返回的状态信息，避免接收方缓存溢出。

当网络中大量的发送方和接收方被要求承担超负荷的通信任务时，可以考虑采取降低发送速率或者最终丢弃部分数据（也可将两者结合使用）的方法。这是将排队理论应用于路由器的基本观测结果：即使路由器能够存储一些数据，但若源源不断的数据到达速率高于发出速率，任何容量的中间存储都会溢出。简言之，当某一路由器在单位时间内接收到的数据量多于其可发送的数据量时，它就需要把多余的部分存储起来。假如这种状况持续，最终存储资源将会耗尽，路由器因此只能丢弃部分数据。

路由器因无法处理高速率到达的流量而被迫丢弃数据信息的现象称为拥塞。当路由器处于上述状态时，我们就说出现了拥塞。即使仅有一条通信连接，也可能造成一个甚至多个路由器拥塞。若不采取对策，网络性能将大受影响以致瘫痪。在最坏情况下，甚至形成拥塞崩溃。为避免或者在一定程度上缓解这种状况，TCP 通信的每一方实行拥塞控制机制。不同的 TCP 版本（包括运行 TCP/IP 协议栈的操作系统）采取的规程和行为有所差异。本章将着重讨论最常用的方

法。

### 16.1.1 TCP 拥塞检测

如前所述, 针对丢包情况, TCP 采取的首要机制是重传, 包括超时重传和快速重传 (参见第 14 章)。考虑如下情形, 当网络处于拥塞崩溃状态时, 共用一条网络传输路径的多个 TCP 连接却需要重传更多的数据包。这就好比火上浇油, 可想而知, 结果只会更糟, 所以这种情况应该尽量避免。

当拥塞状况出现 (或将要出现) 时, 我们可以减缓 TCP 发送端的发送速率; 若拥塞情况有所缓解, 可以检测和使用新的可用带宽。然而这在互联网中却很难做到, 因为对于 TCP 发送方来说, 没有一个精确的方法去知晓中间路由器的状态。换言之, 没有一个明确的信号告知拥塞状况已发生。典型的 TCP 只有在断定拥塞发生的情况下, 才会采取相应的行动。推断是否出现拥塞, 通常看是否有丢包情况发生。在 TCP 中, 丢包也被用作判断拥塞发生与否的指标, 用来衡量是否实施相应的响应措施 (即以某种方式减级发送)。从 20 世纪 80 年代起, TCP 就一直沿用这种方法。其他拥塞探测方法, 包括时延测量和显式拥塞通知 (ECN, 16.11 节会讨论), 使得 TCP 能在丢包发生前检测拥塞。在学习一些“经典”算法后, 我们将讨论上述探测方法。

在当今的有线网络中, 出现在路由器或交换机中的拥塞是造成丢包的主要原因。而在无线网络中, 传输和接收错误是导致丢包的重要因素。从 20 世纪 90 年代中期无线网络获得广泛应用开始, 判断丢包是由于拥塞引起还是传输错误引起, 一直是研究的热点问题。

在第 14 章中, 我们已经看到 TCP 如何利用计时器、确认以及选择确认机制来检测丢包和恢复传输。当有丢包情况出现时, TCP 的任务是重传这些数据包。现在我们关心的是, 当观测到丢包后, TCP 还做了哪些工作, 特别是它如何识别这就是已出现拥塞的信号, 进而需要执行减速操作。下面的章节主要讨论 TCP 何时减速以及怎样减速 (包括如何恢复传输速度)。我们首先介绍 TCP 在建立新连接时如何确立基本数据传输速率, 以及稳定执行大数据量传输操作的经典算法。另外, 我们也整合了近年来对这些算法的研究和改进成果, 并细查了相关扩展资料。在此基础上, 我们讨论总结了 TCP 拥塞控制安全和其他相关问题。拥塞控制是网络研究领域的热点 [RFC6077], 每年都会有相关论文发表。

### 16.1.2 减缓 TCP 发送

一个亟待解决的问题是, 如何减缓 TCP 发送。在第 15 章已经提到, 根据接收方剩余缓存空间大小, 在 TCP 头部设置了通知窗口大小字段, 该数值是 TCP 发送方调节发送速率的依据。进一步说, 当接收速率或网络传输速率过慢时, 我们需要降低发送速率。实现上述操作, 基于对

网络传输能力的估计，可以在发送端引入一个窗口控制变量，确保发送窗口大小不超过接收端接收能力和网络传输能力，即 TCP 发送端的发送速率等于接收速率和传输速率两者中较小值。

反映网络传输能力的变量称为拥塞窗口（congestion window），记作  $cwnd$ 。因此，发送端实际（可用）窗口  $W$  就是接收端通知窗口  $awnd$  和拥塞窗口  $cwnd$  的较小者：

$$W = \min(cwnd, awnd) \quad (16.1)$$

根据上述等式，TCP 发送端发送的数据中，还没有收到 ACK 回复的数据量不能多于  $P$ （以包或字节为单位）。这种已经发出但还未经确认的数据量大小有时称为在外数据值（flight size），它总是小于等于  $W$ 。通常， $W$  可以以包或字节为单位。

当 TCP 不使用选择确认机制时， $W$  的限制作用体现为，发送方发送的报文段序列号不能大于 ACK 号的最大值与  $W$  之和。而对采用选择确认的发送方则有所不同， $W$  被用来限制在外数据值。

这看似合乎逻辑，但实际并非如此。因为网络和接收端状况会随时间变化，相应地， $awnd$  和  $cwnd$  的数值也会随之改变。另外，由于缺少显示拥塞的明确信号（参见前述章节），TCP 发送方无法直接获得  $cwnd$  的“准确”值。因此，变量  $W$ 、 $cwnd$ 、 $awnd$  的值都要根据经验设定并需动态调节。此外，如前所述， $P$  的值不能过大或过小——我们希望其接近带宽延迟积（Bandwidth-Delay Product, BDP），也称作最佳窗口大小（optimal window size）。 $W$  反映网络中可存储的待发送数据量大小，其计算值等于 RTT 与链路中最小通行速率（即发送端与接收端传输路径中的“瓶颈”）的乘积。逼常的策略是，为使网络资源得到高效利用，应保证在网络中传输的数据量达到 BDP。但若在传输数据值远高于 BDP 时，会引入不必要的延时（参见 16.10 节），所以这也是不可取的。在网络中如何确定一个连接的 BDP 是难点，需要考虑诸多因素，如路由、时延、统计复用（即共用传输资源）水平随时间的变化性等。

这里我们主要讨论由 TCP 发送方的数据发送而产生的拥塞，但也要注意因接收方回复 ACK 而产生的相反方向链路上的拥塞，目前也有相关研究针对该问题。在文献 [RFC5690] 中介绍了一种方法，该方法中 TCP 接收方需要根据一定比率回复 ACK（即接收了多少个数据包后才能发送一个 ACK）。

## 16.2 一些经典算法

当一个新的 TCP 连接建立之初，还无法获知可用的传输资源，所以  $cwnd$  的初始值也无法确定。（也有一些例外，如有些系统的缓存容量是预先设定的，在第 14 章我们称其为目的度量

(destination metric)。) TCP 通过与接收端交换一个数据包就能获得 awnd 的值, 不需要任何明确的信号。显而易见, 获得 cwnd 最佳值的唯一方法是以越来越快的速率不断发送数据, 直到出现数据包丢失 (或网络阻塞) 为止。这时考虑立即以可用的最大速率发送 (受 awnd 的限制), 或是慢速启动发送。由于多个 TCP 连接共享一个网络传输路径, 以全速启动会影响其他连接的传输性能, 所以通常会有特定的算法来避免过快启动, 直至稳定传输后才会运行相应的其他算法。

TCP 发送方的拥塞控制操作是由 ACK 的接收来驱动或“控制”的。当 TCP 传输处于稳定阶段 (cwnd 取合适值), 接收到 ACK 回复表明发送的数据包已被成功接收, 因此可以继续发送操作。据此推理, 稳定状态下的 TCP 抑塞行为, 实际是试图使在网络传输路径上的数据包守恒 (参见图 16-1)。这里的守恒是从物理学意义上而言的——某个量 (如动量、能量) 进入一个系统不会凭空消失或出现, 而是以某种表现形式继续存在。

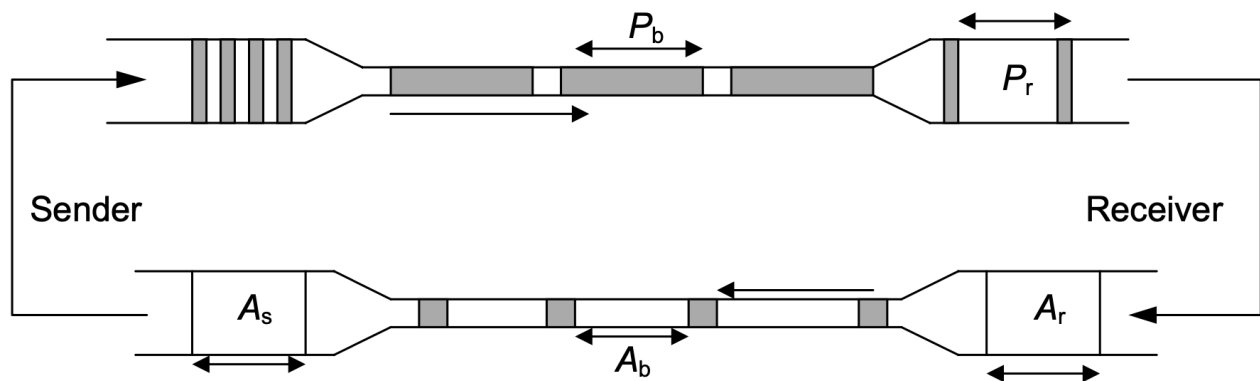


图 16.1: TCP 拥塞控制操作是基于数据包守恒原理运行的。由于传输能力有限, 数据包 ( $P$ ) 会适时地“伸展”。接收方以一定间隔 ( $P$ ) 接收到数据包后, 会陆续 (以  $A_r$  为间隔) 生成相应的 ACK, 以一定的发送间隔 ( $A_s$ ) 返回给发送方。当 ACK 陆续 (以  $A_s$  为间隔) 到达发送端时, 其到达提供了一个信号或者说“ACK 时钟”, 表明发送端可以继续发送数据。在稳定传输状态下, 整个系统可“自同步”控制 (本图改编自 [188], 源于 S. Seshan's CMU Lecture Notes, 2005.3.22)

如图 16-1 所示, 上下两条通道形似“漏桶”。发送方发送的 (较大) 数据包经上通道传输给接收方。相对较狭窄部分表示传输较慢的连接链路, 数据包需要适时地被“伸肥”。两端部分 (位于发送方和接收方) 让数据包发送前和接收后的队列。下通道传输相应 ACK 数据包。在高效传输的稳定状态下, 上下通道都不会出现包堵塞的情况, 而且在上通道中也不会有较大传输间隔。注意到发送方接收到一个 ACK 就表明可向图 16-1 中的上层通道发送一个数据包 (即网络中可容纳另一个包)。这种由一个 ACK 到达 (称作 ACK 时钟) 触发一个新数据包传输的关系称为自同步 (self-clocking)。

现在我们讨论 TCP 的两个核心算法: 慢启动和拥塞避免。这两个算法是基于包守恒和 ACK 时钟原理, 最早在 Jacobson [J88] 的经典论文里被正式提出。几年后, Jacobson 对拥塞避免算法提出了改进 [J90]。这两个算法不是同时运行的——在任一给定时刻, TCP 只运行一个算法,

但两者可以相互切换。下面我们将详细讨论这两个算法，包括如何使用以及对算法的改进。每个 TCP 连接都能独立运行这两个算法。

### 16.2.1 慢启动

当一个新的 TCP 连接建立或检测到由重传超时（RTO）导致的丢包时，需要执行慢启动。TCP 发送端长时间处于空闲状态也可能调用慢启动算法。慢启动的目的是，使 TCP 在用拥塞避免探寻更多可用带宽之前得到 cwnd 值，以及帮助 TCP 建立 ACK 时钟。通常，TCP 在建立新连接时执行慢启动，直至有丢包时，执行拥塞避免算法（参见 16.2.2 节）进入稳定状态。下文引自 [RFC5681]：

在传输初始阶段，由于未知网络传输能力，需要缓慢探测可用传输资源，防止短时间内大量数据注入导致拥塞。慢启动算法正是针对这一问题而设计。在数据传输之初或者重传计时器检测到丢包后，需要执行慢启动。

TCP 以发送一定数目的数据段开始慢启动（在 SYN 交换之后），称为初始窗口（Initial Window, IW）。IW 的值初始设一个 SMSS（发送方的最大段大小），但在 [RFC5681] 中设为一个稍大的值，计算公式如下：

$$IW = 2 * SMSS \quad \text{if } SMSS > 2190 \quad IW = 3 * SMSS \quad \text{if } 2190 \leq SMSS < 1095 \quad IW = 4 * SMSS \quad \text{if } SMSS \leq 1095 \quad (16.2)$$

上述 IW 的计算方式可能使得初始窗口为几个数据包大小（如 3 个或 4 个），为简单起见，我们只讨论  $IW = 1 \text{ SMSS}$  的情况。TCP 连接初始的  $cwnd = 1 \text{ SMSS}$ ，意味着初始可用窗口 W 也为 1 SMSS。注意到大部分情况下，SMSS 为接收方的 MSS（最大段大小）和路径 MTU（最大传输单元）两者中较小值。

假设没有出现丢包情况且每个数据包都有相应的 ACK，第一个数据段的 ACK 到达，说明可发送一个新的数据段。每接收到一个好的 ACK 响应，慢启动算法会以  $\min(N, \text{SMSS})$  来增加 cwnd 值。这里的 N 是指在未经确认的传输数据中能通过这一“好的 ACK”确认的字节数。所谓的“好的 ACK”是指新接收的 ACK 号大于之前收到的 ACK。

已被 ACK 确认的字节数目用于支持适当字节计数（Appropriate Byte Counting, ABC）[RFC3465]，这是 [RFC5681] 推荐的实验规范。ABC 用于计数“ACK 分裂”攻击（将在 16.12 节叙述），指利用许多较小 ACK 使 TCP 发送方加速发送。Linux 利用布尔系统配置变量 `net.ipv4.tcp_abc` 设定 ABC 是否可用（默认不可用）。在最近的几个 Windows 版本中，ABC 默认开启。

因此，在接收到一个数据段的 ACK 后，通常 cwnd 值会增加到 2，接着会发送两个数据段。

如果成功收到相应的新的 ACK,  $cwnd$  会由 2 变 4, 由 4 变 8, 以此类推。一般情况下, 假设没有丢包且每个数据包都有相应 ACK, 在  $k$  轮后  $W$  的值为  $W=2^k$ , 即  $k=10 \lg 2W$ , 需要  $K$  个 RTT 时间操作窗口才能达到  $W$  大小。这种增长看似很快 (以指数函数增长), 但若与一开始就允许以最大可用速率 (即接收方通知窗口大小) 发送相比, 仍显缓慢。 ( $W$  不会超过  $awnd$ 。)

如果假设某个 TCP 连接中接收方的通知窗口非常大 (比如说, 无穷大), 这时  $cwnd$  就是影响发送速率的主要因素 (设发送方有较大发送需求)。如前所述,  $cwnd$  会随着 RTT 呈指数增长。因此, 最终  $cwnd$  ( $W$  也如此) 会增至很大, 大量数据包的发送将导致网络瘫痪 (TCP 吞吐量与  $W/RTT$  成正比)。当发生上述情况时,  $cwnd$  将大幅度减小 (减至原值一半)。这是 TCP 由慢启动阶段至拥塞避免阶段的转折点, 与  $cwnd$  和慢启动阈值 ( $slow\ start\ threshold, ssthresh$ ) 相关。

图 16-2 (左) 描述了慢启动操作。数值部分以 RTT 为单位。假设该连接首先发送一个包 (图上部), 返回一个 ACK, 接着在第二个 RTT 时间里发送两个包, 会接收到两个 ACK。TCP 发送方每接收一个 ACK 就会执行一次  $cwnd$  的增长操作, 以此类推。右图描述了  $cwnd$  随时间增长的指数函数。图中另一条曲线显示了每两个数据包收到一个 ACK 时  $cwnd$  的增长情况。通常在 ACK 延时情况下会采用这种方式, 这时的  $cwnd$  仍以指数增长, 只是增幅不是很大。正因 ACK 可能会延时到达, 所以一些 TCP 操作只在慢启动阶段完成后才返回 ACK。Linux 系统中, 这被称为快速确认 (“快速 ACK 模式”), 从内核版本 2.4.4 开始, 快速确认一直是基本 TCP/P 协议栈的一部分。

## 拥塞避免

如上所述, 在连接建立之初以及由超时判定丢包发生的情况下, 需要执行慢启动操作。在慢启动阶段,  $cwnd$  会快速增长, 帮助确立一个慢启动阈值。一旦达到阈值, 就意味着可能有更多可用的传输资源。如果立即全部占用这些资源, 将会使共享路由器队列的其他连接出现严重的丢包和重传情况, 从而导致整个网络性能不稳定。

为了得到更多的传输资源而不致影响其他连接传输, TCP 实现了拥塞避免算法。一旦确立慢启动阈值, TCP 会进入拥塞避免阶段,  $cwnd$  每次的增长值近似于成功传输的数据段大小。这种随时间线性增长方式与慢启动的指数增长相比级慢许多。更准确地说, 每接收一个新的 ACK,  $cwnd$  会做以下更新:

$$cwnd_{n+1} = cwnd_n + SMSS * SMSS / cwnd_n, \quad (16.3)$$

分析上式, 假设  $cwnd_0 = k * SMSS$  字节分  $k$  段发送, 在接收到第一个 ACK 后,  $cwnd$  的值增长了  $1/k$  倍:

$$cwnd_1 = cwnd_0 + SMSS * SMSS / cwnd_0 = k * SMSS + SMSS * (SMSS / (k * SMSS)) = k * SMSS + (1/k) * SMSS \quad (16.4)$$

随着每个新的 ACK 到达,  $cwnd$  会有相应的小幅增长 (取决于上式中的  $k$  值), 整体增长率呈现轻微的次线性。尽管如此, 我们通常认为拥塞避免阶段的窗口随时间线性增长 (见图 16-3),



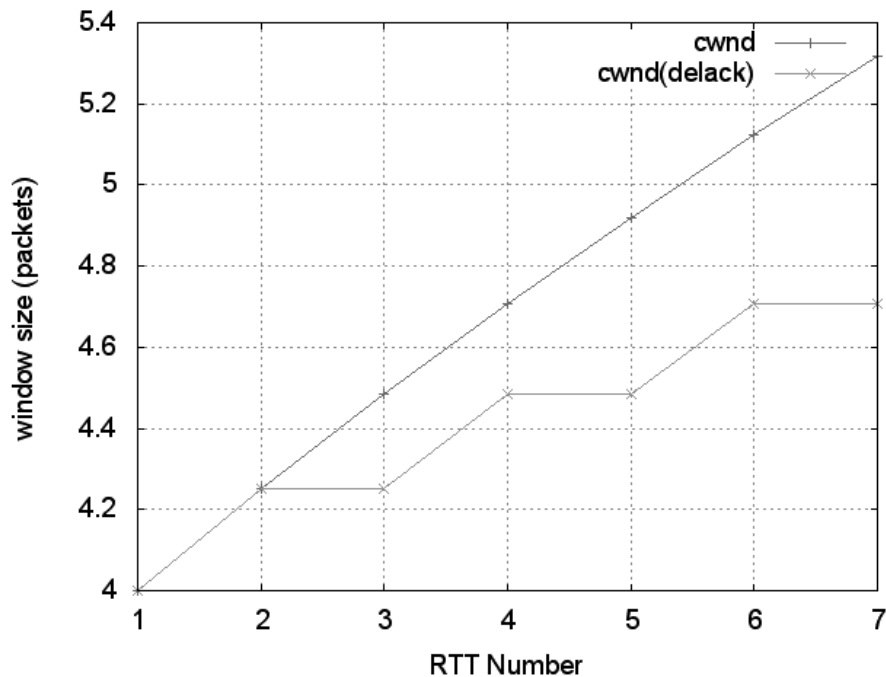


图 16.2: 经典慢启动算法操作。在没有 ACK 延时情况下, 每接收到一个好的 ACK 就意味着发送方可以发送两个新的数据包 (左)。这会使得发送方窗口随时间呈指数增长 (右, 上方曲线)。当发生 ACK 延时, 如每两个数据包生成一个 ACK, *cwnd* 仍以指数增长, 但增幅较小 (右, 下方曲线)

而慢启动阶段呈指数增长 (见图 16-2)。这个函数也称为累加增长, 因为每成功接收到相应数据, *cwnd* 就会增加一个特定值 (这里大约是一个包大小)。

图 16-3 (左) 描述了拥塞避免操作。数值部分仍是以 RTT 为单位。假设连接发送了 4 个数据包 (图上方), 返回了 4 个 ACK, *cwnd* 可以有相应的增长。在第 2 个 RTT 阶段, 增长可达到整数值, 使得 *cwnd* 增加一个 MSS, 这样可以继续发送一个新的数据包。右图描绘了 *cwnd* 随时间近似呈线性增长。另一曲线模拟 ACK 延时, 显示了每两个数据包收到一个 ACK 时 *cwnd* 的增长情况。这时的 *cwnd* 仍近似呈线性增长, 只是增幅不是很大。

拥塞避免算法假设由比特错误导致包丢失的概率很小 (远小于 1%), 因此有丢包发生就表明从源端到目的端必有某处出现了拥塞。如果假设不成立, 比如在无线网络中, 那么即使没有拥塞 TCP 传输也会变慢。另外, *cwnd* 的增大可能会经历多个 RTT, 这就需要有充裕的网络资源, 并得到高效利用。这些问题还有很大的研究空间, 以后我们将会讨论其中一些方法。

### 16.2.2 慢启动和拥塞避免的选择

在通常操作中, 某个 TCP 连接总是选择运行慢启动和拥塞避免中的一个, 不会出现两者同时进行的情况。现在我们考虑, 在任一给定时刻如何决定选用哪种算法。我们已经知道, 慢启动是在连接建立之初以及超时发生时执行的。那么决定使用慢启动还是拥塞避免的关键因素是什

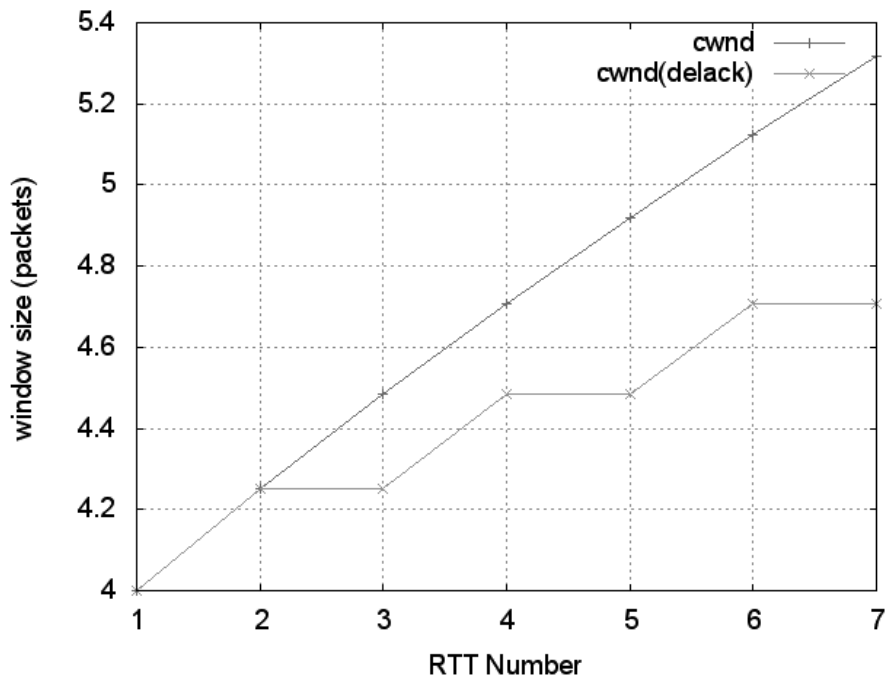


图 16.3: 拥塞避免算法操作。若没有 ACK 延时发生, 每接收一个好的 ACK, 就意味着发送方可继续发送  $1/W$  个新的数据包。发送窗口随时间近似呈线性增长 (右, 上方曲线)。当有 ACK 延时, 如每隔一个数据包生成一个 ACK,  $cwnd$  仍近似呈线性增长, 只是增幅较小 (右, 下方曲线)

么呢?

前面我们已经提到过慢启动阈值。这个值和  $cwnd$  的关系是决定采用慢启动还是拥塞避免的界线。当  $cwnd < ssthresh$ , 使用慢启动算法: 当  $cwnd > ssthresh$ , 需要执行拥塞避免: 而当两者相等时, 任何一种算法都可以使用。由上面描述可以得出, 慢启动和拥塞避免之间最大的区别在于, 当新的 ACK 到达时,  $cwnd$  怎样增长。有趣的是, 慢启动阈值不是固定的, 而是随时间改变的。它的主要目的是, 在没有丢包发生的情况下, 记住上一次“最好的”操作窗口估计值。换言之, 它记录 TCP 最优窗口估计值的下界。

慢启动阈值的初始值可任意设定 (如  $awnd$  或更大), 这会使得 TCP 总是以慢启动状态开始传输。当有重传情况发生, 无论是超时重传还是快速重传,  $ssthresh$  会按下式改变:

$$ssthresh = \max \quad /2, 2 * SMSS \quad (16.5)$$

我们已经知道, 如果出现重传情况, TCP 会认为操作窗口超出了网络传输能力范围。这时会将慢启动阈值 ( $ssthresh$ ) 减小至当前窗口大小的一半 (但不小于  $2*SMSS$ ), 从而减小最优窗口估计值。这样通常会导致  $ssthresh$  减小, 但也有可能会使之增大。分析 TCP 拥塞避免的操作流程, 如果整个窗口的数据都成功传输, 那么  $cwnd$  值可以近似增大 1 SMSS。因此, 若  $cwnd$  在一段时间范围内已经增大, 将  $ssthresh$  设为整个窗口大小的一半可能使其增大。这种情况发生在当

TCP 探测到更多可用带宽时。在慢启动和拥塞避免结合的情况下，`ssthresh` 和 `cwnd` 的相互作用使得 TCP 拥塞处理行为显现其独有特性。下面我们探讨将两者结合的完整的算法。

### 16.2.3 Tahoe、Reno 以及快速恢复算法

至此讨论的慢启动和拥塞避免算法，组成了 TCP 拥塞控制算法的第一部分。它们于 20 世纪 80 年代末期在加州大学伯克利分校的 4.2 版本的 UNIX 系统中被提出，称为伯克利软件版本，或 BSD UNIX。至此开始了以美国城市名命名各个 TCP 版本的习惯，尤其是那些赌博合法的城市。

4.2 版本的 BSD（称为 Tahoe）包含了一个 TCP 版本，它在连接之初处于慢启动阶段，若检测到丢包，不论由于超时还是快速重传，都会重新进入慢启动状态。有丢包情况发生时，Tahoe 简单地将 `cwnd` 减初始值（当时设为 1 SMSS）以达到慢启动目的，直至 `cwnd` 增长为 `ssthresh`。

这种方法带来的一个问题是，对于有较大 BDP 的链路来说，会使得带宽利用率低下。因为 TCP 发送方经重新慢启动，回归到的还是未丢包状态（`cwnd` 启动初始值设置过小）。为解决这一问题，针对不同的丢包情况，重新考虑是否需要重回慢启动状态。若是由重复 ACK 引起的丢包（引发快速重传），`cwnd` 值将被设为上一个 `ssthresh`，而非先前的 1 SMSS。（在大多数 TCP 版本中，超时仍是引发慢启动的主要原因。）这种方法使得 TCP 无须重新慢启动，而只要把传输速率减半即可。

进一步讨论较大 BDP 链路的情况，结合之前提到的包守恒原理，我们可以得出结论，只要接收到 ACK 回复（包括重传 ACK），就有可能传输新的数据包。BSD UNIX 的 4.3 BSD Reno 版中的快速恢复机制就是基于上述结论。在恢复阶段，每收到一个 ACK，`cwnd` 就能（临时）增长 1 SMSS，相应地就意味着能发送一个新的数据包。因此拥塞窗口在一段时间内会急速增长，直到接收一个好的 ACK。不重复的（“好的”）ACK 表明 TCP 结束恢复阶段，拥塞已减少到之前状态。TCP Reno 算法得到了广泛应用，并成为“标准 TCP”的基础。

### 16.2.4 标准 TCP

尽管究竟哪些构成了“标准”TCP 还存在争议，但我们讨论过的上述算法毋庸置疑都属于标准 TCP。慢启动和拥塞避免算法通常结合使用，[RFC5681] 给出了其基本方法。这个规范并不要求严格使用这些精确算法，TCP 实现过程仅利用其核心思想。

总结 [RFC5681] 中的结合算法，在 TCP 连接建立之初首先是慢启动阶段（`cwnd = IW`），`ssthresh` 通常取一较大值（至少为 `awnd`）。当接收到一个好的 ACK（表明新的数据传输成功），`cwnd` 会相应更新：

$$cwnd += SMSS \quad cwnd < ssthresh \quad cwnd += SMSS * SMSS / cwnd \quad cwnd > ssthresh \quad (16.6)$$

当收到三次重复 ACK（或其他表明需要快速重传的信号）时，会执行以下行为：

1. ssthresh 更新为大于等式 (16-1) 中的值。
2. 启用快速重传算法，将 cwnd 设为 (ssthresh + 3\*SMSS)。
3. 每接收一个重复 ACK, cwnd 值暂时增加 1 SMSS。
4. 当接收到一个好的 ACK，将 cwnd 重设为 ssthresh。

以上第 2 步和第 3 步构成了快速恢复。步骤 2 设置 cwnd 大小，首先 cwnd 通常会被减为之前值的一半。然后，考虑到每接收一个重复 ACK，就意味着相应的数据包已成功传输（因此新的数据包就有发送机会），cwnd 值会相应地暂时增大。这一步也可能出现 cwnd 加速递减的情况，因为通常 cwnd 会乘以某个值（这里取 0.5）来形成新的 cwnd。步骤 3 维持 cwnd 的增大过程，使得发送方可以继续发送新的数据包（在不超过 awnd 的情况下）。步骤 4 假设 TCP 已完成恢复阶段，所以 ewnd 的临时膨胀也消除了（有时称这一步为“收缩”）。

以下两种情况总会执行慢启动：新连接的建立以及出现重传超时。当发送方长时间处于空闲状态，或者有理由怀疑 cwnd 不能精确反映网络当前拥塞状态（参见 16.3.5 节）时，也可能引发慢启动。在这种情况下，cwnd 的初始值将被设为重启窗口（RW）。在文献 [RFC5681] 中，推荐 RW 值为  $RW = \min(IW, cwnd)$ 。其他情况下，慢启动中 ownd 初始设为 IW。

## 16.3 对标准算法的改进

经典的标准 TCP 算法在传输控制领域做出了重大贡献，尤其针对网络拥塞崩溃这一难题，取得了显著效果。

在 1986 1988 年，网络拥塞崩溃是引起广泛关注的难点问题。1986 年 10 月，作为早期互联网的重要组成部分，NSFNET 主干网出现了一次严重故障，运行速度仅为其应有速度的千分之十（称为“NSFNET 危机”）。问题的主要形成原因在于对大量的重传没有任何控制操作。持续的拥塞状态导致了严重的丢包现象（由于更多的重传操作）和吞吐量低下。采用经典拥塞控制算法有效地解决了这一问题。

然而，仍然可以找到值得改进的地方。考虑到 TCP 的普遍使用性，越来越多的研究致力于使 TCP 在更广泛的环境里更好地工作。下面我们提出几种方法，现在许多 TCP 版本也已经实现。

### 16.3.1 NewReno

快速恢复带来的一个问题是，当一个传输窗口出现多个数据包丢失时，一旦其中一个包重传成功，发送方就会接收到一个好的 ACK，这样快速恢复阶段中  $cwnd$  窗口的暂时膨胀就会停止，而事实上丢失的其他数据包可能并未完成重传。导致出现这种状况的 ACK 称为局部 ACK (partial ACK)。Reno 算法在接收到局部 ACK 后就停止拥塞窗口膨胀阶段，并将其减小至特定值，这种做法可能导致在重传计时器超时之前，传输通道一直处于空闲状态。力理解出现这种情况的原因，我们首先明确，TCP（无选择确认机制）需要通过三个（或重复阈值）重复 ACK 包作为信号才能触发快速重传机制。假如网络中没有足够的数据包在传输，那么就不可能因丢包而触发快速重传，最终导致重传计时器超时，引发慢启动操作，从而严重影响网络吞吐性能。

为解决上述问题，[RFC3782] 提出了一种改进算法，称为 NewReno。该算法对快速恢复做出了改进，它记录了上一个数据传输窗口的最高序列号（即我们在第 14 章提到的恢复点）。仅当接收到序列号不小于恢复点的 ACK，才停止快速恢复阶段。这样 TCP 发送方每接收一个 ACK 后就能继续发送一个新数据段，从而减少重传超时的发生，特别针对一个窗口出现多个包丢失的情况时。NewReno 是现在比较常用的一个 TCP 版本，它不会出现经典快速重传的问题，实现起来也没有选择确认（SACK）复杂。然而，当出现上述多个丢包情况时，利用 SACK 机制能比 NewReno 获得更好的性能，但需要较为复杂的拥塞控制操作，下面我们会讨论这一问题。

### 16.3.2 采用选择确认机制的 TCP 拥塞控制

在 TCP 引入 SACK 与选择性重复之后，发送方能够更好地确定发送哪个数据段来填补接收方的空缺（参见第 14 章）。为了填补接收数据的空缺，发送方通常只发送丢失的数据段，直至完成所有重传。这和前面提到的基本的快速重传/恢复机制有所差别。

在快速重传/恢复情况下，当出现丢包，TCP 发送方只重传它认为已经丢失的包。如果窗口  $W$  允许，还可以发送新的数据包。在快速恢复阶段，由于窗口大小会随着每个 ACK 的到达而膨胀，在完成重传后，通常发送方能会有更大的窗口发送更多新数据。采用 SACK 机制后，发送方可以知晓多个数据段的丢失情况。因为这些数据都在有效窗口内，理论上可以即时重传。然而，这样可能会在较短时间内向网络中注入大量数据，削弱拥塞控制效果。SACK TCP 会引发以下问题：在恢复阶段，只使用  $cwnd$  作为发送方滑动窗口的界限来表示发送多少个（以及哪些）数据包是不够的，且选择发送哪些数据包与发送时间紧密相关。换言之，SACK TCP 强调拥塞管理和选择重传机制的分离。传统（无 SACK）TCP 则将两者结合。

一种实现分离的方法是，除了维护窗口，TCP 还负责记录注入网络的数据量。[RFC3517] 称其为管道（pipe）变量，这是对在外数据的估计值。管道变量以字节（或包，依不同实现方式而定）为单位，记录传输和重传情况（不考虑丢包，将两者同等对待）。假设  $awnd$  值较大，只要不等式  $cwnd - pipe \geq MSS$  成立，在任何时候 SACK TCP 均可发送数据。这里  $cwnd$  仍被用

来限定可传输至网络中的数据量，但除了窗口本身，网络中数据量的估计值也被记录了。[FF96] 详细分析比较了 SACK TCP 和传统 TCP 的拥塞控制方法，并做了相关仿真工作。

### 16.3.3 转发确认 (FACK) 和速率减半

对基于 Reno (包括 NewReno) 的 TCP 版本来说，当快速重传结束后  $cwnd$  值减小，在 TCP 发送新数据之前至少可以接收一半已发送数据返回的 ACK。这和检测到丢包后立即将拥塞窗口值减半相一致。这样 TCP 发送端在前一半的 RTT 时间内处于等待状态，在后一半 RTT 才能发送新数据，这是我们所不愿看到的。

在丢包后，为避免出现等待空闲而又不违背将拥塞窗口减半的做法，[MM96] 提出了转发确认 (forward acknowledgment, FACK) 策略。FACK 包含了两部分算法，称为“过度衰减”(overdamping) 和“缓慢衰减”(rampdown)。从最初想法的提出到改进，最终在 Hoe 的工作基础上 [H96] 形成了统一的算法，称为速率减半 (rate halving)。为控制算法尽可能有效地运行，进一步添加了界定参数，完整的算法被称为带界定参数的速率减半 (Rate-Halving with Bounding Parameters, RHBP) 算法 [PSCRH]。

RHBP 的基本操作是，在一个 RTT 时间内，每接收两个重复 ACK，TCP 发送方可发送一个新数据包。这样在恢复阶段结束前，TCP 已经发送了一部分新数据，与之前的所有发送都挤在后半个 RTT 时间段内相比，数据发送比较均衡。由于过度集中的发送操作可能持续多个 RTT，对路由缓存造成负担，因此均衡发送是比较有利的。

为了记录较为精确的在外数据估计值，RHBP 利用 SACK 信息决定 FACK 策略：已知的最大序列号的数据到达接收方时，在外数据值加 1。注意区分即将发送数据的最大序列号 (图 15-9 中的  $SND.NXT$ )，FACK 给出的在外数据估计值不包括重传。

RHBP 中区分了调整间隔 (adjustment interval,  $cwnd$  的修正阶段) 和恢复间隔 (repair interval, 数据重传阶段)。一旦出现丢包或其他拥塞信号就立即进入调整间隔。调整间隔结束后  $cwnd$  的最终值为：至检测时间为止，网络中已正确传输的窗口数据量的一半。RHBP 要求发送方传输数据需满足下式：

$$(SND.NXT - fack + retransdata + len) < cwnd \quad (16.7)$$

上面的等式得到了包括重传的在外数据值，确保当继续发送一个  $len$  长度的新数据，也不会超过  $cwnd$ 。假设在 FACK 之前的数据已经不在网络中 (如丢失或被接收)，这样  $cwnd$  就能很好地控制 SACK 发送方的发送。然而由于 SACK 的选择确认特性，可能导致数据包的传输次序过度重排。

Linux 系统实现了 FACK 和速率减半，并默认启用。若 SACK 开启，并将布尔配置变量 `net.ipv4.tcp_fack` 置 1，就会激活 FACK。当检测到网络中出现数据包失序，FACK 的进一步行为将被禁用。

速率减半是调节发送操作或避免集中发送的方法之一。我们已经了解了它的优点，但这种方法仍然存在一些问题。[ASA00] 利用仿真的方法，详细分析了 TCP 发送调度，结果显示在很多情况下，它的性能劣于 TCP Reno。另外，研究表明，在接收窗口限制 TCP 连接的情况下，速率减半方法收效甚微 [MM05]。

### 16.3.4 限制传输

[RFC3042] 提出了限制传输 (limited transmit)，它对 TCP 做出了微小改进，目的在于使 TCP 能在可用窗口较小的情况下更好工作。之前已经提到，在 Reno 算法中，通常需要三次重复 ACK 表明数据包丢失。在窗口较小的情况下，当出现丢包，网络中可能没有足够的包去引发快速重传/恢复机制。

采用限制传输策略，TCP 发送方每接收两个连续的重复 ACK，就能发送一个新数据包。这就使得网络中的数据包维持一定数量——足以触发快速重传。TCP 因此也可以避免长时间等待 RTO（可能达到几百毫秒，相对时间较长）而导致吞吐性能下降。限制传输已经成为 TCP 推荐策略。速率减半也是限制传输的一种形式。

### 16.3.5 拥塞窗口校验

TCP 拥塞管理可能会出现一个问题，那就是发送端可能在一段时间内停止发送（由于没有新数据需要发送或者其他原因阻住发送）。通常情况下，发送操作不会暂停。发送端发送数据，同时接收 ACK 反馈，以此估计一定时间内（一个 RTT）的 cwnd 和 ssthresh。

在发送操作持续一段时间后，cwnd 可能会增至一个较大值。若发送需要暂停（一定时间后会恢复），根据此时 cwnd 的值，在暂停前发送方仍可向网络中（高速）注入大量数据。若暂停时间足够长，之前的 cwnd 可能无法准确反映路径中的拥塞状况。

[RFC2861] 提出了一种拥塞窗口校验 (Congestion Window Validation, cwv) 机制。在发送长时间暂停的情况下，由 ssthresh 维护 cwnd 保存的“记忆”，之后 cwnd 值会衰减。为理解这种机制，需要区分空闲 (idle) 发送端和应用受限 (application-limited) 发送端。对空闲发送端而言，没有发送新数据的需求，之前发送的数据也已经成功接收 ACK。因此，整个连接处于空闲状态——除了必要的窗口更新外（参见第 15 章），没有数据和 ACK 的传输。应用受限发送端则需要传输数据，但由于某种原因无法发送（可能由于处理器正忙或者下层阻住数据发送）。这种情况会导致连接利用率低下，但并非完全空闲，之前已发送数据返回的 ACK 仍可传输。

CwV 算法原理如下：当需要发送新数据时，首先看距离上次发送操作是否超过一个 RTO。如果超过，则

- 更新 ssthresh 值—设  $\max(ssthresh, (3/4) * cwnd)$ 。
- 每经一个空闲 RTT 时间，cwnd 值就减半，但不小于 1 SMSS。

对于应用受限阶段（非空闲阶段），执行相似的操作：

- 已使用的窗口大小记为  $W_{\text{used}}$ 。
- 更新  $ssthresh$  值——设为  $\max(ssthresh, (3/4) * cwnd)$ 。
- $cwnd$  设为  $cwnd$  和  $W_{\text{used}}$  的平均值。

上述操作均减小了  $cwnd$ ，但  $ssthresh$  维护了  $cwnd$  的先前值。第一种情况中，如果传输通道长时间空闲， $cwnd$  将会显著减小。在某些情况下，这种拥塞窗口的处理方法可以取得更好效果。根据作者的研究，避免空闲阶段可能发生的大数据量注入，可以减轻对有限的路由缓存的压力，从而减少丢包情况的产生。注意到 CWV 减小了  $cwnd$  值，但没有减小  $ssthresh$ ，因此采用这种算法的通常结果是，在长时间发送暂停后，发送方会进入慢启动阶段。Linux TCP 实现了 CWV 并默认启用。

## 16.4 伪 RTO 处理——Eifel 响应算法

在第 15 章已经提到，若 TCP 出现突发的延时，即使没有出现丢包，也可能造成重传超时的假象。这种伪重传现象的发生可能由于链路层的某些变化（如峰宽转换），也可能是由于突然出现严重拥塞造成 RTT 大幅增长。当出现重传超时，TCP 会调整  $ssthresh$  并将  $cwnd$  置为  $IW$ ，从而进入慢启动状态。假如没有出现实际丢包，在 RTO 之后到达的 ACK 会使得  $cwnd$  快速增大，但在  $cwnd$  和  $ssthresh$  值重新稳定前，仍然会有不必要的重传，浪费传输资源。

针对上述问题已有相关探测方法。我们在第 14 章讨论了其中的一些方法（如 DSACK、Eifel、F-RTO）。其中任一探测方法只要结合相关响应算法，就能“还原”TCP 对拥塞控制变量的操作。一种比较常用（即在 IETF 标准化过程中）的响应算法就是 Eifel 响应算法 [RFC4015]。

Eifel 算法包含检测算法和响应算法两部分，两者在理论上是独立的。任何使用 Eifel 响应算法实现的 TCP 操作，必须使用相应的标准操作规范或实验 RFC（即被记录的 RFC）中规定的检测算法。

Eifel 响应算法用于处理重传计时器以及重传计时器超时后的拥塞控制操作。这里我们只讨论与拥塞相关的响应算法。在首次发生超时重传时，Eifel 算法开始执行。若认为出现伪重传情况，会撤销对  $ssthresh$  值的修改。在所有情况下，若因 RTO 而需改变  $ssthresh$  值，在修改前需要记录一个特殊变量： $pipe\_prev = \min(\text{在外数据值}, ssthresh)$ 。然后需要运行一个检测算法（即之前提到的检测方法中的某个）来判断 RTO 是否真实。假如出现伪重传，则当到达一个 ACK 时，执行以下步骤：

1. 若接收的是包含 ECN-Echo 标志位的好的 ACK，停止操作（参见 16.11 节）。
2.  $cwnd = \text{在外数据值} + \min(\text{bytes\_acked}, IW)$ （假设  $cwnd$  以字节为单位）。



### 3. ssthresh = pipe\_prevo

在改变 ssthresh 之前需要设置 pipe\_prev 变量。pipe\_prev 用于保存 ssthresh 的记录值，以便在步骤 3 中重设 ssthresh。步骤 1 针对带 ECN 标志位的 ACK 的情况（在 16.11 节中将详细讨论 ECN）。这种情况下撤销 ssthresh 修改会引入不安全因素，所以算法终止。步骤 2 和步骤 3 是算法的主要部分（针对 cwnd）。步骤 2 将 cwnd 设置一定值，允许不超过 IW 的新数据进入传输通道。因为即使在未知链路拥塞与否的状况下，发送 IW 的新数据也被认为是安全的。步骤 3 在真正的 RTO 发生前重置 ssthresh，至此撤销操作完成。

## 16.5 扩展举例

下面我们通过一个例子来演示一下前面章节提到的操作算法。利用 sock 程序，在一条 DSL 线路上传输 2.5MB 数据。发送方和接收方分别为 Linux（2.6）和 FreeBSD（5.4）。链路在发送方向上限速约为 300Kb/s。FreeBSD 接收端处于高带宽连接。发送端至接收端的最小 RTT 为 15.9ms，需经 17 个跳步。大部分处理操作均使用基础算法（如慢启动和拥塞避免），以避免不同操作系统的实现细节差异（后面我们会提到相关问题）。下面开始实验，首先在接收端运行如下操作命令：

FreeBSD8

```
sock -l -r 32768 -R 233016 -8 6666
```

该命令为 sock 程序设置了一个较大的套接字接收缓存（228KB），并执行大数据量的读操作（32KB）。对于传输链路来说，接收缓存已足够大。接着设置发送端为发送模式，命令如下：

Linux? sock -n20 -i -w 131072 -S 262144 128.32.37.219 6666

该命令选择了一个较大的发送缓存并发送了 20×131 072 字节（2.5MB）数据。利用发送端的 topdump 可以记录数据包的传输轨迹，命令如下：

Linux# tcpdump -s 128 -w sack-to-free-12.td port 6666

该命令确保每个数据包至少记录 128 字节，对获取 TCP 和 IP 头部信息已足够。得到相关记录后，可以采用工具 tcptrace [TCPTRACE] 来收集连接相关的统计信息，命令如下：

Linux? toptrace -Wl sack-to-free-12.td

该命令需要提供拥塞窗口的相关信息，其输出格式较长（详细），输出如下：

从上述输出中可以得到很多连接方面的信息。我们首先关注输出的左半部分（a->b）。可以看到在 a b 方向上共传输了 1903 个包，其中 1902 个为 ACK。这和预计是相符的，因为通常第

一个传输的包是 SYN—唯一一个没有 ACK 标志位的包。纯 ACK 包是指不包含传输数据的包。发送端一共发送了两个纯 ACK 包，一个是在连接初始阶段响应接收端发送的 SYN+ACK 包，另一个是在连接结束时发送的。第二栏 (b->a 方向) 显示，接收端共发送了 1272 个包，全部是 ACK。其中，1270 个是纯 ACK 包，并有 79 个是 SACK 包（即包含 SACK 选项的 ACK）。两个“不纯”的 ACK 分别是连接之初的 SYN+ACK 以及结束时的 FIN+ACK。

从下面的 5 个值可以看出部分数据经过了重传。可以看到，单次传输的数据为 2621440 字节（即没有重传），但总的传输量达到了 2659240 字节，说明有 2659240-2621440=37800 字节数据经历了多于一次的传输。接下来的两个字段验证了这一点，这些数据被分成 27 个数据包进行重传，平均每个重传数据段大小为 1399 字节。由于在 100.476s 时间内完成了 2659240 字节的传输，平均吞吐量为 26466B/s（约 212kb/s）。平均优质吞吐量（goodput，即单位时间内无重传的数据量）为 2621440/100.476=26090B/s，约 209kb/s。可以看出，传输性能受到了严重干扰。我们可以利用 Wireshark 查看 TCP 操作并分析干扰产生的原因。

为得到记录结果的图像，可以使用 Wireshark 的统计菜单中的“统计 TCP 流图 | 时间序列图”（Statistics | TCP Stream Graph | Time-Sequence Graph）功能（tcptrace），如图 16-4 所示（为方便讨论已用箭头标记）。

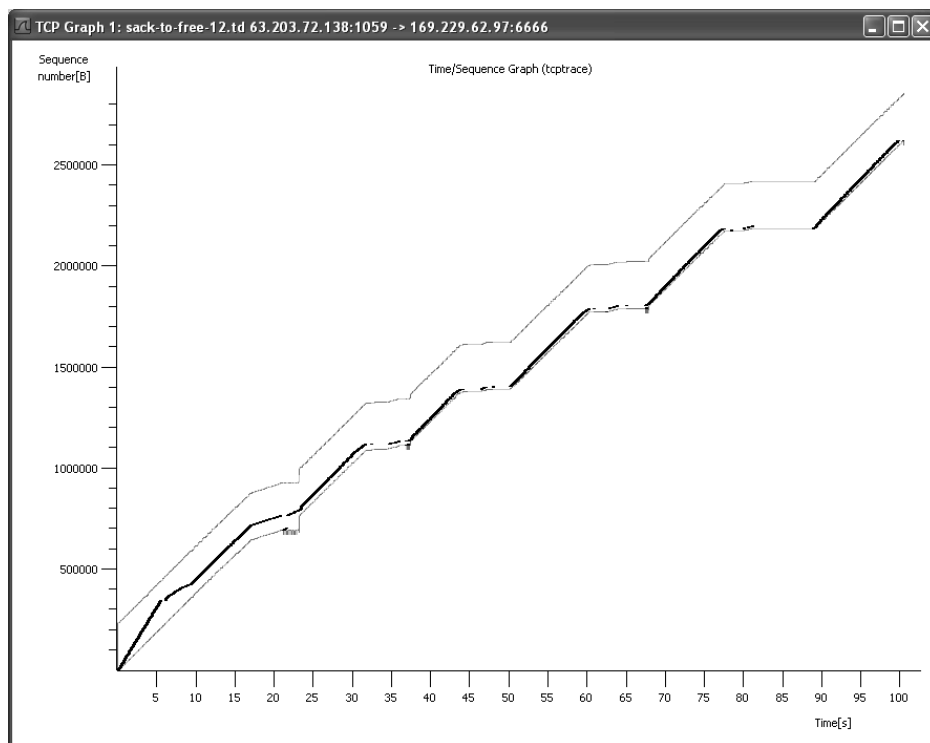


图 16.4: 拥塞避免算法操作。若没有 ACK 延时发生，每接收一个好的 ACK，就意味着发送方可继续发送  $1/W$  个新的数据包。发送窗口随时间近似呈线性增长（右，上方曲线）。当有 ACK 延时，如每隔一个数据包生成一个 ACK，*cwnd* 仍近似呈线性增长，只是增幅较小（右，下方曲线）

图 16-4 的 y 轴表示 TCP 序列号，每小格代表 100000 个序列号。x 轴是时间，以秒为单位。黑体实线由许多小的 1 字形线段组成，每段代表 TCP 序列号范围。1 形线段的最高点表示用户数据负载大小，以字节为单位。线段的斜率为数据到达速率。斜率减小表示出现重传。在给定时间范围内的线段斜率，代表了该时间段内的平均吞吐量。从图中可以看到，在 100s 时刻，发送的最大序列号为 260000，表示粗略地对平均优质吞吐量的估计值为 26 000B/s，这与前面对 tcptrace 输出结果的分析相一致。

图中上方曲线为接收端在对应时刻可接收数据的最大序列号（最大通知窗口）。可以看到，在起始时刻，其值约为 250000，teptrace 输出中的 b->a 栏中的精确数据显示为 233016。下方曲线代表发送端在对应时刻接收到的最大 ACK 号。之前已经提到过，当 TCP 进行操作时，会增大拥塞窗口，以获取新的带宽。这和接收端的通告窗口并不冲突。这一点可以从图中看出，随着时间推移，实线部分逐渐从下方曲线向上方曲线靠近。若始终达不到上方曲线，影响网络吞吐量的主要因素可能为发送端或者网络传输资源的限制。若黑线部分始终紧贴下方曲线，则影响因素主要在于接收窗口限制。

Linux 2.6.10 TCP 发送端传输 2.5MB 文件的 Wireshark 记录，DSL 线路速率约为 300Kb/s。黑体实线代表发送序列号。上方曲线为接收端通知窗口的最高序列号（窗口右边界），下方曲线表示发送方接收到的最大 ACK 号。图中标记的 11 个事件为拥塞窗口的变化情况

### 16.5.1 慢启动行为

在分析之前，首先观察我们之前介绍过的慢启动算法的相关操作。在 Wireshark 中选择记录结果的第一个包，利用菜单中的“统计 | 流图”（Statistics | Flow Graph）功能，捕绘出在连接初始阶段包交换的过程（参见图 16-5）。

从图中可以看到初始阶段的 SYN 和 SYN+ACK 的交换过程。0.032s 时刻的 ACK 是一次窗口更新（参见第 15 章）。前两次数据包传输出现在 0.126s 和 0.127s 时刻。在 0.210s 时刻返回的 ACK 不是仅对一个数据包的确认。它的序列号为 2801，由于 TCP ACK 的累积确认性质，它是对前两个发送的数据包的响应。这是延时 ACK 的一个例子，延时 ACK 通常是每两个数据包生成一个 ACK（或如 [RFC5681] 中建议的更频繁）。对接收端（FreeBSD 5.4）来说更为特殊，它需要在每个 ACK 确认一个包和两个包之间切换。这表明平均来说，每三个数据包会返回两个 ACK（假设没有出现传输错误和重传）。在第 15 章中我们已经讨论过延时 ACK 和窗口更新问题了。

一个 ACK 完成对两个数据包的确认，使得滑动窗口可以向前滑动两个包，因此可以继续发送两个新的数据包。由于连接处于初始慢启动阶段，发送端每接收一个好的 ACK，拥塞窗口相应加 1（Linux TCP 管理的拥塞窗口以包为单位）。在上述情况下，cwnd 从 2 增至 3。因此可以继续传输三个数据包，分别在 0.2158、0.216s 和 0.217s 时刻发送。

0.264s 到达的 ACK 是对单个包的确认，表明接收方期望下次接收序列号为 4201 的数据包。

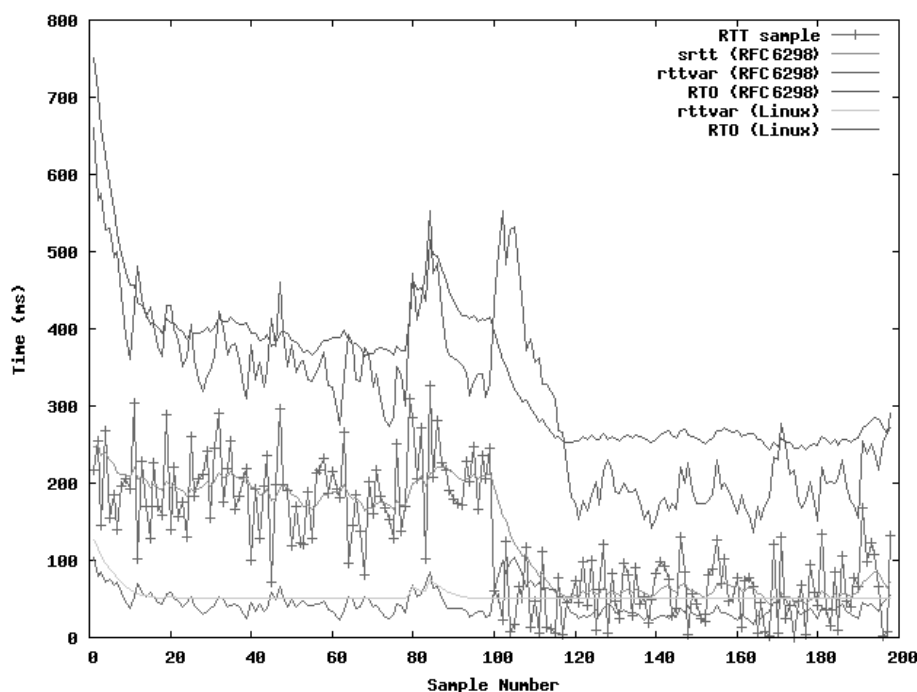


图 16.5: 拥塞避免算法操作。若没有 ACK 延时发生, 每接收一个好的 ACK, 就意味着发送方可继续发送  $1/W$  个新的数据包。发送窗口随时间近似呈线性增长 (右, 上方曲线)。当有 ACK 延时, 如每隔一个数据包生成一个 ACK,  $cwnd$  仍近似呈线性增长, 只是增幅较小 (右, 下方曲线)

然而, 4201 号以及之后的 5601 号数据包已被发送, 但仍未到达。因此, 0.264s 时刻的 ACK 使得  $cwnd$  由 3 变力 4, 但由于两个包仍处于传送状态, 只能允许继续发送两个数据包 (该 ACK 使得滑动窗口前行, 另外, 接收到这个好的 ACK 允许  $cwnd$  加 1)。这两个包的发送时间为 0.268s 和 0.268s (在同一个 1/1000 秒内)。

以上是发送端执行慢启动情况下接收端延时返回 ACK 的典型例子。这个过程持续 (每接收一个 ACK 发送两三个新数据包) 直到 5.6s。下面我们进一步讨论此时发生的情况。

### 16.5.2 发送暂停和本地拥塞 (事件 1)

如图 16-4 所示, 在 5.512s 时刻发送一个数据段后, 直到 6.162s 时刻才开始再次发送, 这中间出现了一个暂停。利用 Wireshark 的图像放大功能可以得到图 16-6。

可以看到, 在发送暂停阶段没有新数据的传输, 也没有重传, 但暂停结束后却出现了传输速率的下降, 这是为什么呢? 我们再次通过传输流记录功能一探究竟 (参见图 16-7)。

暂停前最后一次传输的数据段开启了 PSH 标志, 表明发送缓存已经清空, 所以在 5.559s 时刻 TCP 发送端已经终止发送。导致发送终止的原因可能有多种, 如发送方系统忙于处理其他任务, 无暇顾及数据传输。

我们可以看到这次暂停并不意味着重传恢复阶段的开始, 但暂停结束后线段的斜率有所下

降，表明发送速率在减小。下面将仔细观察并探讨这种行为产生的原因。

暂停前最后发送数据的序列号为  $343001+1400-1=344400$ ，该序列号之前没有发送过，所以不是重传数据。在 5.486s 时刻（已标记出）发送完数据段后，网络中已发出但未收到 ACK 的数据量达到最大值： $341\ 601+1400-205\ 801=137\ 200$  字节（98 个包），即 cwnd 值为 98 个包。5.556s 时刻到达的 ACK 表明又有两个包被成功接收。暂停前最后又发送了一个数据包，序列号为 344400，这样一共有 97 个包还来成功接收。

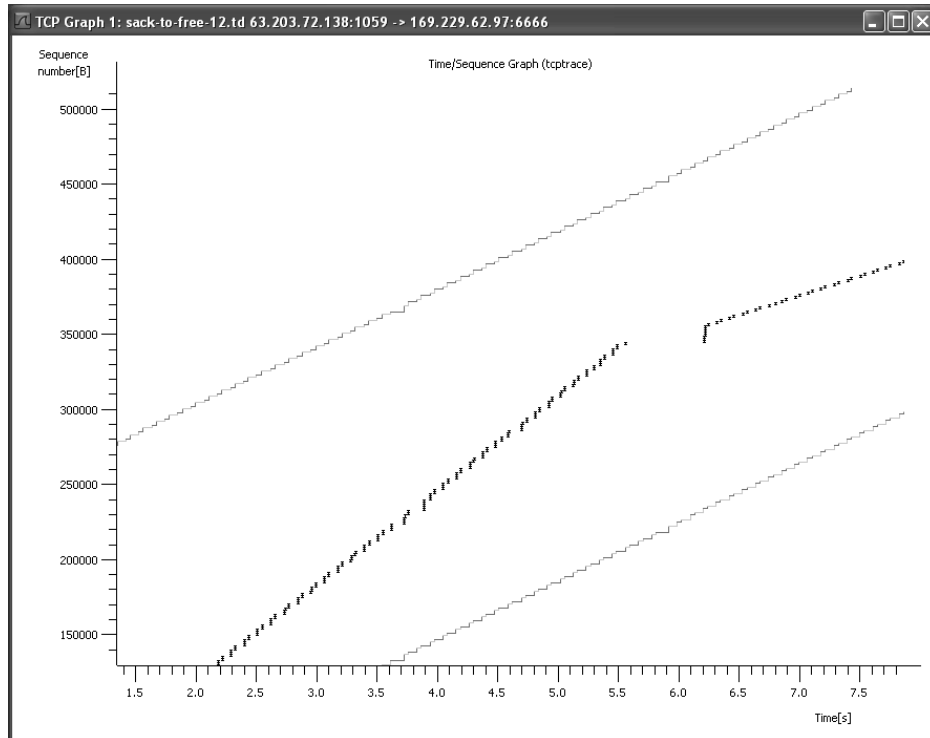


图 16.6: 在慢启动阶段后，连接暂停持续了约 512ms，接着在 5.512s 时刻恢复发送

在发送暂停阶段，共有 11 个 ACK 到达（之前提过，每个 ACK 确认一个或两个数据段）。最后一个 ACK 表明序列号为 233800 的数据段已成功传输，同时仍有 110600 字节（79 个包）的数据没有收到确认。此时，发送方开始继续发送，它可以发送的数据包个数为  $98-79=19$  个，但从图中看到，它只发送了 8 个。至 6.128s，它发送的数据段的最终序列号为  $354201+1400-1=355600$ 。

从传输流记录图中并不能很清楚地看到 TCP 当时的状况。我们预料应该会发送 19 个包，但结果只发送了 8 个。原因可能在于，下层产生的大量数据包堵塞了本地（下层）队列，使得后续包无法传送。为明确是否由下层原因导致上述问题，由于数据包经过 PppO 网络接口传输，所以在 Linux 中使用如下命令：

```
Linuxs te -s -d adise show dev ppp0 gdisc pfifo_fast 0: bands
3 prionap
1222120011111111
```

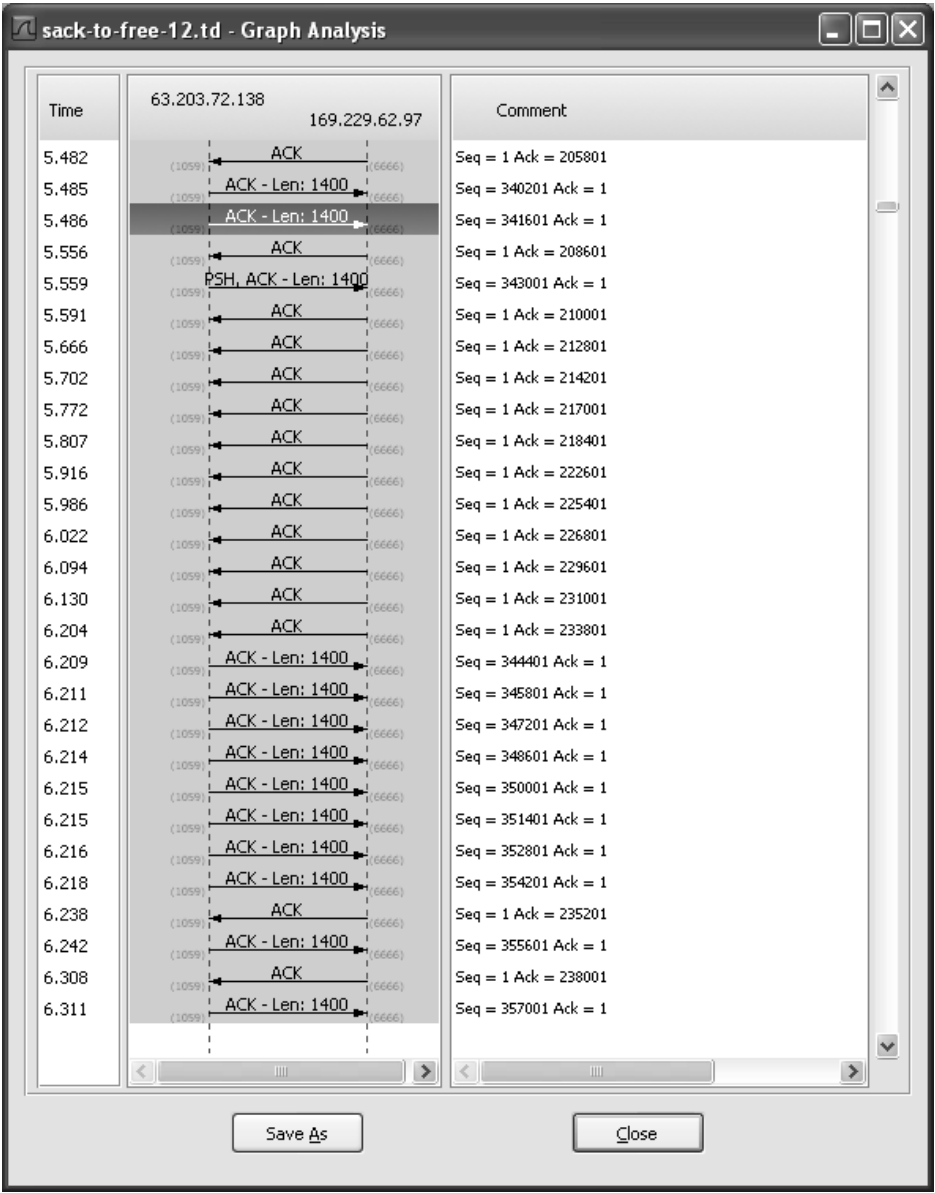


图 16.7: 在 5.559s 时刻发送方暂停发送。6.209s 时刻重新开始发送，由于本地拥塞，可发送包个数被限制为 8 个。有些 TCP 版本就是利用限制发送速率的方法来避免发送端队列拥塞

```
Sent 122569547 bytes 348574 pkts (dropped 2,  
overlimits  
0 requeues 0)
```

上述命令中的 tc 是 Linux 中用于管理包调度和流量控制子系统的指令 [LARTC]。-s 和-d 选项提供具体的记录细节。指令 qdisc show dev pppO 为显示设备 pppO 的排队规则，即管理和调度包发送的方法。注意到这里出现了两个丢包，这不是在网络传输过程中的丢包，而是出现在发送端 TCP 下层的丢包。由于丢包发生于 TCP 层以下，但又是在包的操作处理层以上，所以

传输流记录中并不记录这些包，这也是我们只看到 8 个包传输的原因。在发送端系统产生的丢包有时称为本地拥塞，产生原因在于，TCP 产生数据包的速度大于下层队列的发送速度。

Linux 流量控制子系统以及一些路由器和操作系统支持的优先级策略或 QoS 特性（如 Microsoft 的 qWave API [WQOS]），可能使用不同的排队规则，按照数据包的特性（如 IP DSCP 值或 TCP 端口号）会有不同的调度方法。对某些数据包（如多媒体数据包，TCP 纯 ACK 包等）采用优先级策略，可以提升交互式应用的用户体验。一般来说，互联网并不支持优先级策略，但许多局域网和有些企业 IP 网络中会采用这种策略。

本地拥塞是 Linux TCP 实行拥塞窗口缩减（Congestion Window Reducing, CWR）策略 [SK02] 的原因之一。首先，将 `ssthresh` 值设置为 `cwnd/2`，并将 `cwnd` 设力 `min (cwnd., 在外数据值 +1)`。在 CWR 阶段，发送方每接收两个 ACK 就将 `cwnd` 减 1，直到 `cwnd` 达到新的 `ssthresh` 值或者由于其他原因结束 CWR（如出现丢包）。这本质上和前面提到的速率减半（rate-halving）算法一致。若 TCP 发送端接收到 TCP 头部中的 ECN-Echo 也会进入 CWR 状态（参见 16.1.1 节）。

了解了这些之后，我们就可以理解前面情况的产生原因了。当 TCP 结束暂停后，它只能继续发送 8 个包。由于本地拥塞，无法传输额外的包，TCP 进入 CWR 状态。`ssthresh` 立即减为  $98/2 = 49$  个包，`cwnd` 也变为  $79+8=87$  个包。每接收两个 ACK，`cwnd` 就会减 1，这样就导致发送速率减慢，直到 8.364s 时刻 `cwnd` 值变为 66 个包。

发送速率的减小也可以从图 16-6 中观察出来，在 5.5s 时刻前，线段的斜率显示数据传输速率约为 500Kb/S。这个值大于传输方向上的最大速率，必然会使得链路中的一个或多个队列出现拥堵，导致 RTT 增大。我们可通过“统计 ITCP 流图 IRTT 图”（Statistics | TCPStream Graph | Round Trip Time Graph）进行观察（参见图 16-8）。

如图 16-8 所示，y 轴代表 RTT 估计值，以秒为单位。x 轴表示序列号。可以看到，在序列号约为 340000 处，RTT 开始减小。这和之前提到的发送暂停前最后发送的序列号相一致（344400）。RTT 的减小意味着发送方减慢发送速率，使得网络传输负载减轻（即数据发出速率大于新数据到达速率）。这样路由器队列逐渐为空，等待时间减小，RTT 也相应减小。TCP 处于 CWR 状态，发送速率会持续减小。最终，RTT 会减至绝对最小值约 17ms。通常，TCP 会避免这种情况的发生，因为它需要保持传输通道处于“满”的状态，以此确保充分利用可用的网络资源。

### 16.5.3 延伸 ACK 和本地拥塞恢复

在 8.3645 时刻，随着进入 CWR 状态 `cwnd` 逐渐减小，使得 TCP 传输速率更快地减小。从 8.362s 时刻的 ACK 可以得到在外数据值，`cwnd` 与这部分在外数据量的关系导致了发送速率的快速降低（图 16-9 中标记部分）。

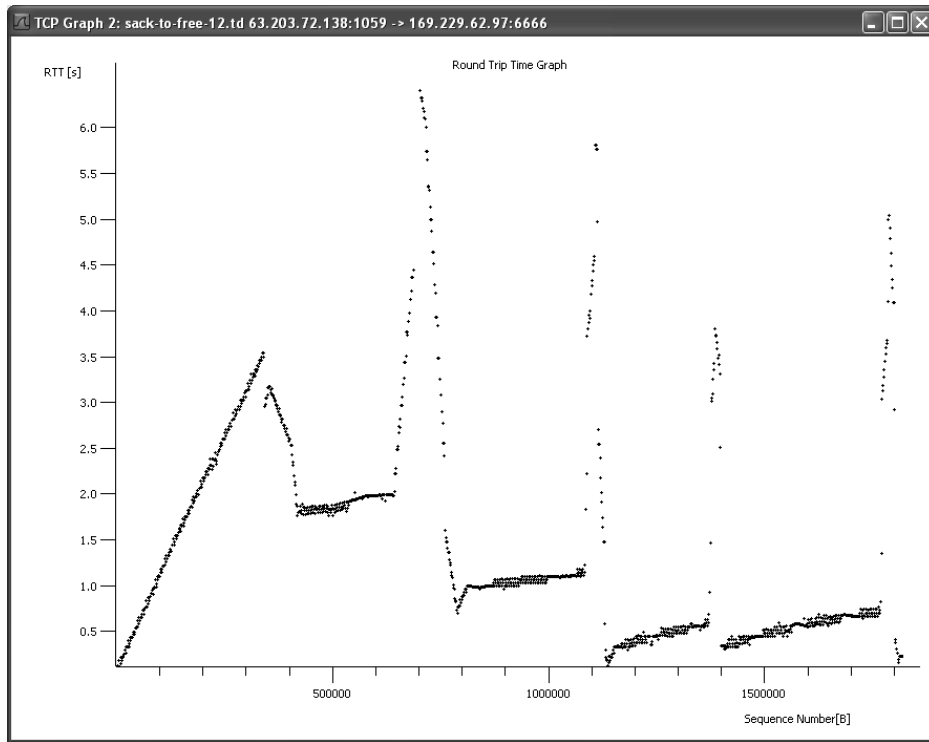


图 16.8: 发送端对  $RTT$  的估计值曲线。 $RTT$  增长阶段（图中稠密的增长值群组）对应于由过大的发送速率导致路由器缓存溢出的情况， $RTT$  降低阶段则表示发送端减慢发送速率，等待队列逐渐减小

8.362s 时刻的 ACK 号为 317801，而前一个 ACK 是 313601，所以这个 ACK 确认的数据为  $317801 - 313601 = 4200$  字节（3 个包）。这通常称为延伸 ACK（stretch ACK），即一个 ACK 确认两个最大段以上长度的数据。其形成原因有多种，最简单的就是 ACK 丢失。通常很难判断延伸 ACK 产生的确切原因，但这并不重要。在这个例子中，我们假设先前的 ACK 丢失。这个延伸 ACK 使得  $cwnd$  从 68 减为 66。

Linux 的 TCP 实现在每次接收 ACK 时，总是试图调整已发送但未经确认数据（记为在外数据，outstanding data）的估计值。（当它发送数据段后，也会根据之前提到的拥塞窗口校验算法去修改拥塞窗口，但这里并不生效。）在 CWR 阶段，如果在外数据包由于某种原因减少，如这里的接收延伸 ACK 后， $cwnd$  调整为在外数据估计值加 1。需要注意的是，CWR 通常只会在接收到每一对 ACK 后将  $cwnd$  的值减 1，因此这是额外的操作。通常情况下每接收一个 ACK， $cwnd$  会减小 1 或 0，然后  $cwnd$  被设置为  $\min(\text{在外数据值} + 1, [\text{也可能减少的}] cwnd)$ 。CWR 阶段一直持续到  $cwnd$  达到  $ssthresh$  或其他事件的发生，如丢包或重传。

在收到延伸 ACK 前的 8.258s 时刻，在外数据估计值为  $407\,401 + 1400 - 313601 = 95\,200$  字节（68 个包）。接收延伸 ACK 后，在外数据包为 65 个， $cwnd$  调整为 66。在 CWR 阶段，在外数据估计值和  $cwnd$  紧密相关。在这里出现了 ACK 延时的情况，导致每两个 ACK 到达  $cwnd$  值减 2，但只能发送一个新数据包。原因如下：假设在 ACK 到达前， $cwnd$  值  $co$ ，在外数据估计



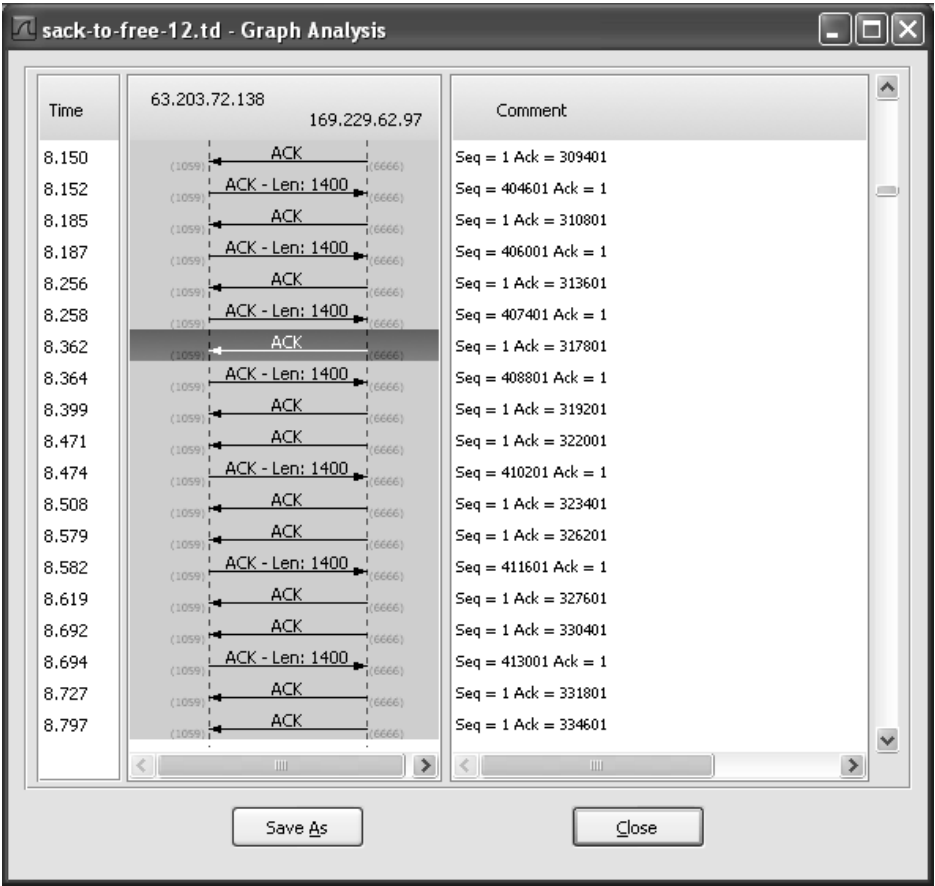


图 16.9: 一个“延伸 ACK”确认了三个数据包长度的数据。这种 ACK 可能使得发送端突发操作，当其他 ACK 在传输中丢失时，可能会出现延伸 ACK

值  $f6=co$ 。当第一个 ACK 到达（对一个包的确认）， $尤=f6-1$ ， $cwnd$  更新  $ci=\min (co-1,f+1)=co-1$ 。当第二个 ACK 到达（由于延时 ACK，因此为对两个包的确认）， $\frac{1}{2}=f-2=co-3$ ， $cwnd$  设  $cz=\min (Ci,5+1)=\min (Co-1,c0-2)=Co-2$ 。至此拥塞窗口减 2，但已有 3 个包被确认，因此在接收第二个 ACK 后，只可继续发送一个包。

在图 16-10 中，用圆圈标记出的数据包表明发送端结束 CWR，继续执行拥塞避免算法。图 16-11 更为详细地显示了这一行为。

发送端继续执行拥塞避免，逐渐达到相对稳定的吞吐量。然而，在 17.232s 时刻，开始形成严重的拥塞，致使 RTT 大幅增长。在图 16-8 中可以看到，在序列号 720000 处，RTT 约增至 6.5s—是稳定阶段（2s）的三倍多。这是大规模拥塞的常见现象。最终，严重的网络拥塞导致了丢包，TCP 发送端开始了首次重传。至 9.369s 时刻，发送端恢复正常模式，继续执行每接收一个 ACK 发送一个或两个新数据包操作

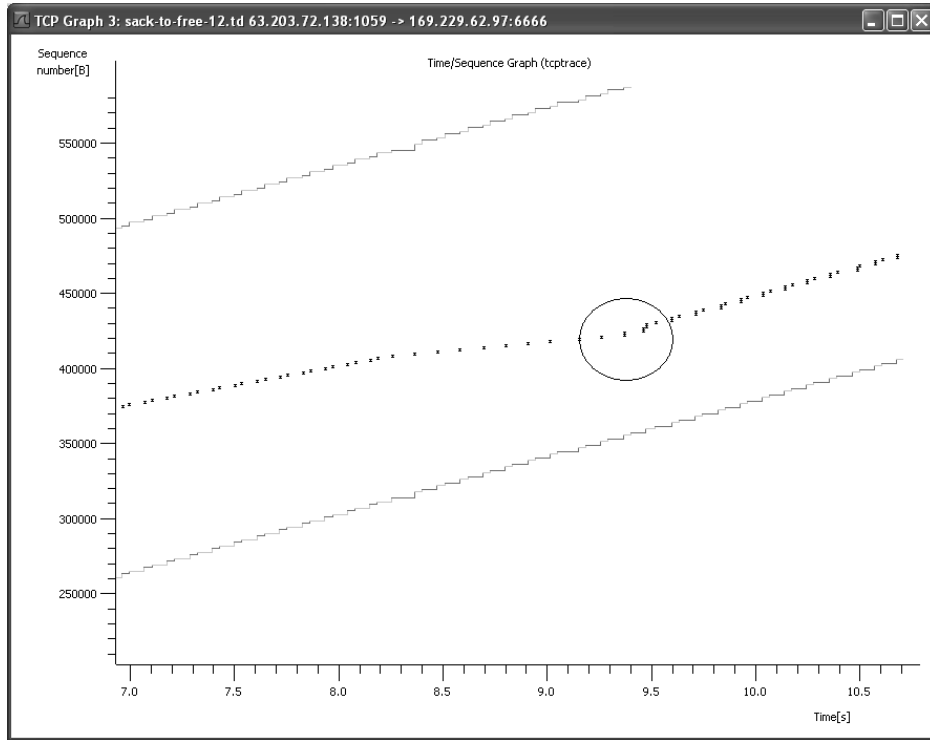


图 16.10: 在 9.37s 时刻,  $cwnd$  达到  $ssthresh$  为 49, 发送端结束 CWR 阶段。TCP 返回正常操作模式, 继续执行拥塞避免 (参见图 16-10 和图 16-11)

#### 16.5.4 快速重传和 SACK 恢复 (事件 2)

由于 RTT 大幅增长, 在 21.209s 时刻出现了首次重传。从图 16-12 可以清楚地看到, 首次重传 (图中圈出部分) 数据包的起始序列号为 690201, 对应接收到的最高 ACK 号 (也是 690201)。这次重传是由于接收到带有 SACK 块为 [698601, 700001] 的重复 ACK, 这里的数值区间是接收端成功接收的序列号区间, 表明这是对单个数据包的确认。

至 21.209s 时刻首次重传发生为止, 已发送数据的最大序列号为  $761601 + 1400 - 1 = 763000$ ,  $cwnd$  值 52。重传发生后,  $ssthresh$  值从 49 减为 26, TCP 进入恢复阶段, 一直持续至接收到序列号为 763000 (或更高) 的累积 ACK 为止。另外,  $cwnd$  也减为 (在外数据值 +1)。由于有些数据很可能已丢失, 因此并不能直接确定在外数据值, 需要利用下式:

$$flight_{size} = packets_{outstanding} + packets_{retransmitted} - packets_{removed} \quad (16.8)$$

等式右边的第一项表示经首次发送 (非重传) 但仍未收到 ACK 的数据; 第二项为重传但并未经 ACK 确认的数据; 最后一项则表示那些网络中已经不存在, 但还没有收到 ACK 的数据。由于 TCP 无法确认获知最后一项  $packets_{removed}$  值, 因此必须通过估算的方法。它包括已被接收 (失序) 的数据加上在传输中丢失的部分。利用 SACK 可以知道前半部分的值, 但丢包个数仍需要估算。

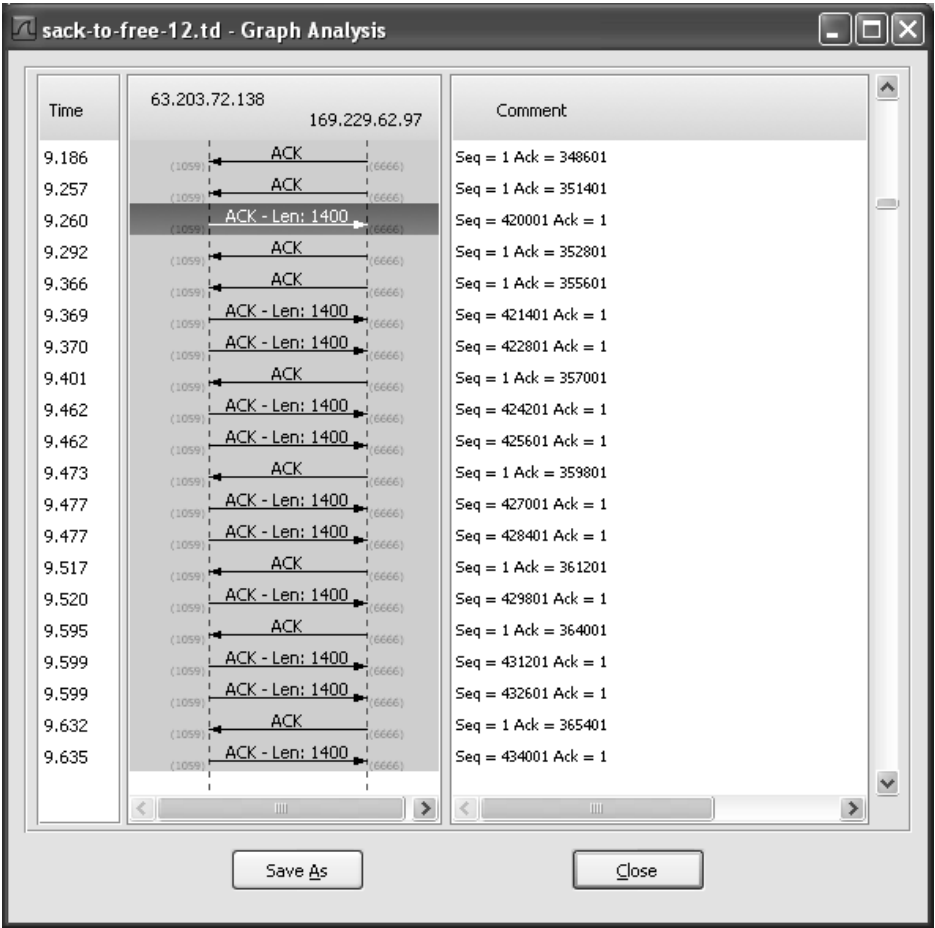


图 16.11: TCP 完成了恢复阶段并返回正常（拥塞避免）状态。每接收一个 ACK 就发送一个或两个新的数据包

packets\_outstanding 的计算值  $(763\,001 - 690\,201) / 1400 = 72800 / 1400 = 52$ 。根据 SACK 块的序列号区间，可以推算出已被接收缓存的包个数为  $(700\,001 - 698601) / 1400 = 1400 / 1400 = 1$ 。利用 FACK（这里默认启用），经 SACK 推算出的级存之间的空缺被认为是丢包。这里估算共有  $698\,601 - 690201 - 8400$ （6 个包）丢失。因此，flight size 的值为  $52 + 1 - (1 + 6) = 46$ ，相应地，cwnd 值设为 47。与 CWR 相似，在恢复阶段，每接收两个包的 ACK，cwnd 减 1。首次重传后，又发生了 7 次重传。接着在 21.2 21.7s 时间内，又开始了新数据的发送，相应的每个 ACK 都包含了 SACK 信息（参见图 16-13）。

图 16-13 去换了 Wireshark 中其他不相关信息，可以清楚地看到每个 ACK 包含的 SACK 信息。观察 SACK 的序列号区间（SLE 和 SRE），有两个常见值：[698601, 700001] 和 [702801, 763001]。前者表示只缓存了一个数据包（序列号范围为 698601 700001），后者则增加至 43 个包（序列号范围为 702801 763001）。通常 CWR 阶段的速率波半算法每接收两个包，cwnd 至少减 1。这里每个 ACK 是对一个包的确认，flight size 相应减 1，因此可以发送一个新的数据包。注意这里和 CWR 状态的区别，CWR 情况下，每个 ACK 提供对两个包的确认，而这里一个 ACK

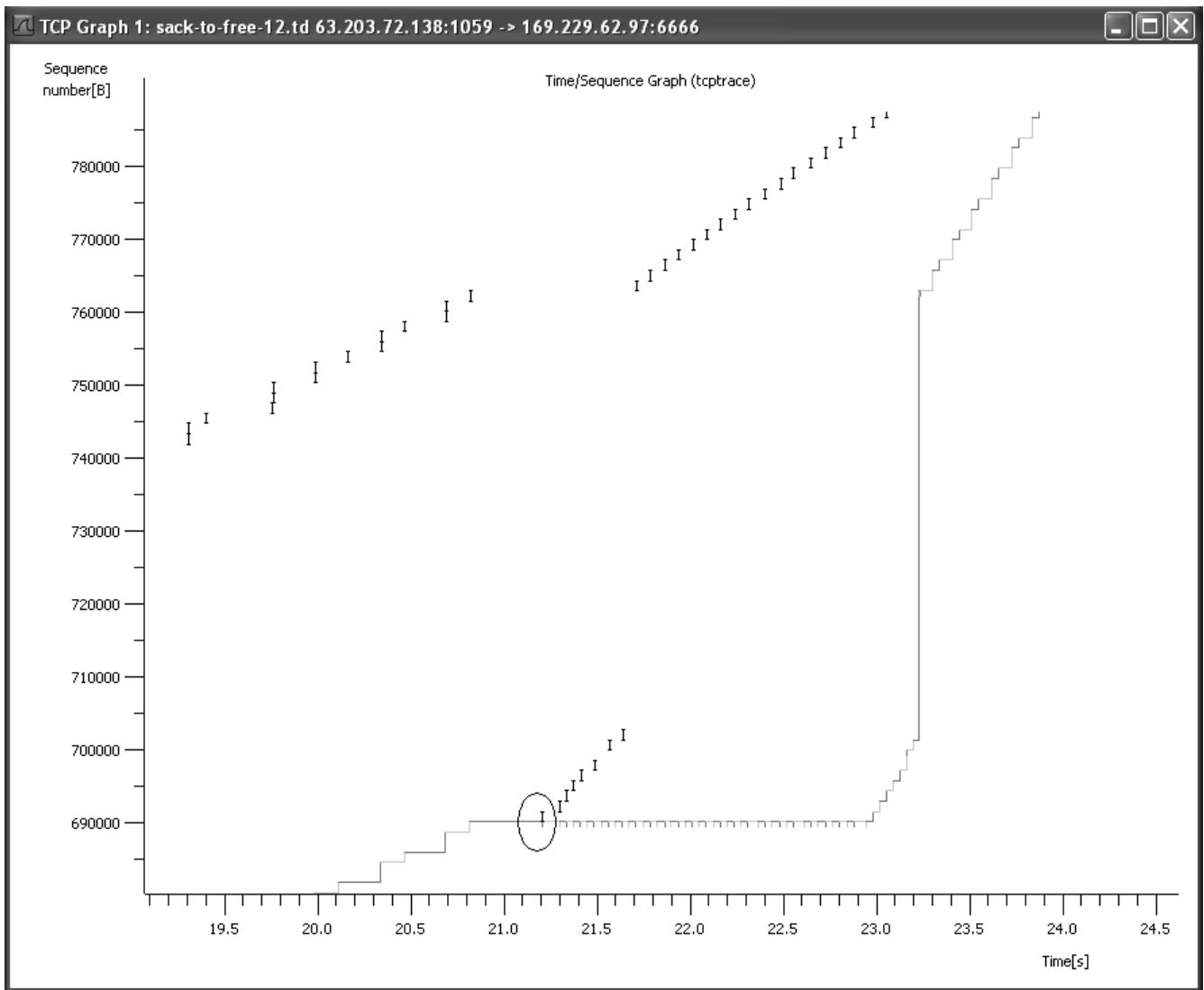


图 16.12: 首次重传(已圈出)发生在 21.209s 时刻。SACK 块用于告知发送方发送哪些数据包。在 21.0~22.0s 时间内, 共出现了 8 次重传

只确认一个包, 因此不论是否为重传包, 每接收一个 ACK,  $cvnd$  减 1。在整个恢复期间,  $cwnd$  从 47 减为 20。

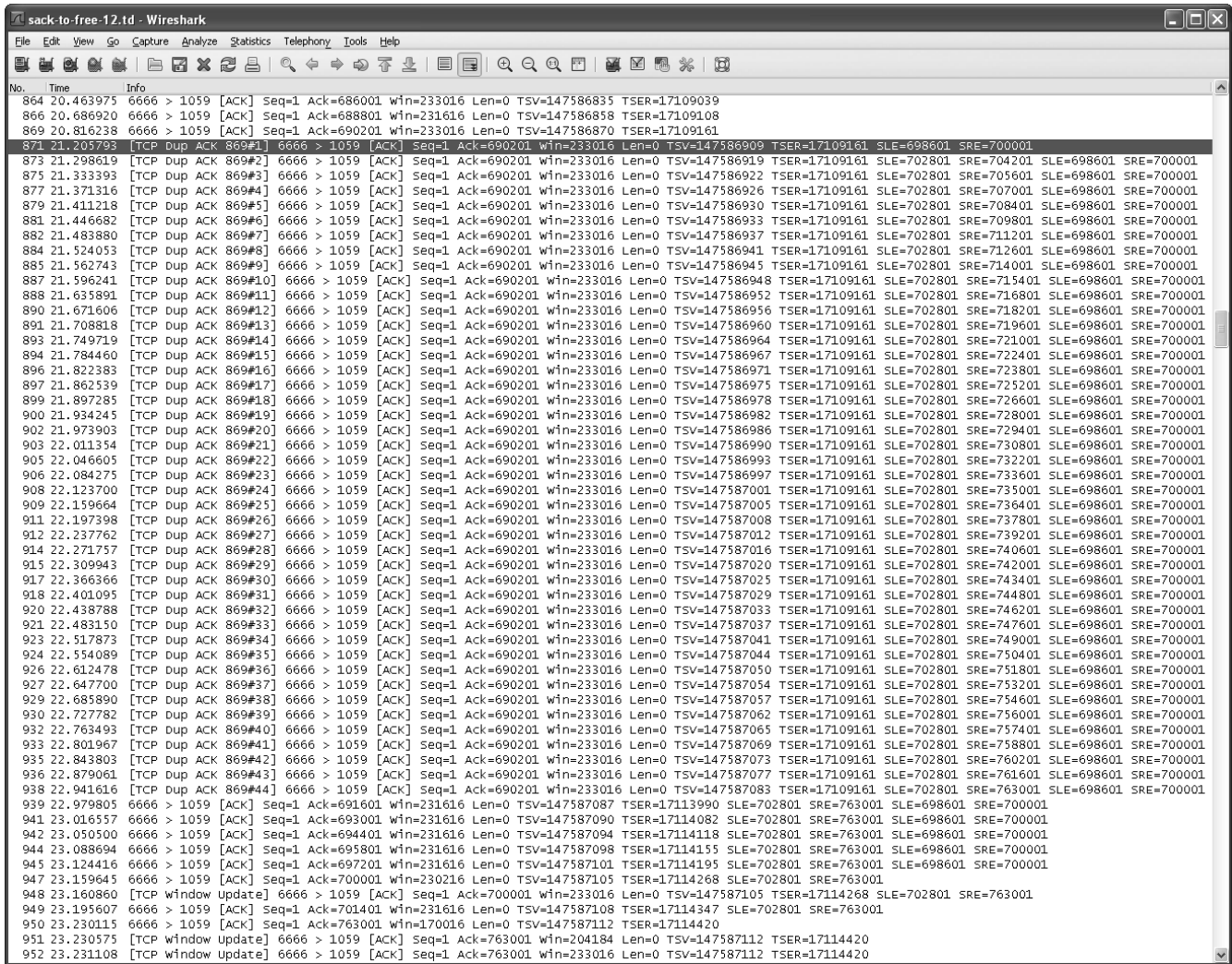
Wireshark 显示, 大多数包含 SACK 的 ACK 是序列号为 690201 的重复 ACK (其中 44 个)。有 5 个好 ACK 为包含 SACK 块 [702801, 763001] 和 [698601, 700001] 的非重复 ACK。还有两个 ACK 只包含了 [1702801, 763001] SACK 块。这些 ACK 并不能使发送端结束恢复状态, 因为 ACK 号低于之前提到的 763000 (恢复点)。它们属于我们在前面讨论过的局部 ACK。

23.301s 时刻接收了序列号为 765801 (大于前面提到的 763000) 的好 ACK, 表明已经达到恢复点, 此时的  $cwnd$  为 20,  $ssthresh$  值为 26, 说明 TCP 正处于慢启动状态。又经历几轮传输后, 到 23.659s 时刻,  $cwnd$  达到 27, TCP 恢复正常操作, 继续执行拥塞避免算法。至此首个快

速重传恢复阶段完成。

### 16.5.5 其他本地拥塞和快速重传事件

下面再次发生了本地拥塞、快速重传和其他两个本地拥塞相关事件。它们和之前的相关内容有相似的地方，这里只是进行简要概括。



| No. | Time      | Info   |
|-----|-----------|--|
| 864 | 20.463975 | 6666 > 1059 [ACK] Seq=1 Ack=686001 win=233016 Len=0 TSV=147586835 TSER=17109039  |
| 866 | 20.686920 | 6666 > 1059 [ACK] Seq=1 Ack=688801 win=231616 Len=0 TSV=147586858 TSER=17109108  |
| 869 | 20.816238 | 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586870 TSER=17109161  |
| 871 | 21.205793 | [TCP Dup ACK 869#1] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586909 TSER=17109161 SLE=698601 SRE=700001                        |
| 873 | 21.298619 | [TCP Dup ACK 869#2] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586919 TSER=17109161 SLE=702801 SRE=704201 SLE=698601 SRE=700001  |
| 875 | 21.333393 | [TCP Dup ACK 869#3] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586922 TSER=17109161 SLE=702801 SRE=705601 SLE=698601 SRE=700001  |
| 877 | 21.371316 | [TCP Dup ACK 869#4] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586926 TSER=17109161 SLE=702801 SRE=707001 SLE=698601 SRE=700001  |
| 879 | 21.411218 | [TCP Dup ACK 869#5] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586930 TSER=17109161 SLE=702801 SRE=708401 SLE=698601 SRE=700001  |
| 881 | 21.446682 | [TCP Dup ACK 869#6] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586933 TSER=17109161 SLE=702801 SRE=709801 SLE=698601 SRE=700001  |
| 882 | 21.483880 | [TCP Dup ACK 869#7] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586937 TSER=17109161 SLE=702801 SRE=711201 SLE=698601 SRE=700001  |
| 884 | 21.524053 | [TCP Dup ACK 869#8] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586941 TSER=17109161 SLE=702801 SRE=712601 SLE=698601 SRE=700001  |
| 885 | 21.562743 | [TCP Dup ACK 869#9] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586945 TSER=17109161 SLE=702801 SRE=714001 SLE=698601 SRE=700001  |
| 887 | 21.596241 | [TCP Dup ACK 869#10] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586948 TSER=17109161 SLE=702801 SRE=715401 SLE=698601 SRE=700001 |
| 888 | 21.635891 | [TCP Dup ACK 869#11] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586952 TSER=17109161 SLE=702801 SRE=716801 SLE=698601 SRE=700001 |
| 890 | 21.671606 | [TCP Dup ACK 869#12] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586956 TSER=17109161 SLE=702801 SRE=718201 SLE=698601 SRE=700001 |
| 891 | 21.708818 | [TCP Dup ACK 869#13] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586960 TSER=17109161 SLE=702801 SRE=719601 SLE=698601 SRE=700001 |
| 893 | 21.749719 | [TCP Dup ACK 869#14] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586964 TSER=17109161 SLE=702801 SRE=721001 SLE=698601 SRE=700001 |
| 894 | 21.784460 | [TCP Dup ACK 869#15] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586967 TSER=17109161 SLE=702801 SRE=722401 SLE=698601 SRE=700001 |
| 896 | 21.822383 | [TCP Dup ACK 869#16] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586971 TSER=17109161 SLE=702801 SRE=723801 SLE=698601 SRE=700001 |
| 897 | 21.862539 | [TCP Dup ACK 869#17] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586975 TSER=17109161 SLE=702801 SRE=725201 SLE=698601 SRE=700001 |
| 899 | 21.897285 | [TCP Dup ACK 869#18] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586978 TSER=17109161 SLE=702801 SRE=726601 SLE=698601 SRE=700001 |
| 900 | 21.934245 | [TCP Dup ACK 869#19] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586982 TSER=17109161 SLE=702801 SRE=728001 SLE=698601 SRE=700001 |
| 902 | 21.973903 | [TCP Dup ACK 869#20] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586986 TSER=17109161 SLE=702801 SRE=729401 SLE=698601 SRE=700001 |
| 903 | 22.011354 | [TCP Dup ACK 869#21] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586990 TSER=17109161 SLE=702801 SRE=730801 SLE=698601 SRE=700001 |
| 905 | 22.046605 | [TCP Dup ACK 869#22] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586993 TSER=17109161 SLE=702801 SRE=732201 SLE=698601 SRE=700001 |
| 906 | 22.084275 | [TCP Dup ACK 869#23] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147586997 TSER=17109161 SLE=702801 SRE=733601 SLE=698601 SRE=700001 |
| 908 | 22.123700 | [TCP Dup ACK 869#24] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587001 TSER=17109161 SLE=702801 SRE=735001 SLE=698601 SRE=700001 |
| 909 | 22.159664 | [TCP Dup ACK 869#25] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587005 TSER=17109161 SLE=702801 SRE=736401 SLE=698601 SRE=700001 |
| 911 | 22.197398 | [TCP Dup ACK 869#26] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587008 TSER=17109161 SLE=702801 SRE=737801 SLE=698601 SRE=700001 |
| 912 | 22.237762 | [TCP Dup ACK 869#27] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587012 TSER=17109161 SLE=702801 SRE=739201 SLE=698601 SRE=700001 |
| 914 | 22.271757 | [TCP Dup ACK 869#28] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587016 TSER=17109161 SLE=702801 SRE=740601 SLE=698601 SRE=700001 |
| 915 | 22.309943 | [TCP Dup ACK 869#29] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587020 TSER=17109161 SLE=702801 SRE=742001 SLE=698601 SRE=700001 |
| 917 | 22.366366 | [TCP Dup ACK 869#30] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587025 TSER=17109161 SLE=702801 SRE=743401 SLE=698601 SRE=700001 |
| 918 | 22.401095 | [TCP Dup ACK 869#31] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587029 TSER=17109161 SLE=702801 SRE=744801 SLE=698601 SRE=700001 |
| 920 | 22.438788 | [TCP Dup ACK 869#32] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587033 TSER=17109161 SLE=702801 SRE=746201 SLE=698601 SRE=700001 |
| 921 | 22.483150 | [TCP Dup ACK 869#33] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587037 TSER=17109161 SLE=702801 SRE=747601 SLE=698601 SRE=700001 |
| 923 | 22.517873 | [TCP Dup ACK 869#34] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587041 TSER=17109161 SLE=702801 SRE=749001 SLE=698601 SRE=700001 |
| 924 | 22.554089 | [TCP Dup ACK 869#35] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587044 TSER=17109161 SLE=702801 SRE=750401 SLE=698601 SRE=700001 |
| 926 | 22.612478 | [TCP Dup ACK 869#36] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587050 TSER=17109161 SLE=702801 SRE=751801 SLE=698601 SRE=700001 |
| 927 | 22.647700 | [TCP Dup ACK 869#37] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587054 TSER=17109161 SLE=702801 SRE=753201 SLE=698601 SRE=700001 |
| 929 | 22.685890 | [TCP Dup ACK 869#38] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587057 TSER=17109161 SLE=702801 SRE=754601 SLE=698601 SRE=700001 |
| 930 | 22.727782 | [TCP Dup ACK 869#39] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587062 TSER=17109161 SLE=702801 SRE=756001 SLE=698601 SRE=700001 |
| 932 | 22.763493 | [TCP Dup ACK 869#40] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587065 TSER=17109161 SLE=702801 SRE=757401 SLE=698601 SRE=700001 |
| 933 | 22.801967 | [TCP Dup ACK 869#41] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587069 TSER=17109161 SLE=702801 SRE=758801 SLE=698601 SRE=700001 |
| 935 | 22.843803 | [TCP Dup ACK 869#42] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587073 TSER=17109161 SLE=702801 SRE=760201 SLE=698601 SRE=700001 |
| 936 | 22.879061 | [TCP Dup ACK 869#43] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587077 TSER=17109161 SLE=702801 SRE=761601 SLE=698601 SRE=700001 |
| 938 | 22.941616 | [TCP Dup ACK 869#44] 6666 > 1059 [ACK] Seq=1 Ack=690201 win=233016 Len=0 TSV=147587083 TSER=17109161 SLE=702801 SRE=763001 SLE=698601 SRE=700001 |
| 939 | 22.979805 | 6666 > 1059 [ACK] Seq=1 Ack=691601 win=231616 Len=0 TSV=147587087 TSER=17113990 SLE=702801 SRE=763001 SLE=698601 SRE=700001                      |
| 941 | 23.016557 | 6666 > 1059 [ACK] Seq=1 Ack=693001 win=231616 Len=0 TSV=147587090 TSER=17114082 SLE=702801 SRE=763001 SLE=698601 SRE=700001                      |
| 942 | 23.050500 | 6666 > 1059 [ACK] Seq=1 Ack=694401 win=231616 Len=0 TSV=147587094 TSER=17114118 SLE=702801 SRE=763001 SLE=698601 SRE=700001                      |
| 944 | 23.088694 | 6666 > 1059 [ACK] Seq=1 Ack=695801 win=231616 Len=0 TSV=147587098 TSER=17114155 SLE=702801 SRE=763001 SLE=698601 SRE=700001                      |
| 945 | 23.124416 | 6666 > 1059 [ACK] Seq=1 Ack=697201 win=231616 Len=0 TSV=147587101 TSER=17114195 SLE=702801 SRE=763001 SLE=698601 SRE=700001                      |
| 947 | 23.159645 | 6666 > 1059 [ACK] Seq=1 Ack=700001 win=230216 Len=0 TSV=147587105 TSER=17114268 SLE=702801 SRE=763001  |
| 948 | 23.160860 | [TCP window update] 6666 > 1059 [ACK] Seq=1 Ack=700001 win=233016 Len=0 TSV=147587105 TSER=17114268 SLE=702801 SRE=763001                        |
| 949 | 23.195607 | 6666 > 1059 [ACK] Seq=1 Ack=701401 win=231616 Len=0 TSV=147587108 TSER=17114347 SLE=702801 SRE=763001  |
| 950 | 23.230115 | 6666 > 1059 [ACK] Seq=1 Ack=763001 win=170016 Len=0 TSV=147587112 TSER=17114420  |
| 951 | 23.230575 | [TCP window update] 6666 > 1059 [ACK] Seq=1 Ack=763001 win=204184 Len=0 TSV=147587112 TSER=17114420  |
| 952 | 23.231108 | [TCP window update] 6666 > 1059 [ACK] Seq=1 Ack=763001 win=233016 Len=0 TSV=147587112 TSER=17114420  |

图 16.13: 首次重传（已圈出）发生在 21.209s 时刻

### 再次 CWR（事件 3）

在 30.745s 时刻，由于本地拥塞再次出现了 CWR 事件。此时在外数据值为 1 090601+1400-1 051 401 = 4060029 个包），cwnd 为 31。按理可再发送两个包，但由于出现本地拥塞，导致实际

并未发送。在这种情况下, cwnd 被设置为在外数据值 + 1 = 30, ssthresh 减为 15。在 34.759s 时刻, 在 RTT 再次出现大幅增长后, cwnd 降为 ssthresh 时, TCP 退出 CWR 状态。

## 二次快速重传 (事件 4)

在 36.914s 时刻, cwnd = 16, 再次出现快速重传。利用 Wireshark 的基本显示功能, 可以清楚地看到重传 (参见图 16-14)。在 36.878s 时刻, 接收了一个带 SACK 块 [1117201, 1118601] 的

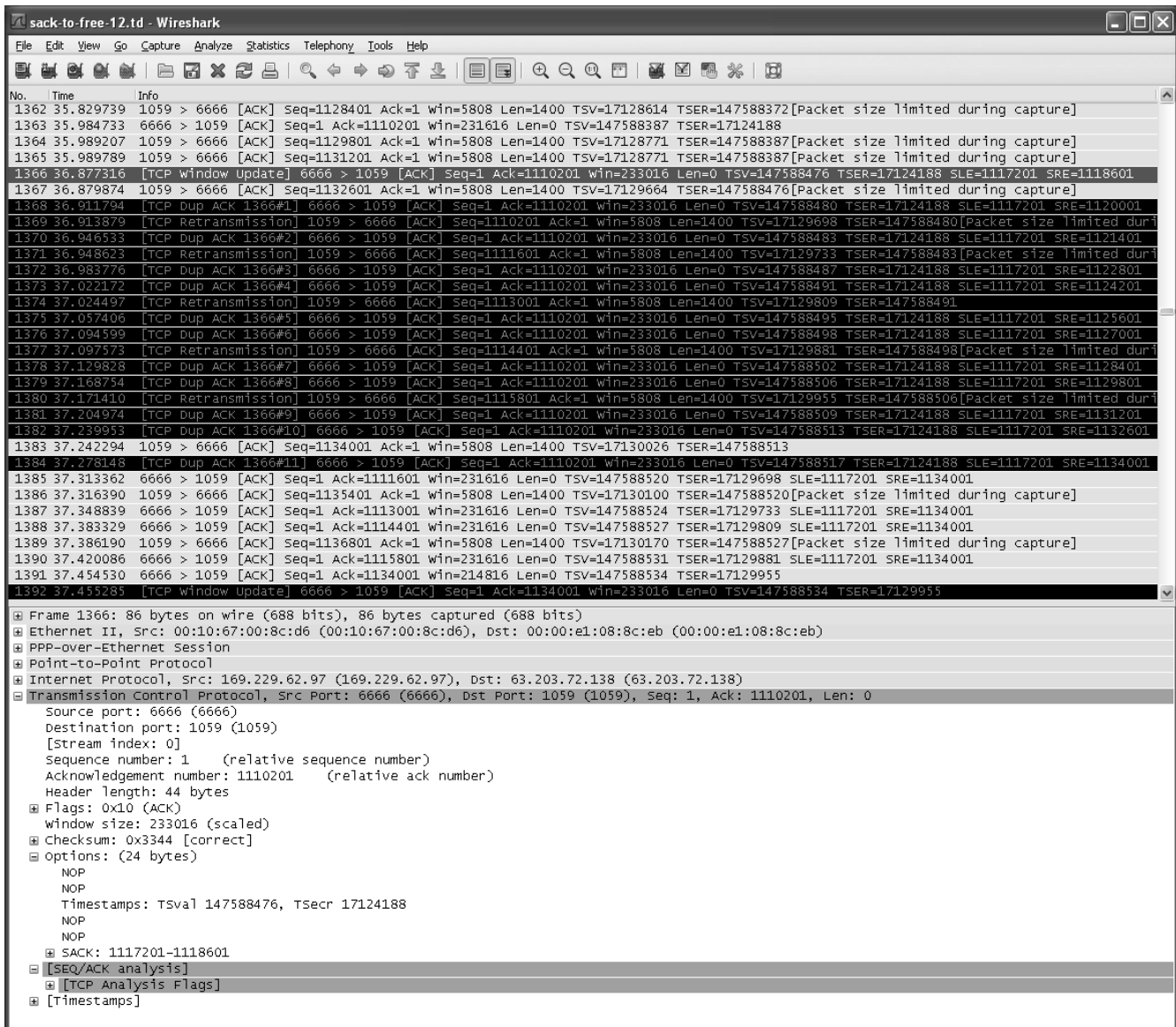


图 16.14: linux

ACK (1366 号包, ACK 号为 1110201)。这使得 Linux TCP 进入了失序状态, 到达一个 ACK 就能触发一个新数据包的传输 (和限制传输相似), 1367 号包就是经该 ACK 触发发送的。

一旦接收到一个重复 ACK 或一个带 SACK 信息的 ACK, Linux TCP 发送端会进入失序 (Disorder) 状态。在此状态下, 数据包的到达会触发新数据的传输。之后再次接收重复 (或带 SACK 信息的) ACK, 就会进入恢复 (Recovery) 状态, 并开始重传

36.912s 时刻接收了包含 SACK 块 [1117201,1120001] 的重复 ACK (1368 号包), 因此 TCP 进入恢复阶段, 并在 36.914s 时刻触发快速重传 (1369 号包)。至此已发送数据的最大序列号为 1132601+1400 1 113400。随者 37435 日时刻序列号为 1134001 的 ACK (1331 号包) 到达, 恢复阶段结束。注意到紧随这个 ACK 的是一次窗口更新。对于批量数据传输来说, 接收窗口相对于网络带宽延迟积较大, 因此这样的窗口更新通常不是很重要。但在交互式传输、接收窗口较小或者很少从网络中读数据的传输中, 这些更新就相当重要 (第 15 章已经提到)。当 36.914s 时刻的第一次快速重传开始, ssthresh 从 16 减为 8。到 37.455s 时刻恢复阶段完成, cwnd =4, ssthresh =8。由于 cwnd 小于 ssthresh, 发送端进入慢启动状态。

### 再次 CWR (事件 5 和事件 6)

随着 43.356s 时刻序列号为 1359401 的 ACK 的到达, 由于本地拥塞致使后续数据包无法发送, TCP 再次进入 CWR 状态。这使得 ssthresh 减为 8, cwnd 变为 15。在 CWR 状态的再次传输失败, 致使 ssthresh 变为 12。最终结束 CWR 时, cwnd=7, ssthresh =8。

另一次的本地拥塞发生在 59.6525 时刻, 此时的 cwnd = 19, ssthresh = 10, 导致 TCP 再次进入 CWR 状态。这次出现了超时, 致使 TCP 由 CWR 状态进入丢失 (Loss) 状态。这是我们需要讨论的新的事件类型。

### 16.5.6 超时、重传和撤销 cwnd 修改

TCP 设置了超时计时器, 用于快速重传中出现丢包的情况。至此我们还没有看到重传超时的发生, 这从一定角度说是好事, 因为一旦超时发生, 就意味着网络中出现了严重的拥塞, 性能极差。在下面的传输过程中, 如图 16-15 所示, 我们将看到重传计时器超时后 TCP 的处理操作。

发送端经历了首次超时, 其 RTO=1.57S。在这里, 发送端认为这是一次伪超时, 并撤销了对拥塞控制变量的变更

### 首次超时 (事件 7)

62.486s 时刻出现了一次重传 (2157 号包), 序列号为 1773801 (图 16-15 中标记部分)、在此之前, 并没有重复 ACK 或 SACK 信息。如图 16-15 所示, 在 62.486s 时刻, 至上个 ACK 到达已过了 1.58s, 但根据图 16-8, 此时的 RTT 估计值只有 800ms。因此, 我们认为发生了重传超时。TCP 进入丢失 (Loss) 状态, cwnd 减小为 1, ssthresh 设为 5, 进入慢启动状态。超时也使得之前保存的 SACK 信息被丢弃。然而接收端仍会返回 SACK 信息, 因此对新接收的 SACK 可以继续使用。

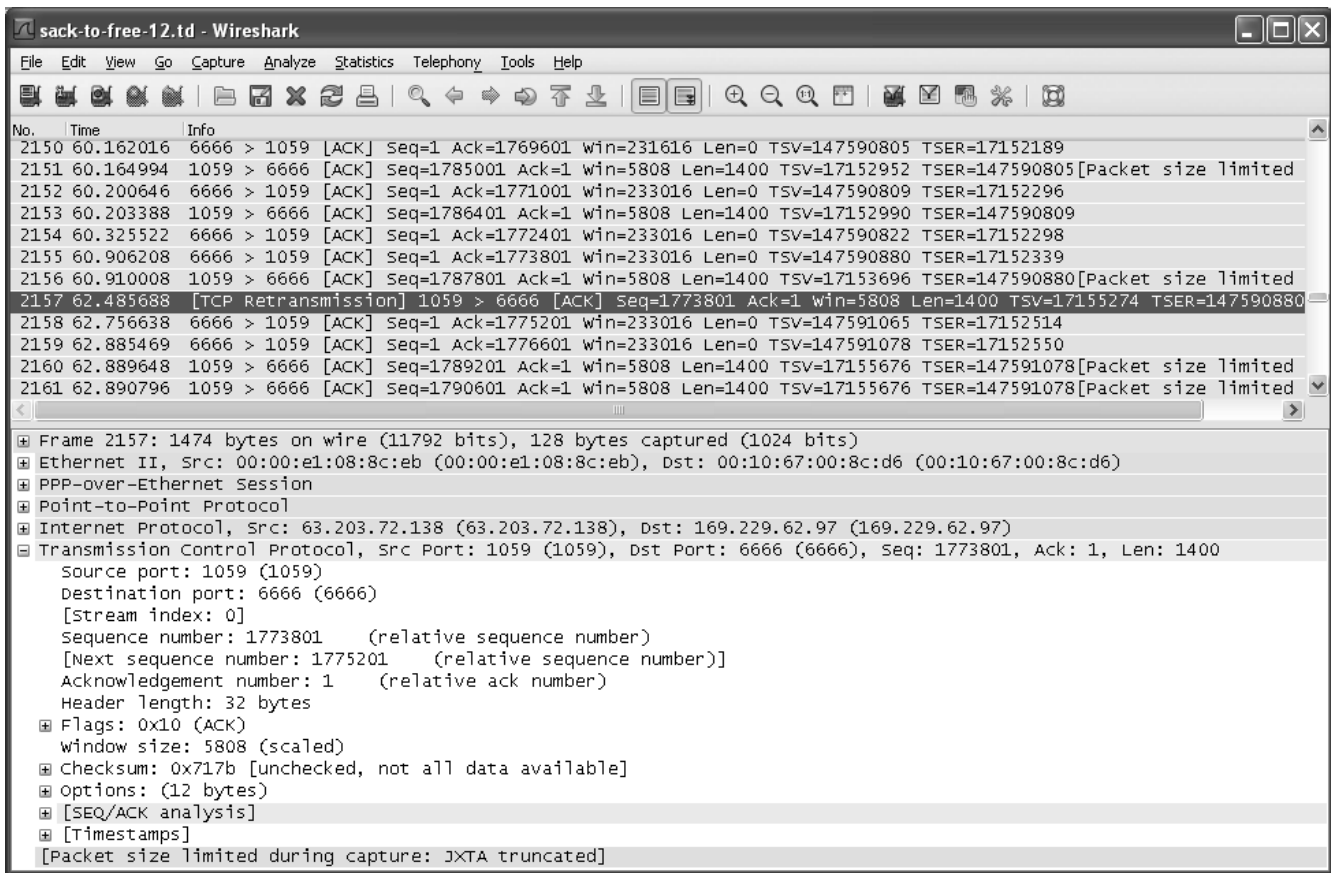


图 16.15: linux

当经历超时后，TCP 应当“忘记”之前的 SACK 信息，原因在于接收端可能会改变它之前发出的 SACK 信息。根据 [RFC2018]，当接收端需要调整其缓存时，可能将之前存储的失序数据删除。尽管并不常见，但这种行为是允许的。当接收端需要整理缓存时，只有第一个 SACK 信息中最近接收到的数据块不会删除。其他的 SACK 信息都不再可信。

然而有趣的是，这里的拥塞操作都撤销了。之前已经提过，当 TCP 认为重传超时出错会执行 Eifel 响应算法。在此处可以凭借时间戳的证据来判定出错。62.752s 时刻接收的序列号为 1775201 的 ACK (2158 号包) 携带了一个 TSOPT (时间戳选项)，其 TSV (时间戳值) 为 17152514，而重传包的 TSV 为 17155274。由于 ACK 的 TSER (时间戳回显重试) 字段包含了重传的数据段，并且早于该重传包，因此认为此处的重传是无用的，接收方在重传前已经接收到了该数据段。所以重传超时也是无效的。

由于超时出错，TCP 触发了 Eifel 响应算法，恢复了 cwnd 和 ssthresh 的值，并转为正常状态，继续执行拥塞避免算法。



### 快速重传（事件 8）

在 67.510s 时刻，接收了一个序列号 1789201 的重复 ACK（2179 号包），包含 SACK 块 [1792001,1793401]，因此 TCP 再次进入失序（Disorder）状态。至此已发送数据的最大序列号为 1806000。再次到来的 SACK 信息使得 TCP 进入恢复（Recovery）状态，并在 67.550s 时刻开始了序列号为 1789201 的又一次快速重传（2182 号包）。这使得 ssthresh 减至 5，cwnd 也相应开始减小。随着 67.916s 时刻序列号为 1806001 的 ACK（2197 号包）到达，恢复阶段结束。

### 再次 CWR（事件 9）

在 77.121s 时刻，又一次出现了本地拥塞事件，此时的  $cwnd = 18$ 。这使得 ssthresh 被置为 9 并再次进入 CWR 状态。然而，由于出现超时，这次 CWR 中 cwnd 的减值过程被中断，cwnd 只减小了 1，最终值为 8。

### 再次超时（事件 10）

再次超时又引发了新一轮的重传，在 78.515s 时刻又发送了序列号为 2175601 的包（图中未标记）。cwnd 更新为 1，ssthresh 仍 9，重传数据段的 TSOPT TSV 值 17171306。80.093s 时刻到达的序列号 2179801 的 ACK（2641 号包）的 TSOPT TSER 值 17169948，和事件 7 的超时一样，拥塞操作也被撤销了。此时 flight size 估计值为  $2\ 184\ 001 + 1400 - 2179\ 801 = 5600$  字节（4 个包），由于拥塞操作撤销，因此 cwnd 仍为 8，这样将允许再发送 4 个新数据包。但这样的突发操作可能会造成丢包，所以应避免发送。

为防止这种突发行为，Linux TCP 实现了拥塞窗口调整（congestion window moderation）机制。它将单个 ACK 能触发的新数据包发送个数限制为最大突出值（maxburst），这里取值为 3。因此，cwnd 被设置为（在外数据值 + 最大突出值） $= 4 + 3 = 7$ 。拥塞窗口调整机制和 TCP 中的相关方法一致，并经网络仿真工具 NS-2 验证。NS-2 在开发和探讨新的 TCP 算法研究中被广泛使用。

### 超时和最后一次恢复（事件 11）

如图 16-16 所示，在 88.929s 时刻出现了重传计时器超时，引发了序列号为 2185401 数据包的重传。这次超时使得发送端进入慢启动状态，ssthresh=5。这次 TCP 不能撤销超时，因此 cwnd 被置为 1，继续执行慢启动。从下面的传输流记录图可以更清楚地观察（参见图 16-17）。序列号为 2185401 的重传已在图中标出。在重传后，与连接建立初始一样，开始了慢启动操作。根据每个到达 ACK 确认的数据包个数，继续发送两个或三个新数据包。直到 89.4345s，cwnd 达到 ssthresh 的值（5），TCP 继续执行拥塞避免。

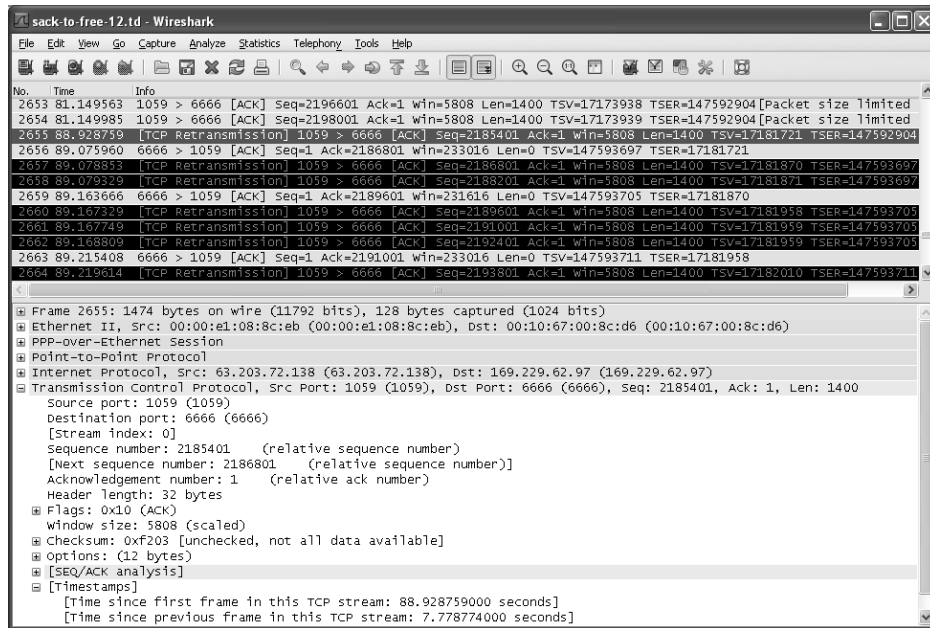


图 16.16: linux

### 16.5.7 连接结束

最后一次包交换传输中，首先发送方在 99.757s 时刻发送了一个 FIN 包。接着接收方返回了 13 个 ACK 和一个 FIN 包。在 100.476s 时刻发送了最后一个包（即最后的 ACK）。该交换过程参见图 16-18。

传输的最大序列号为  $2620801 + 640 - 1 = 2621440$ ，等于总共传输数据大小 2.5MB。在 99.757s 时刻，在外数据量为  $(2619\ 401 + 1400 - 2\ 594\ 201) / 1400 + 1 = 20$  个包。到达的 13 个 ACK（其中 7 个是对两个包的确认）完成了全部  $(2 * 7) + (13 - 7) = 20$  个包的确认。注意到 100.472s 时刻达到的 ACK 确认的两个数据包长度分别为 1400 和 640，分别对应  $2\ 621\ 442 - 2619\ 401 = 1400 + 640$ 。

这个扩展示例涉及了我们之前讨论过的大部分算法，包括基本 TCP 算法（慢启动、拥塞避免）以及选择确认、速率减半，包括一些比较新的方法如伪 RTO 检测等。下面我们将讨论一些算法的改进，以及那些不太普遍但更具理论性或者更新的方法。Linux TCP 协议栈实现了很多这样的方法，但不是默认启用的。通常只要利用 `sysctl` 程序稍加修改就能使用。Windows 的一些较新版本（Windows Vista 及以后版本）也都实现了这些功能的改进。

## 16.6 共享拥塞状态信息

前面的讨论和举例都是针对单一的 TCP 连接的拥塞处理操作。然而，相同的主机之间随后可能建立新的连接，这些新连接也需要重新进行拥塞处理，建立自己的 `ssthresh` 和 `cwnd` 值。在许多情况下，新连接可能会用到相同主机之间的其他连接的信息，包括已关闭的连接或者正处

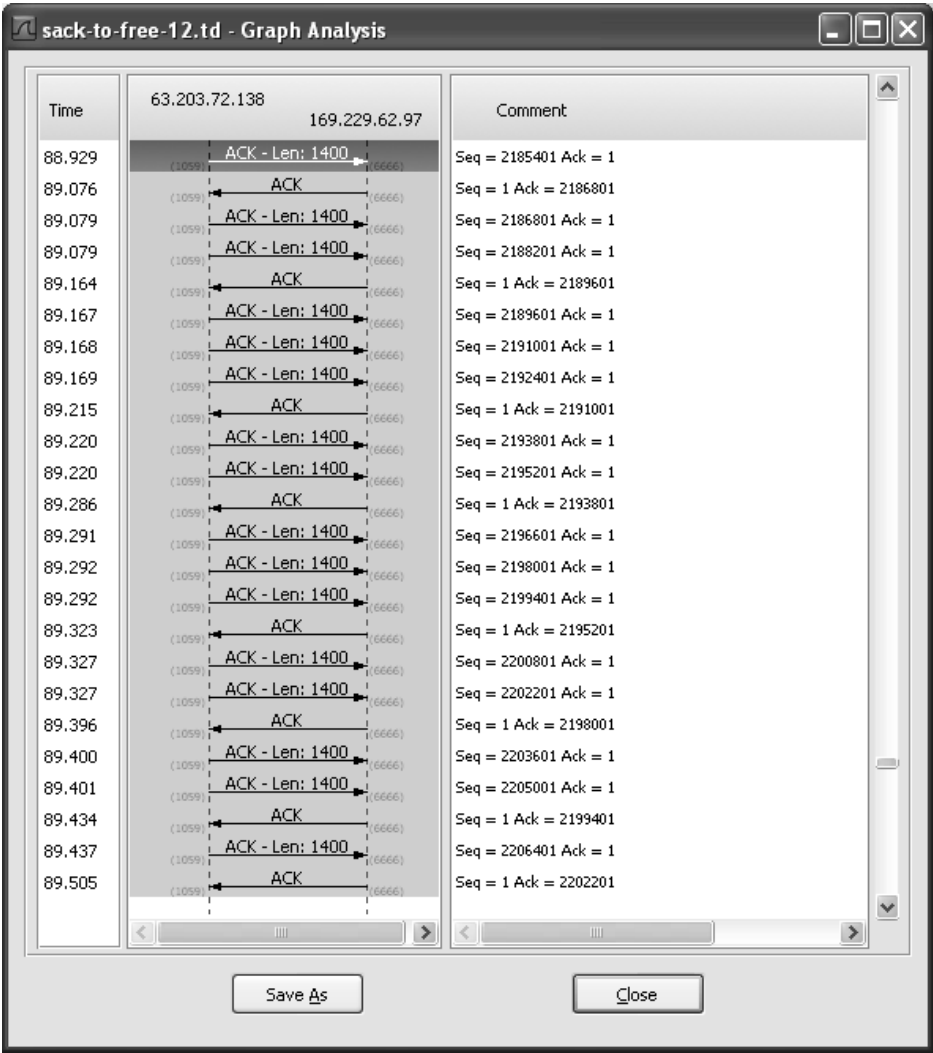


图 16.17: linux

于活动状态的其他连接。这就是相同主机间多个连接共享拥塞状态信息。之前的一篇名为“TCP 控制块相互依赖性”的文章 [RFC2140] 描述了相关内容，其中注意区分了暂时共享（temporal sharing，新连接与已关闭连接间的信息共享）和总体共享（ensemble sharing，新连接与其他活动连接间的信息共享）。

为将上述思想形成除 TCP 外的新的应用协议，[RFC3124] 提出了拥塞管理（Congestion-Manager）机制。该机制使得本地操作系统可实现相关协议来了解链路状态信息，如丢包率、拥塞估计、RTT 等。

Linux 在包含路由信息的子系统中实现了上述思想，即第 15 章已经提到的目的度量。这些度量默认开启（在前面的扩展示例中，我们通过设置 sysctl 变量 `net.ipv4.tcp_no_metrics_save` 为 1 禁用了该项功能）。当一个 TCP 连接关闭前，需要保存以下信息：RTT 测量值（包括 `srtt` 和 `rttvar`）、重排估计值以及拥塞控制变量 `cwnd` 和 `ssthresh`。当相同主机间的新连接建立时，就

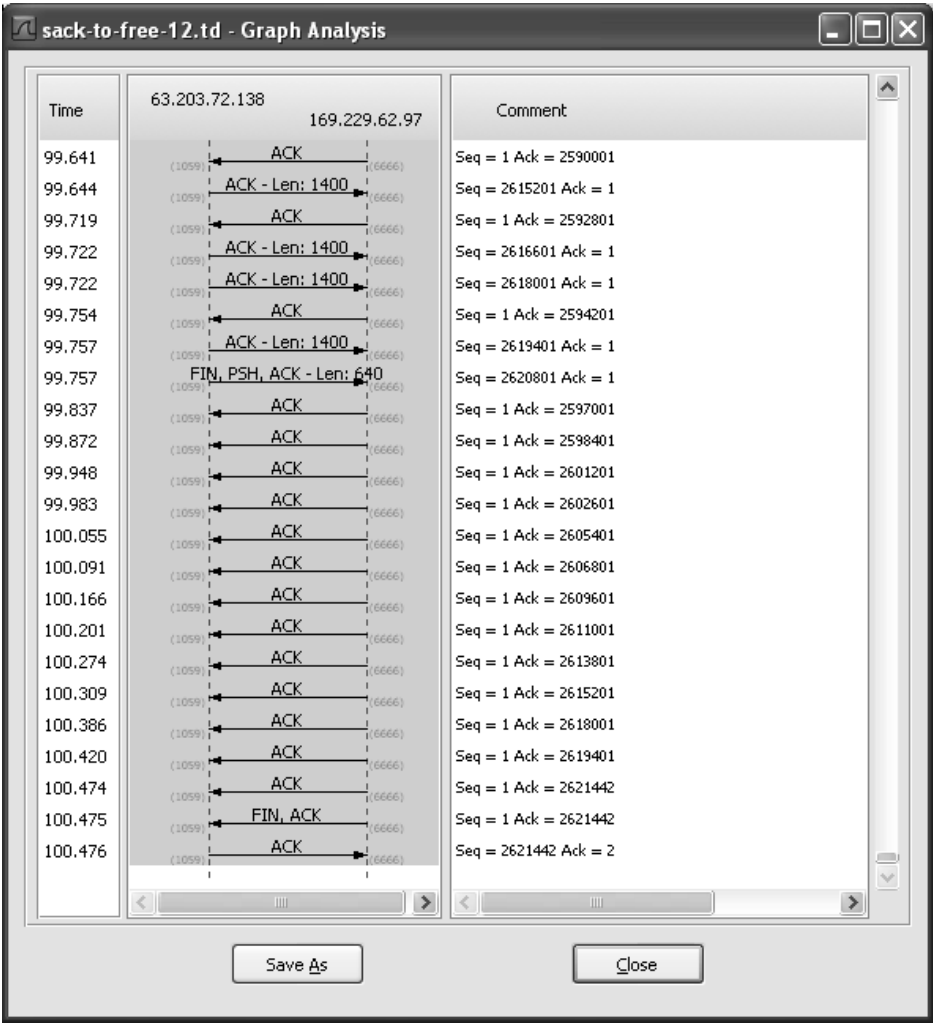


图 16.18: 在连接关闭过程中，接收端返回了 13 个纯 ACK 来确认发送端发送的所有数据都已成功接收。最后的 FIN-ACK 传输关闭了接收端至发送端的连接。注意 FIN 报文段包含了有效的 ACK 号

可以通过这些信息来初始化相关变量。

### 16.7 TCP 友好性

TCP 作为最主要的网络传输协议，在传输路径中会经常出现几个 TCP 连接共享一个或多个路由的情况。然而，它们并非均匀地共享带宽资源，而是根据其他连接动态地调节分配。但也会出现例外情形，如 TCP 与其他（非 TCP）连接或者使用不同设置的 TCP 连接竞争带宽。

为避免多个 TCP 连接对传输资源的恶性竞争，研究者提出了一种基于计算公式的速率控制方法，限制特定环境下 TCP 连接对带宽资源的使用。该方法称为 TCP 友好速率控制（TCP Friendly Rate Control, TFRC）[RFC5348] [FHPW00]，它基于连接参数和环境变量（如 RTT、丢包率）实现速率限制。与传统 TCP 相比，它能实现更高的带宽利用率，因此更适用于流媒体

这种大传输量（如视频传输）的应用。TFRC 使用如下公式来决定发送率：

$$X = s / (RV2bp/3 + 3phero) + 32p / V3bp/8 \quad (16.9)$$

这里的 X 指吞吐率限制（字节/秒），s 为包大小（字节，包含头部），R 是 RTT（秒），P 为丢包率 [0,0.1]，hro 为重传超时（秒），b 指一个 ACK 能确认的最大包个数。建议 hRTo 设为 4R，b 设为 1。

从另一方面来看 TCP 发送率，即在拥塞避免阶段，怎样根据接收的无重复 ACK 来调整窗口大小。回顾前面讨论过的标准 TCP，使用拥塞避免算法时，每接收一个好的 ACK，cwnd 就会增加 1/cwnd，而每当出现一次丢包，cwnd 就会减半，这被称为和式增加/积式减少（Additive Increase/Multiplicative Decrease, AIMD）拥塞控制。通过将 1/cwnd 和 1/2 替换为 a 和 b，我们得到了一般化的 AIMD 等式：

$$cwnd, +1 = cwnd, +a / ewnd, cwnd, +1 = cwnd, -b * cwnd, \quad (16.10)$$

根据 [FHPW00] 给出的结果，上述等式得出的发送率为（以包个数 /RTT 为单位）：

对于传统 TCP, a=1, b=0.5，这样上式就简化为  $T=1.2/VP$ ，称为简化的标准 TCP 响应函数。它的 TCP 速率（cwnd 调节）只和丢包率相关，而没有考虑重传超时。当 TCP 没有受其他因素（发送方或接收方缓存、窗口缩放等）影响时，在这样的良性环境下，简化函数能很好地控制 TCP 性能。

对 TCP 响应函数的任何修改都会影响它（或实现了相似拥塞控制模式的其他协议）与标准 TCP 的竞争。因此，通常会使用相对公平（relative fairness）的方法来分析新的拥塞控制模式。根据丢包率，相对公平给出了改进拥塞控制模式协议和标准 TCP 协议的速率比。这是衡量改进模式在带宽共享方面公平性的重要指标。

要建立与标准 TCP 公平竞争的速率调节机制，理解上述公式只是第一步。针对特定的协议，实现 TFRC 会存在具体的细节差异，包括怎样正确测量 RTT、丢包率、包大小等。这些问题在 [RFC5348] 中有详细讨论。

## 16.8 高速环境下的 TCP

在 BDP 较大的高速网络中（如 1Gb/s 或者更大的无线局域网），传统 TCP 可能不能表现出很好的性能。因为它的窗口增加算法（特别是拥塞避免算法）需要很长一段时间才能使窗口增至传输链路饱和。也就是说，即使没有拥塞发生，TCP 也不能很好地利用高速网络。产生这一问题的原因主要在于拥塞避免算法中的增量为固定值。如果一个 TCP 使用 1500 字节的数据包在一个 10Gb/s 的长距离链路上传输，假设没有出现丢包和传输错误，要想完全利用所有的带宽需要 83 000 个报文段。若每个 RTT 为 100 毫秒，完成 50 亿个数据包传输大约需要 1.5 个小时。

为了弥补这一不足，研究人员致力于改进 TCP 协议，使其在高速网络环境下能够获得更好的性能，并且在一定程度上保持与标准 TCP 的公平性，特别是在更为普遍的低速环境中。

### 16.8.1 高速 TCP 与受限的慢启动

高速 TCP (HSTCP) 的技术说明 [RFC3649] [RFC3742] 指出，当拥塞窗口大于一个基础值 `Low_Window` 时，应当调整标准 TCP 的处理方式。其中 `Low_Window` 设置为 38 个 MSS。这个值与前面提到的简化的标准 TCP 响应函数所给出的 10<sup>-3</sup> 丢包率相一致。其发送速率和丢包率在双对数坐标系中是线性相关的，所以它是一个具有幂律特性的函数。

在双对数坐标系中形成一条直线的函数称为幂律 (power law) 函数，其方程式为  $y=ax$ ，也可表示为  $\log y = \log a + k \log x$  ( $a$  和  $k$  是常数)，在双对数坐标系中是一条斜率为  $k$  的直线。

为建立幂律函数，我们需要选择两个点，然后建立一个方程式，使这个方程式所描述的直线经过这两个点。假设这两个点分别为  $(P_1, w_1)$  和  $(P_0, W_0)$ ，其中  $M_i > W_0 > 0$  且  $0 < P_1 < P_0$ 。在一个线性坐标系中，这两个点会建立一个斜率为  $(w_1 - W_0) / (P_1 - P_0)$  的直线，但是在双对数坐标系中，它们所形成的直线的斜率  $S = (\log M_1 - \log W_0) / (\log P_1 - \log P_0)$ 。然后，基于前面提到的公式，我们可以得到  $w = C p^S$ 。我们还需要一个点来定义  $C$ ，这个点可以是  $(P_0, W_0)$ 。经过一系列代数计算，我们可以得出  $C = P_0^S W_0$ ，即  $p P_0^S$ 。

图 16-19 给出了基于点  $(P_0, W_0) = (0.0015, 31)$  且  $S = -0.82$  的传统 TCP 和 HSTCP 响应函数的图示。在丢包率较大的情况下 (大于 0.001)，两者没有差别，所以该表达式只适用于  $P$  值较大的情况。比较这两条直线，当丢包率足够小时，HSTCP 可以达到更快的发送速率。

为了使 TCP 实现上述响应函数，需要调整拥塞避免机制。当窗口发生改变时，需要考虑当前的窗口大小。与传统 TCP 类似，在接收一个好的 ACK 后需要参考当前窗口大小来调整窗口值。具体响应如下：

$$cwnd_{i+1} = cwnd_i + a(cwnd_i) / cwnd_i, \quad (16.11)$$

当对于拥塞事件进行响应时 (例如丢包，或者发现 ECN 标志)，它的响应如下：

$$cwnd_{i+1} = cwnd_i - b(cwnd_i) * cwnd_i, \quad (16.12)$$

这里的  $a()$  是和式增加函数，而  $b()$  是积式减小函数。在标准 TCP 中，它们都是当前窗口大小的相关函数。为了获得预期的响应函数，我们首先由公式 (16-3) 推广得出以下公式：

变换得：

$$a(w) = 2P / (M - 3bw / 2 - bw) \quad (16.13)$$

上述关系式有多个解，也就是说，有多种  $a()$  和  $b()$  的组合方式可以满足，然而有些解对于调度算法来说并不适用。

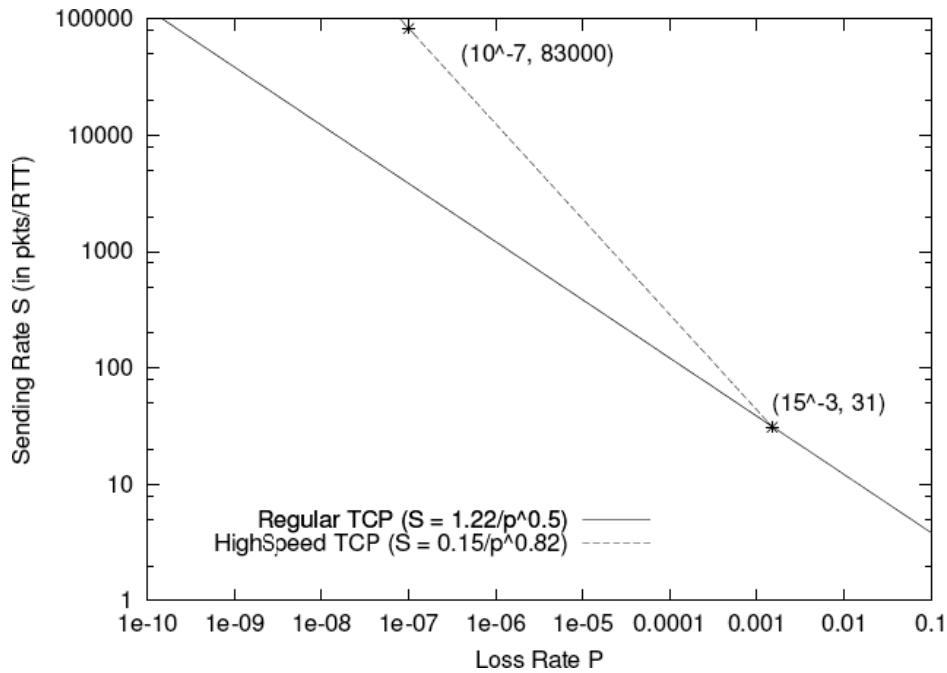


图 16.19: 在高速 TCP 中, 针对更低的丢包率和更大的窗口, TCP 响应函数需要做相应的调整, 从而在高带宽延迟的网络中获得更大的吞吐量。图片来自 Sally Floyd 2003 年 3 月在 IETF TWVWG 的演讲

[RFC3649] 中还给出了其他的 HSTCP 对传统 TCP 的拥塞避免改进的细节。[RFC3742] 描述了如何修改慢启动阶段, 使其在高速环境下得到运行中的拥塞窗口值。它被称为受限的慢启动, 即减慢速度的慢启动, 这样在处理大窗口的情况下 (几千甚至几万个数据包), TCP 不会在一个 RTT 中使窗口翻倍。

在受限的慢启动阶段, 引入了一个新的参数称为 `max_ssthresh`。这个值不是 `ssthresh` 的最大值, 而是 `cwnd` 的一个阈值: 如果 `cwnd <= max_ssthresh`, 则慢启动阶段与传统 TCP 相同。如果 `max_ssthresh < cwnd <= ssthresh`, 那么 `cwnd` 在每个 RTT 中最大只能增长  $(\text{max\_ssthresh}/2)$  个 SMSS。具体管理 `cwnd` 的方式如下:

```

    if (cwnd <= max_ssthresh) /
wnd = cwnd + SMSS
(regular slow start)
y else {
K = int (cwnd / (0.5 * max_ssthresh) )
cwnd = cwnd + int ( (1/K) * SMSS)
(limited slow start)

```

建议 `max_ssthresh` 的初始值为 100 个包, 或者  $100 \times \text{SMSS}$  个字节。

## 16.8.2 二进制增长拥塞控制 (BIC 和 CUBIC)

HSTCP 是为在大 BDP 网络中实现高吞吐量的一种 TCP 修改方案。它兼顾了在普通环境下与传统 TCP 的公平性，但在特环境中能够达到更快的发送速度。在多个具有不同 RTT 的连接竞争带宽时，HSTCP 不会直接控制这些竞争行为（称为“RTT 公平性”）。对标准 TCP 的研究表明，当使用相同的数据包大小和 ACK 策略时，较小的 RTT 在共享传输路径上能获得更大的带宽 [F91]。对于能根据自身窗口大小来调节 cwnd 增长值的 TCP（称为带宽可扩展 TCP）来说，这种不公平性表现得更加严重。是否遵守 RTT 公平性一直是一个存在争论的话题。尽管第一感觉认为 RTT 公平性是必要的，但是拥有较大 RTT 的连接可能会使用更多的网络资源（例如经过更多的路由器），所以拥有较小的吞吐量也是合理的。不论是哪种观点，了解 RTT 的公平性（或不公平性）行为是我们接下来探讨各个 TCP 改进版本的动因。

### BIC-TCP 算法

为建立一种可扩展的 TCP 及解决 RTT 公平性问题，提出了 BIC-TCP 算法（之前称为 BI-TCP 算法）[XHR04]，并从 Linux 2.6.8 内核版本中开始应用。BIC TCP 算法的主要目的在于，即使在拥塞窗口非常大的情况下（需要使用高带宽的连接），也能满足线性 RTT 公平性（linear RTT fairness）。线性 RTT 公平性是指连接得到的带宽与其 RTT 成反比，而不是一些更复杂的函数。

该方法使用了两种算法来修改标准 TCP 发送端：二分搜索增大（binary search increase）和加法增大（additive increase）。这些算法在出现一个拥塞信号（如丢包）后被调用，但是在任一给定时刻只运行一种算法。二分搜索增大算法的操作过程如下：当前最小窗口是最近一次在一个完整 RTT 中没有出现丢包的窗口大小，最大窗口是最近一次出现丢包时的窗口大小。预期窗口位于这两个值之间。BIC-TCP 使用二分搜索技术选择这两个值的中点作为一个试验的窗口，然后进行递归。如果这个点依然会发生丢包，那么将它设置为最大窗口，然后继续重复上述过程。如果不发生丢包，那么将它设置为新的最小窗口，然后同样继续重复上述过程。直到最大窗口和最小窗口的差值小于一个预先设置好的阈值时，这一过程才停止，其中这个阈值被称为最小增量（minimum increment），或者  $s_{\min}$ 。

这一算法往往会在一个对数级的试验次数内找到预期窗口，也被称为饱和点（saturation-point）。而标准 TCP 则需要多项式级的次数（平均为窗口大小差值的一半）。因此，这种方法使 BIC-TCP 在特定的处理阶段拥有比标准 TCP 更快的速度。它是为充分利用高速网络没有必要延时的优点而设计的。与其他协议相比较，BIC-TCP 表现得有所不同，它的增长速率在一些时间点是下降的，也就是说，越接近饱和点，它增长得越慢。而大多数其他算法在接近饱和点时都会增长得更快。

加法增大算法运作过程如下：当使用二分搜索增大算法时，可能会出现当前窗口大小与中间点（从二分搜索意义上说）之间差距很大。由于可能出现突然大量数据注入网络的情况，所以在



一个 RTT 内将窗口增大到中间点可能并不是一个好方法。这种情况下就要采用加法增大算法。当中间点与当前窗口大小之间的差值大于一个特定值  $S_{mux}$  的时候，将调用加法增大算法。此时，增量被限制为每个 RTT 增加  $S_{lhx}$ ，这一增量被称为窗口夹 (windowclamping)。一旦中间点距离试验窗口比距离  $S_{mux}$  值更近时，则转换为使用二分搜索增大算法。总的来说，当检测到丢包现象，窗口会使用乘法系数  $B$  来减小，而窗口增大时，首先使用加法增大算法，之后一旦确认加法增量小于  $S_{mux}$  时就转为使用二分搜索增大算法。这种混合的算法称为二进制增长 (binary increase)，或者 BI。

当窗口增至超过当前最大值，或由于还没有丢包发生而没有已知的最大值时，增长会终止。这是由最大位探测 (max probing) 机制所实现的。最大值探测的目的是有效地利用带宽。它使用一种加法增大和二分搜索增大的对称方式。初始时，它设置了一个较小的增量。之后如果没有检测到拥塞，它就会使用更大的增量。因为在饱和点附近变化量较小，而且在饱和点处网络能够表现出最佳的性能，所以这种方法具有良好的稳定性。

Linux 系统 (内核版本 2.6.8 至 2.6.17) 中实现了 BIC-TCP 算法，并默认开启。有 4 个系统参数用来控制它的操作：`net.ipv4.tcp_bic`、`net.ipv4.tcp_bic_beta`、`net.ipv4.tcp_bic_low_window` 和 `net.ipv4.tcp_bic_fast_convergence`。第一个布尔变量用来控制是否使用 BIC (与传统的快速重传和快速恢复相对应)。第二个变量包含了一个比例因子，它可以通过 `cwnd` 值来决定  $S_{mut}$  值 (默认为 819)。第三个参数控制在运行 BIC-TCP 算法前的最小拥塞窗口的大小。它的默认值为 14，这意味着标准 TCP 拥塞控制决定最小窗口。最后一个参数是一个标志位，默认状态下是开启的。在二分搜索增大算法处于下降趋势时，它会影响新的最大窗口和目标窗口的选择。在窗口减小的过程中，新的最大窗口和最小窗口将被分别设置 `cwnd` 和一定比例的 `cwnd` 值 (由参数  $B$  决定，即  $B * cwnd$ )。如果启用快速收敛，并且新的最大值小于它之前的值，那么最大窗口将在它与最小窗口的平均值范围内继续减小。在这之后，不论快速收敛是否开启，目标窗口将设置为最大窗口与最小窗口的平均值。这种方式有助于在多个 BIC-TCP 流共享一个路由器时，更快地分配带宽。

## CUBIC

BIC-TCP 的开发者的基本算法进行改进，形成新的控制算法，称 CUBIC [HRX08]。自 2.6.18 内核版本起，它一直是 Linux 系统的默认拥塞控制算法。CUBIC 改进了 BIC-TCP 在一些情况下增长过快的不足，并对窗口增长机制进行了简化。它不像 BIC-TCP 那样使用阈值 ( $S_{mux}$ ) 来决定何时调用加法增大算法和二分搜索增大算法，而是使用一个高阶多项式函数 (具体来说是一个三次方程) 来控制窗口的增大。三次方程的曲线既有凸的部分也有凹的部分。这就意味着，在一些部分 (凹的部分) 增长比较缓慢，而在另一些部分 (凸的部分) 增长比较迅速。在 BIC 算法和 CUBIC 算法之前，所有 TCP 研究提出的都是凸的窗口增长函数。CUBIC 算法中，这个特

殊的窗口增长函数如下所示：

$$W(t) = C t - K' + W_{aux} \tag{16.14}$$

在这个表达式中， $W(t)$  代表在时刻  $t$  的窗口大小， $C$  是一个常量（默认为 0.4）， $t$  是距离最近的一次窗口减小所经过的时间，以秒为单位。 $K'$  是在没有丢包的情况下窗口从  $W$  增长到  $W_{aux}$  所用的时间。 $W_{mux}$  是最后一次调整前的窗口大小。其中  $K'$  可依据以下表达式计算：

其中  $B$  是积式减少的常量（默认为 0.2）。图 16-20 为  $K=2.71$ 、 $W_{mm}=10$ 、 $C=0.4$  时，在时间段为  $t=[10,5]$  时 CUBIC 窗口增大算法的图示。

图中显示了 CUBIC 窗口增大函数既包含凸的部分也包含凹的部分，当发生快速重传时， $W_m$  被设置为  $ewnd$ ，新的  $cwnd$  值和  $ssthresh$  值被设置为  $B * cwnd$ 。CUBIC 算法中的  $B$  默认为 0.8。 $I (+ RTT)$  值是下一个目标窗口的值。当在拥塞避免阶段，每收到一个 ACK， $cwnd$  值增加  $(W(t + RTT) - cwnd) / cwnd$ 。

值得注意的是，将  $t$  设置为距上次窗口减小经过的时间，有助于确保 RTT 的公平性。这里并不使用固定值对窗口进行改变，而用关于  $t$  的函数来调节窗口大小。这种方法将窗口变更操作从传统模式中分离出来。

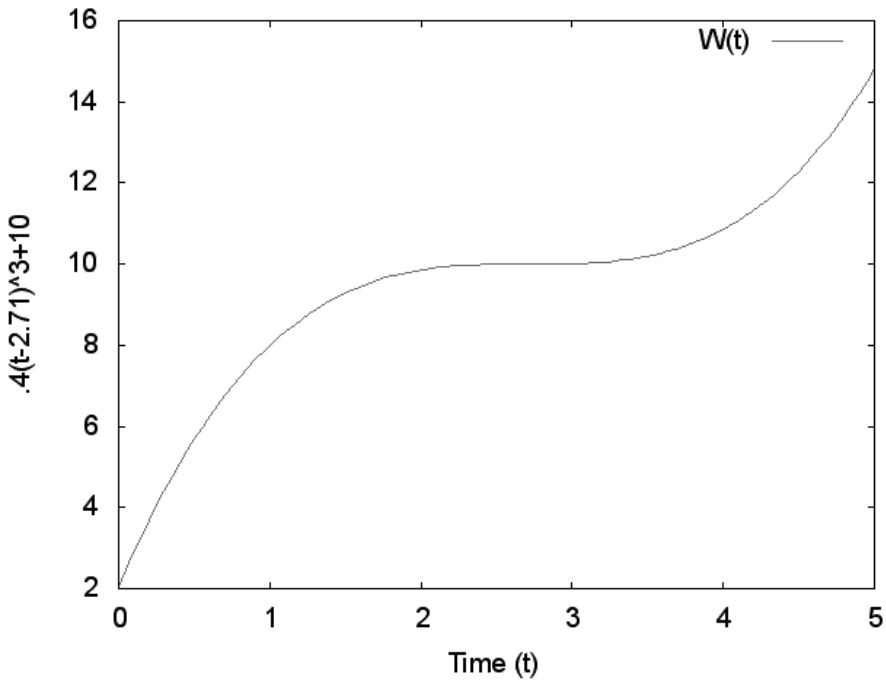


图 16.20: CUBIC 窗口增长函数是一个关于的三次函数。它在  $W(t) < W_{max}$  的区域是凹函数。在这一区域， $cwnd$  的增长越来越慢。在达到  $W_a$  之后，增长函数变为凸函数。在这一区域。 $cwnd$  的增长越来越快

除了三次方程之外，CUBIC 还有“TCP 友好”策略。当窗口太小使得 CUBIC 不能获得比传统 TCP 更好的性能时，它将会开始工作。根据 1 可以得到标准 TCP 的窗口大小  $Tep(A)$ ：

当在拥塞避免阶段有一个 ACK 到达时, 如果 `cwnd` 值小于 `Wep` ( ), 那么 CUBIC 将 `cwnd` 值设置为 `Wop` ( )。这种方法确保了 CUBIC 在一般的中低速网络中的 TCP 友好性, 而在这些网络中标准 TCP 相对 CUBIC 算法更具优势。

如前所述, 从 Linux 2.6.18 内核版本起, CUBIC 算法就是 Linux 系统的默认拥塞控制算法。而从 2.6.13 版开始, Linux 支持可装卸的拥塞避免模块 [PO7], 用户可以选择想用的算法。变量 `net.ipv4.tcp_congestion_control` 表示当前默认的拥塞控制算法 (默认为 `cubic`)。变量 `net.Ipv4.tep_available_congestion_control` 表示系统所载人的拥塞控制算法 (一般, 其他的算法可被作为内核模块载人)。变量 `net.jpv4.top_allowed_congestion_control` 表示用户允许应用程序使用的算法 (可以具体选择或者设为默认)。它默认支持 CUBIC 算法和 Reno 算法。

## 16.9 基于延迟的拥塞控制算法

前面介绍的拥塞控制方法都是通过检测丢包、利用一些 ACK 或 SACK 报文探测、ECN 算法 (如果可用)、重传计时器的超时来触发的。ECN 算法 (16.11 节) 允许一个 TCP 发送端向网络报告拥塞状况, 而不用检测丢包。但是这要求网络中每一个路由器的参与, 比较难以实现。然而, 在没有 ECN 的情况下, 判断网络中的主机是否发生拥塞也是可能的。当发送端不断地向网络中发送数据包时, 不断增长的 RTT 值就可以作为拥塞形成的信号。我们在图 16-8 中看到过这种情况。新到达的数据包没有被发送, 而是进入等待队列, 这就造成了 RTT 值不断增大 (直到数据包最终被丢弃)。一些拥塞控制技术就是根据这种情况提出的。它们被称为基于延迟的拥塞控制算法, 与我们至今为止看到的基于丢包的拥塞控制算法相对。

### 16.9.1 Vegas 算法

TCP Vegas 算法于 1994 年被提出 [BP95]。它是 TCP 协议发布后的第一个基于延迟的拥塞控制方法, 并经过了 TCP 协议开发组的测试。Vegas 算法首先估算了一定时间内网络能够传输的数据量, 然后与实际传输能力进行比较。若本该传输的数据并没有被传输, 那么它有可能被链路上的某个路由器挂起。如果这种情况持续不断地发生, 那么 Vegas 发送端将降低发送速率。这与标准 TCP 中利用丢包来判断是否发生拥塞的方法相对。

在拥塞避免阶段, Vegas 算法测量每个 RTT 中所传输的数据量, 并将这个数除以网络中观察到的最小延迟时间。算法维护了两个阈值  $a$  和  $B$  ( $a$  小于  $B$ )。当吞吐量 (窗口大小除以观察到的最小 RTT) 与预期不同时, 若得到的吞吐量小于  $Q$ , 则将拥塞窗口增大; 若吞吐量大于  $B$ , 则将拥塞窗口减小。吞吐量在两阈值之间时, 拥塞窗口保持不变。拥塞窗口所有的改变都是线性的, 这意味着这种方法是一种和式增加/和式减少 (Additive Increase/AdditiveDecrease, AIAD) 的拥塞控制策略。

作者通过链路瓶颈处的缓冲区利用率来描述  $a$  和  $B$ 。 $a$  和  $B$  的最小值分别设置为 1 和 3。设置该值的原因是：在网络中至少有一个数据包缓冲区会被占用（也就是说，路由器上的队列表示网络链路上的最小带宽）才能保持网络资源被充分利用。如果 Vegas 只维护一个缓冲区，那么当有其他可用带宽时，就要等待额外的 RTT 时间。因此为了传输更多的数据，需要多使用两个缓冲区（达到 3， $a$  的值）。此外，保持一个区间 ( $B-a$ ) 可以保留一部分空间，使得吞吐量可以有小幅改变而不至引发窗口大小的改变。这种缓冲机制可以减少网络的震荡。

稍加修改后，这种方法也适用于慢启动阶段。这里，每隔一个 RTT， $cwnd$  值才随着一个好的 ACK 响应而加 1。对于  $cwnd$  没有增长的 RTT，需要测量吞吐量是否在增长。如果没有，发送端将转为 Vegas 拥塞避免方式。

在特定情况下，Vegas 算法会盲目地相信前向的延迟会高于它实际的值。这种情况发生在它相反的方向产生了拥塞（回忆一下，TCP 连接的两个方向的链路可能会不同，并产生不同程度的拥塞状态）。在这种情况下，虽然不是发送端导致了（反向的）拥塞，但是返回发送端的数据包（ACK）会产生延迟。这就使得在这种不是真正需要调整拥塞窗口的情况下，减小了窗口大小。这是大多数基于测量 RTT 来进行拥塞控制判断的方法所共有的潜在缺陷。甚至，在反方向上严重的拥塞问题会导致 ACK 时钟（图 16-1）的严重紊乱 [M92]。

Vegas 与其他的 Vegas TCP 连接平等共用一条链路，因为每一次向网络中传输的数据量都很小。然而，Vegas 与标准 TCP 流共用链路时则是不平等的。标准 TCP 的发送端想要占满网络中的等待队列，反之 Vegas 则是想使它们保持空闲。因此，当标准 TCP 发送端发送数据包时，Vegas 发送端会发现延迟在增长，那么它就会降低发送速率。最终，这导致了只对标准 TCP 有利的状态。Linux 系统支持 Vegas，但不是默认开启的。对于 2.6.13 之前的内核版本来说，布尔型 `sysctl` 变量 `net.ipv4.tcp_ Vegas_cong_avoid` 决定了是否使用 Vegas（默认为 0）。变量 `net.ipv4.tcp_ Vegas_alpha`（默认为 2）和变量 `net.ipv4.tcp_ Vegas_beta`（默认为 6）决定了上面提到的  $a$  和  $B$  值，但是它们的单位是半个数据包（也就是说，6 对应着 3 个数据包）。变量 `net.ipv4.tcp_ Vegas_gamma`（默认为 2）用于配置在经过多少个半数据包后 Vegas 结束慢启动阶段。对于 2.6.13 之后的内核版本，Vegas 需要作为分离的内核模块被载入，通过设置 `net.ipv4.tcp_congestion_control` 来启动 Vegas。

### 16.9.2 FAST 算法

FAST TCP 算法是为处理大带宽延迟的高速网络环境下的拥塞问题而提出的 [WJLH06]。原理上与 Vegas 算法相同，它依据预期的吞吐量和实际的吞吐量的不同来调整窗口。与 Vegas 算法不同的是，它不仅依据窗口大小，而且还依据当前性能与预期值的不同来调整窗口。FAST 算法会使用速率起搏（rate-pacing）技术每隔一个 RTT 都会更新发送率。如果测量延迟远小于阈值时，窗口会进行较快增长，一段时间后会逐渐平缓增长。当延迟增大时则相反。FAST 算法与我们之前所说的方法不同，因为在其中包含了一些专利，并且它正被独立地商业化。FAST 算法

被一些研究机构质疑缺乏安全性，但是一个评估报告 [S09] 显示它具有良好的稳定性和公平性。

### 16.9.3 TCP Westwood 算法和 Westwood+ 算法

TCP Westwood 算法 (TCPW) 和 TCP Westwood + 算法 (TCPW +) 的设计目的在于，通过修改传统的 TCP NewReno 发送端来实现对大带宽延迟积链路的处理。TCPW + 算法是对 TCPW 算法的修正，所以这里只对 TCPW 算法进行说明。在 TCPW 算法中，发送端的合格速率估计 (ERE) 是一种对连接中可用带宽的估计。类似 Vegas 算法 (基于预期速率与实际速率的差别)，该估计值被不断计算。但是不同的是，对于这个速率的测量会有一个测量间隔，该间隔基于 ACK 的到达动态可变。当拥塞现象不明显时，测量间隔会比较小，反之亦然。当检测到一个数据包丢失的时候，TCPW 不会将 `cwnd` 值减半，而是计算一个估计的 BDP 值 (ERE 乘以观察到的最小 RTT)，并将这个值作为新的 `ssthresh` 值。另一方面，在连接处于慢启动阶段时，使用一种灵活的探测机制 (Agile Probing) [WYSGOS] 适应性地反复设置 `ssthresh` 值。因此当 `ssthresh` 值增长时 (由于初始的慢启动)，`cwnd` 值会以指数形式增长。在 Linux 2.6.13 之后的内核版本中，可以通过加载一个 TCPW 模块，并设置 `net.ipv4.tcp_congestion_control` 为 Westwood 来启动 Westwood。

### 16.9.4 复合 TCP

类似于 Linux 系统中的可装卸的拥塞避免模块，从 Windows Vista 系统开始，用户也可以自主选择使用何种 TCP 拥塞控制算法。该选项 (除了 Windows Server 2008 外，默认不开启) 称为复合 TCP (Compound TCP, CTCP) [TSZS06]。CTCP 不仅依据丢包来进行窗口的调整，还依据延迟的大小。可以认为它是一种标准 TCP 和 Vegas 算法的结合，而且还包含了 HSTCP 可扩展的特点。

Vegas 算法和 FAST 算法的研究结果显示，基于延迟的拥塞控制方法可以得到更好的利用率、更少的自诱导的丢包率、更快的收敛性 (对于正确的操作来说)，并且使 RTT 更具公平性和稳定性。然而，就像前面提到过的，基于延迟的方法在与基于丢包的拥塞控制方法竞争时会失去优势。CTCP 就是希望通过将基于延迟的方法和基于丢包的方法相结合来解决该问题。为了达到这一目的，CTCP 定义了一个新的窗口控制变量 `dwnd` (“延迟窗口”)。可用窗口大小  $W$  则变成了

$$W = \min(cwnd + dwnd, awnd) \quad (16.15)$$

对 `cwnd` 值的处理与标准 TCP 类似，但是如果延迟允许，新加入的 `dwnd` 值会允许额外的数据包发送。在拥塞避免阶段当 ACK 报文到达时，`cwnd` 值根据下面的公式进行更新：

$$cwnd = cwnd + 1 / (cwnd + dwnd) \quad (16.16)$$

dwnd 值的控制是基于 Vegas 算法的，并且只在拥塞避免阶段才是非零值（CTCP 使用传统的慢启动方式）。当连接建立时，使用一个变量 baseRTT 来表示测量到的最小 RTT 值。然后预期数据与实际数据的差值 diff 将使用如下公式进行计算： $\text{diff} = (1 - (\text{baseRTT}/\text{RTT}))$ 。其中 RTT 是估算的（平滑的）RTT 估计。diff 的值估算了网络队列中的数据包数量（或字节数）。与大多数基于延迟的方法类似，CTCP 算法试图将 diff 值保持在一个阈值内，以此保证网络的充分利用而不至于出现拥塞，这个阈值定义为  $y$ 。为了达到这一目的，对于 dwnd 值的控制可依据以下公式：

其中  $(x)^+$  表示  $\max(x, 0)$ 。注意这里 dwnd 值非负。而当 CTCP 像标准 TCP 那样工作的时候，dwnd 值应为 0。

在第一种情况下，网络没有被充分利用，CTCP 根据多项式  $a \cdot \text{win}(d)^k$  增大 dwnd 值。这是一种多项式级的增长，而当缓冲区的占用率小于  $y$  时，会更快速地增长（类似于 HSTCP）。在第二种情况下，缓冲区的占用率已经超过了阈值  $y$ ，固定值  $B$  表示延迟窗口的递减速率（dwnd 经常为 cwnd 的加数）。这就使得 CTCP 的 RTT 更具公平性。当检测到丢包时，dwnd 值会有自己的积式递减系数  $B$ 。

可以看到，CTCP 需要使用参数  $K$ 、 $a$ 、 $B$ 、 $k$  和  $y$ 。 $k$  的值表示速度的等级。与 HSTCP 类似，可以将  $k$  值设置为 0.8，但是由于实现方面的原因， $k$  值被设置为 0.75。 $a$  和  $B$  值表示了平滑度和响应性，分别被默认设置为 0.125 和 0.5。对于  $y$  值，这里凭借经验将其设置为 30 个数据包。如果这个值太小，将不会有足够的数据包，以致不能得到较容易测量的延迟。相反，如果这个值太大则会导致长时间的拥塞。

CTCP 算法相对比较新，通过更深入的实验和改进，会使其与标准 TCP 相比拥有更好的性能，并且能够很好地适应不同的带宽。在一个仿真实验中，[W08] 注意到在网络缓冲区较小时（小于  $y$  值），CTCP 算法的性能会很差。他们还提出 CTCP 也存在着一些 Vegas 算法中的问题，包括重新路由问题（适应具有不同延迟的新链路）和持续的拥塞问题。他们发现，如果有很多的 CTCP 流，其中每一个都要维护  $y$  个数据包，并且共用一条相同瓶颈的链路时，CTCP 的性能会非常差。

像前面所提到的，CTCP 在大多数版本的 Windows 系统中不是默认开启的。然而，下面的命令可以用来选择 CTCP 作为拥塞控制方法。

```
C: \> netsh interface tcp set global congestionprovider=ctcp
```

它可以通过另选一个不同的（或不设置）控制算法来关闭。CTCP 也作为一个可装卸式的拥塞避免模块而移植到 Linux 系统中，当然它也不是默认启用的。

## 16.10 缓冲区膨胀

虽然存储单元的价格昂贵（高端路由器也是如此），但是现在的网络设备中仍包含大量的内存和几百万字节的包缓冲区。然而，这样庞大的内存（与传统的网络设备相比）会导致像 TCP 这样的协议性能下降。这一问题被称为缓冲区膨胀 [G11] [DHGSOT]。它主要存在于家用网关的上行端以及家庭或小型办公室的接入点处，与排队等待而产生的大量延迟有关。标准 TCP 协议的拥塞控制算法会在链路的瓶颈处将缓冲区填满。而由于拥塞的信号（一个数据包丢失）需经很长时间才能反馈到发送端，此时在发送端和接收端之间缓存了大量数据，TCP 协议也不能很好地运作。

KWNP10] 中指出，在美国包括电缆和 DSL 在内，上传带宽范围是 256Kb/s 4Mb/s，在商用路由器上的缓冲区大小应该在 16KB 至 256KB 之间。图 16-21 显示了在几种缓冲区大小下延迟和数据传输速率的关系，可以证明之前结论的正确性。

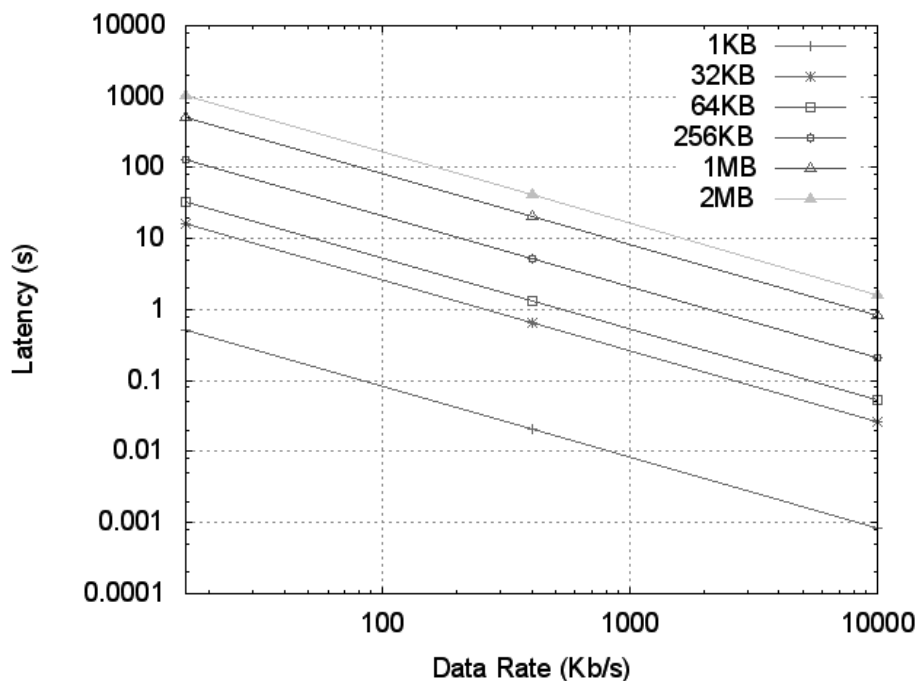


图 16.21: 双对数坐标系显示了队列等待的延迟随拥塞队列长度的变化情况。当缓冲区被占满时（“缓冲区膨胀”），交互式的应用程序将会产生无法容忍的延迟

图 16-21 展示了针对不同缓冲区大小（1KB 2MB）数据在队列等待所产生的延迟情况。如果缓冲区大小为几百 KB 或者更多，那么家庭网络上传带宽速率（一般在 250Kb/s 至 10Mb/s 之间）会引起几百秒的延迟。为了给用户带来更好的体验，一般的交互式应用程序需要把单向延迟控制在 150ms 以下 [G114]。因此，如果缓冲区被一个或多个大的上传文件所占满（如 BT 共享文件），会严重影响交互式应用性能。

不是所有的网络设备中都存在缓冲区膨胀的问题。实际上，主要问题是缓冲区端用户接入设备过满。有很多方法可以解决这一问题，包括修改协议（如像 Vegas 这样的基于延迟的拥塞控制方式，但是它可能会因为网络抖动而产生相反效果 [DHGSO7]）、使用缓冲区大小可动态改变的接入设备（[K WNP10] 中提到），或将两者结合。接下来介绍一种综合的方法，它不仅可以解决缓冲区膨胀的问题，而且还有一些其他的好处。

## 16.11 积极队列管理和 ECN

到现在为止，TCP 能够推断出拥塞产生的唯一方法就是发生丢包现象。特别是路由器（最有可能产生拥塞）通常不会通知连接两端的主机，TCP 即将产生拥塞。而是当缓存没有多余的可用空间时，只好将新到达的数据包丢弃（称为“尾部丢弃”）。然后依照先进先出（FIFO）的方法继续转发那些先前到达的数据包。当网络路由器像这样被动工作时（指它们在超负荷的时候仅仅丢弃数据包，而不会提供它们已经处于拥塞状态的任何反馈信息），TCP 除了事后再做出反应以外无能为力。然而，如果路由器可以更积极地管理它们的等待队列（也就是说使用更精确复杂的调度算法和缓存管理策略），也许这种情况就能得到改善。若可以将拥塞状态报告给端节点，效果会更好。

应用 FIFO 和尾部丢弃以外的调度算法和缓存管理策略被认为是积极的，路由器用来管理队列的相应方法称为积极队列管理（（AQM）机制。[RFC2309] 中提到了 AQM 机制的潜在优势。若可以通过将路由器和交换机的状态传输给端系统来实现 AQM 时，它将更具利用价值。这些在 [RFC3168] 中有详细描述，[RFC3540] 利用相关实验描述了扩展安全性的 AQM。这些 RFC 都描述了显式拥塞通知（Explicit Congestion Notification, ECN），它对经过路由器的数据包进行标记（设置 IP 头中的两个 ECN 标志位），以此得到拥塞状况。

随机早期检测（RED）网关 [FJ93] 机制能够探测拥塞情况的发生，并且控制数据包标记。这些网关实现了一种衡量平均占用时间的队列管理方法。如果占用队列的时间超过最小值（minthresh），并且小于最大值（maxthresh），那么这个数据包将被标记上一个不断增长的概率值。如果平均队列占用时间超过了 maxthresh，数据包将被标记一个可配置的最大的概率值（MaxP），MaxP 可以设置为 1.0。RED 也可以将数据包丢弃而不是标记它们。

RED 算法有多种版本（如思科的 WRED 就是基于 IP DSCP 和优先级的 RED 算法），很多路由器和交换机都支持。

当数据包被接收时，其中的拥塞标记表明这个包经过了一个拥塞的路由器。当然，发送端（而不是接收端）才真正需要这些信息，以此降低发送速率。因此，接收端通过向发送端返回一个 ACK 数据包来通知拥塞状况。

ECN 机制主要在 IP 层进行操作，也可以应用于 TCP 协议之外的其他传输层协议。当一



个包含 ECN 功能的路由器经过长时间的拥塞，接收到一个 IP 数据包后，它会查看 IP 头中的 ECN 传输能力 (ECT) 标识（在 I 头中由两位 ECN 标志位定义）。如果有效，负责发送数据包的传输层协议将开启 ECN 功能，此时，路由器会在 IP 头设置一个已发生拥塞 (CE) 标识（将 ECN 位都置为 1），然后继续向下转发数据报。若拥塞情况不会持续很长时间（例如由于队列溢出导致最新的一个数据包被丢弃），路由器不会将 CE 标识置位。因为即使是一个单独的 CE 标识，传输协议也会做出反应。

如果 TCP 接收端发现接收到的数据包的 CE 标识被置位，那么它必须将该标识发送回发送端（[RFC5662] 中的实验表明，也可以将 ECN 添加到 SYN+ACK 报文段中发送）。因为接收端经常会通过 ACK 数据包（不可靠的）向发送端返回信息，所以拥塞标识很有可能会丢失。出于对这种情况的考虑，TCP 实现了一个小型的可靠连接协议，通过这个协议可以将标识返回给发送端。TCP 接收端接收到 CE 标识被置位的数据包之后，它会将每一个 ACK 数据包的“ECN 回显” (ECN-Echo) 位字段置位，直到接收到一个从发送端发来的 CWR 位字段设置为 1 的数据包。CWR 位字段被置位说明拥塞窗口（也就是发送速率）已经降低。

RED

ECN

20

RED

2005

ECN “ ” KOS

ECN

(16.17)

TCP 发送端接收到含有 ECN-Echo 标识的 ACK 数据包时，会与探测到单个数据包丢失时一样调整 cwnd 值。同时发送端还会重新设置后续数据包的 CWR 位字段。常规的拥塞处理方式：调用快速重传和快速恢复算法（当然，数据包不会进行重传），这样就可以使 TCP 在丢包之前降低发送速率。值得注意的是，TCP 的处理不应该过度。特别是它不能对同一个数据进行多次响应。否则，ECN TCP 相对于其他来说会处于不利地位。

在 Windows Vista 及之后的版本中，激活 ECN 功能需要使用以下命令：

$$C : netsh \int t t p s e t g l o b a l e n c a p a b i l i t y = e n a b l e d$$

(16.18)

在 Linux 系统中，如果布尔型 Sysctl 变量 net.ipv4.tcp\_ecn 的值非零，则 ECN 功能被激活。这种基于 Linux 的改变默认设置的方法正在广泛使用。在 Mac OS 10.5 及更新的版本中，变量 net.inet.top.ecn\_initiate\_out 和 net.inet.tcp.ecn\_negotiate\_in 分别控制向外传输和向内传输的 ECN 功能的开启。当然，没有路由器和交换机的协作，ECN 的实用性在任何情况下都会受到限制。AQM 在整个全球互联网络中发挥作用还需要时间。

由于设计目的不同,RED 机制和 ECN 机制被用于完全不同的操作环。Microsoft 和 Stanford 开发了 Data Center TCP (DCTCP) [A10]。它使用了更简化的参数,在第 2 层交换机上实现了 RED 机制,可以在网络产生瞬时的拥塞时对数据包进行标记。它们还调整了 TCP 接收端的行沟,只有在最后一个接收到的数据包包含 CE 标记时,才将 ACK 中的 ECN-Echo 标记置位。报告显示达到相同的 TCP 吞吐量情况下,缓冲区的占用率下降了 90%,并使背景流量增长 10 倍。

## 16.12 与 TCP 拥塞控制相关的攻击

我们已经看到生成数据包是如何攻击 TCP,使其改变自身的连接状态机,从而断开连接的。当 TCP 处于 ESTABLISHED 状态时,它也会受到攻击(至少为非正常工作状态)。大多数针对 TCP 拥塞控制的攻击都是试图强迫 TCP 发送速度比一般情况更快或者更慢。

更早期的攻击方式是利用 ICMPv4 Source Quench(源抑制)报文的结构。当这些报文被发送到运行 TCP 协议的主机上时,任何与该 IP 地址相连的连接都会减慢发送速率。该 1 地址包含在 ICMP 报文中的违规数据报中。然而随着 1995 年路由器不再使用 Source Quench 报文进行拥塞控制([RFC1812]5.3.6 节),这种攻击方式也变得不可行了。另一方面,对于终端主机,[RFC1122] 规定 TCP 针对 Source Quench 报文必须降低速率。综合以上两点,解决这种攻击最简单的方法就是在路由器和主机上阻止 ICMP Source Quench 报文的传输。这种方式已得到普遍使用。

一种更复杂、更常用的攻击方式是基于接收端的不当行为 [SCWA99]。这里将描述三种攻击形式,它们都可以使 TCP 发送端以一个比正常状态更快的速率进行数据发送。这些攻击可用于使某个 web 客户端得到比其他客户端更高的优先权,分别为 ACK 分割攻击、重复 ACK 欺骗攻击、乐观响应攻击,还有一种在 TCP 中实现的变体,这里把它称为“TCP Daytona”。

ACK 分割攻击的原理是,将原有的确认字节范围拆分成多个 ACK 信号并返回给发送端。由于 TCP 拥塞控制是基于 ACK 数据包的到达进行操作的(而不是依据 ACK 信号中的 ACK 字段)。这样发送端的 cwnd 会比正常情况更快速地增长。要解决这一问题,与 ABC(适当字节数)类似,可通过计算每个 ACK 能确认的数据量(而不是一个数据包的到达)来判断是否力真的 ACK。

重复 ACK 欺骗攻击可以使发送端在快速恢复阶段增长它的拥塞窗口。回想之前讨论过的,在标准快速恢复模式中,每次接收到重复 ACK cwnd 都会增长。这种攻击会比正常情况更快地生成多余的重复 ACK。因为还没有一种明确的方法可以将接收到的重复 ACK 和它们所确认的报文段相对应(一个基于时间的伪随机数可以解决这一问题,我们将在第 18 章详细讨论),因此这种攻击更加难以防治。利用时间戳选项可以解决这一问题,可以设置该选项在每个连接中开启或关闭。然而解决这一问题的最好方法是,限制发送端在恢复阶段的在外数据值。

乐观响应攻击原理是对那些还没有到达的报文段产生 ACK。因为 TCP 的拥塞控制计算是基于端到端的 RTT 的。对那些还没有到达的数据提前进行确认就会导致发送端计算出的 RTT 比实际值要小，所以发送端将会比正常情况下更快地做出反应。但如果发送端收到了一个未发送数据的 ACK，通常会选择忽略该响应。与其他攻击方式不同，这种方法不能在 TCP 层保证数据的可靠传输（也就是说，已经被确认的数据可能会丢失）。丢失的数据会被应用层或者会话层协议重建，这是很常见的（例如在 HTTP/1.1 中）。为防范这类攻击，可定义一个可累加的随机数，使得发送数据段大小可随时间动态改变，以此来更好地匹配数据段和它对应的 ACK。当发现得到的 ACK 和数据段不匹配时，发送端就可以采取相应的行动。

接收端异常行为的问题也受到了一些研究 ECN 的专家的关注，回想一下使用 ECN 的 AQM 机制，TCP 接收端会在 ACK 消息中向发送端返回一个 ECN 标识。然后发送端据此将会降低它的发送速率。如果接收端不能向发送端返回 ECN 标识（或者网络中的路由器将这一标识清除），那么发送端将不会知道是否产生拥塞，也就不会降低发送速率。[RFC3540] 进行了相关实验，即将一个 IP 数据包中的 ECN 字段（2 比特）中的 ECT 位字段设置为随机数。发送方将该字段值设置为一个随机的二进制数，接收方将该字段的值加 1（一个异或操作）。当生成 ACK 响应时，接收端会把该值放置到 TCP 头部的第 7 位（一般保留为 0）。行异常的接收端有一半的概率可以猜到这一数值。因为每一个数据包都是相对独立的，所以一个行为异常的接收端必须猜对每一个 ECT 值，这样对于 k 个数据包来说全部猜对的概率只有  $1/2^k$ （对于任何一个长时间使用的连接这都是很微小的）。

## 16.13 总结

TCP 被设计为互联网中主要的可靠传输协议。虽然其最初的设计包含了流量控制功能，能够在接收方无法跟上时降低发送方的速度，但是最初并没有提供方法从防止发送方淹没双方之间的网络。在 20 世纪 80 年代末期，为了控制发送方的攻击性行为，TCP 开发了慢启动与拥塞避免算法，从而避免了因网络拥塞而造成的丢包问题。这些算法都依赖于使用一个隐含的信号、数据包丢失以及拥塞的指示。当检测到丢包时就会触发这些算法，无论是通过快速重传算法还是超时重传。

慢启动与拥塞避免通过在发送方设置一个拥塞窗口来实现对其操作的控制。该拥塞窗口将与传统的窗口一起使用（基于接收方提供的窗口广告）。一个标准的 TCP 会将其窗口的最小值限定为 2。随着时间的增长，慢启动要求拥塞窗口的数值指数地增加，而拥塞避免则会随着时间的推移而线性增长。在任何时刻都只能选择两种算法中的一种运行，而做出这一选择则需要比较拥塞窗口当前的数值与慢启动的阈值。如果拥塞窗口超过了阈值，那么采用拥塞避免；否则使用慢启动。慢启动起初只在建立 TCP 连接以及因超时而重新启动后使用。这也适用于连接长时间处于空闲状态的情况。在整个连接的过程中，慢启动的阈值会动态地进行调整。

多年来, 拥塞控制已经成为网络研究界关注的重要焦点之一。在通过 TCP 与它的慢启动、拥塞避免过程获得经验后, 一些改进方法被提出、执行以及标准化。通过跟踪 TCP 何时从一系列丢包中恢复, NewReno (TCP 的一个改进版本) 能够避免当多个数据包在同一个窗口中被丢弃时伴随 Reno 变异发生的停滞现象。SACK TCP 通过允许发送者在一个 RTT 中智能地修复多个数据包改善 NewReno 的性能。在使用 SACK TCP 时, 需要仔细地核算, 以确保发送者在与共享同一个互联网路径的其他 TCP 通信方比较时不会显得过分积极。

近期, 关于 TCP 拥塞管理的一些修改包括: 速率减半、拥塞窗口的验证与调制, 以及“撤销”过程。速率减半算法能够在检测出丢包后使拥塞窗口逐步而不是快速地减小。拥塞窗口验证尝试在发送应用程序空闲或不能发送的情况下确保拥塞窗口不会过大; 拥塞窗口调制限制了在接收到单一 ACK 后作为响应的突发传输的大小。“撤销”过程, 例如 Eifel 响应算法, 在数据包丢失信号被认为是虚假以及使用若干技术进行条件检测时撤销对拥塞窗口的修改。在上述情况下, 为了将减小拥塞窗口所带来的负面影响降至最低, 恢复拥塞状态至其对应条件优于减小拥塞窗口。

经过 TCP 有意义的实践, 发现拥塞避免过程需要花费相当长的一段时间才能找到并利用额外的可用带宽资源。因此, 大量关于“带宽可扩展”的建议成为 TCP 修改的方向。一个较为知名的版本 (在 IETF 中) 是 HSTCP。相比于传统的 TCP 而言, 它允许拥塞窗口在大数值且少有数据包丢失的情况下更加积极地增长。此外, 还有一些建议, 如 FAST 和 CTCP。它们的窗口增长过程都是基于数据包丢失与延迟的测量。在 Linux 系统上广泛部署的 BIC.TCP 与 CUBIC 算法使用了增长函数。该函数在某些区间呈凸形, 而在另一些区间则呈凹形。这样就能够支持在饱和点的小窗口变化, 从而可能以对新的可用带宽的迟缓响应 (但仍快于标准的 TCP) 为代价来增强稳定性。

随着显式拥塞通知 (ECN) 规范的提出, TCP 与互联网路由器的运营做出了一个重大改变, 即在出现丢包之前允许 TCP 检测是否开始发生拥塞。虽然模拟与研究的结果表明它是可取的, 但它需要适度地调整 TCP 的实现, 并使互联网路由器的操作方式发生重要改变。这种能力将部署到何种程度还有待观察。

虽然 TCP 提供了最广泛使用的互联网可靠数据传输方法, 但是它并没有以自身的安全方式实现。一般来说, 它非常容易受到伪造数据包攻击, 从而导致连接中断; 攻击者只需要猜出一个可行的 (窗口) 序列号就能够发起上述攻击。此外, 没有任何完全可行的方法能够阻止一个过分积极的发送者仅仅违反所有的拥塞控制规则。

将所有为 TCP 而开发的算法和技术都结合到一个 TCP 实现中并非易事 (Linux 2.6.38 的 TCP/IPv4 大约有 20000 行 C 语言代码), 而分析真实世界中 TCP 活动的记录需要耗费时间。诸如 tcpdump、Wireshark 以及 tcptrace 这样的工具使这项工作变得相对容易。由于动态地适应网络性能, 使用基于时间序列图的可视化技术能够更容易理解 TCP 的行为, 例如本章所采用的例子。

## 16.14 参考文献



# TCP 保活机制

---

## 17.1 引言

许金 TCPAP 的初学者会惊奇地发现，在一个空用的 TCP 连接中不会有任何数据交换。也就是说，如果 TCP 连接的双方都不向对方发送数据，那么 TCP 连接的两端就不会有任何的数据交换。例如，在 TCP 协议中，没有其他网络协议中的轮询机制。这意味着我们可以启动一个客户端进程，与服务器端建立连接，然后离开几个小时、几天、几星期，甚至几个月，而连接依然会保持。理论上，中间路由器可以崩溃和重启，数据线可以断开再连接，只要连接两端的主机没有被重新启动（或者更改 IP 地址），那么它们将会保持连接状态。

上述假设只是在特定情况下发生的。首先，客户端和服务端都没有实现应用层的非活动状态检测计时器，该计时器超时会导致任何一个应用进程的终止。其次，中间路由器不能保存连接的相关状态，例如一个 NAT 配置信息。某些特定操作中常常需要这些状态，而它们也会由于非活动状态而删除，或者由于系统故障而丢失。这些前提条件在现在的网络环境中是很难实现的。

一些情况下，客户端和服务端需要了解什么时候终止进程或者与对方断开连接。而在另一些情况下，虽然应用进程之间没有任何数据交换，但仍然需要通过连接保持一个最小的数据流。TCP 保活机制就是为了解决上述两种情况而设计的。保活机制是一种在不影响数据流内容的情况下探测对方的方式。它是由一个保活计时器实现的。当计时器被激发，连接一端将发送一个保活探测（简称保活）报文，另一端接收报文的同时会发送一个 ACK 作为响应。

保活机制并不是 TCP 规范中的一部分。对此主机需求 RFC [RFC1122] 给出了 3 个理由。(1) 在出现短暂的网络错误的时候, 保活机制会使一个好的连接断开; (2) 保活机制会占用不必要的带宽; (3) 在按流量计费的情况下会在互联网上花掉更多的钱。然而, 大部分的实现都提供了保活机制。

TCP 保活机制存在争议。许多人认为, 如果需要, 这一功能也不应在 TCP 协议中提供, 而应在应用程序中实现。另一种观点认为, 如果许多应用程序中都需要这一功能, 那么在 TCP 协议中提供的话就可以使所有的实现都包含这一功能。保活机制是一个可选择激活的功能。它可能会导致一个好的连接由于两端系统之间网络的短暂断开而终止。例如, 如果在中间路由器崩溃并重新启动的时候保活探测, 那么 TCP 协议将错误地认为对方主机已经崩溃。

保活功能一般是为服务器应用程序提供的, 服务器应用程序希望知道客户主机是否崩溃或离开, 从而决定是否为客户端绑定资源。利用 TCP 保活功能来探测离开的主机, 有助于服务器与非交互性客户端进行相对短时间的对话, 例如, Web 服务器、POP 和 IMAP 电子邮件服务器。而更多地实现长时间交互服务的服务器可能不希望使用保活功能, 如 sh 和 Windows 远程桌面这样的远程登录系统。

河以通过一个简单例子来说明保活功能的可用性, 即用户利用 ssh (安全 shell) 证程務录程序穿越 NAT 路由器登录远程主机。如果建立连接, 并做了相关操作, 然后在一天结束时没有退出, 而是直接关闭了主机, 那么便会留下一个半开放的连接。在第 13 章中已经提到过, 通过一个半开散的连接发送数据会返回一个重置信息, 但那是来自正在发送数据的客户端。如果客户端离开了, 只剩下服务暑端的一个半开放的连接, 而服务器又在等待客户缩发来的数据, 那么服务器将会永远地等待下去。在服务器端探测到这种半开放的连接时, 就可以使用保活功能。

相反的情况下同样需要使用保活机制。如果用户没有关闭计算机, 而是整个晚上保持连接, 第二天可以继续使用, 那么连接将连续几个小时处于空闲状态。在第 7 章中我们提到过, 大部分 NAT 路由器包含超时机制。当连接在一段时间内处于非活动状态时, 路由器格断开连接。如果 NAT 超时时限小于用户重新登录之前的几个小时, 且 NAT 不能探测到端主机并确认它还处于活动状态, 或者 NAT 路由器崩溃, 那么该连接将被终止。为了避免这种情况的发生, 用户可以配置 ssh, 启动 TCP 保活功能。ssh 还能够使用应用程序管理的保话功能。两种功能的行为模式不同, 特别是安全性方面 (参见 17.3 节了解细节)。

## 17.2 描述

保活功能在默认情况下是关闭的。TCP 连接的任何一端都可以请求打开这一功能。保话功能可以被设置在连接的一端、两端, 或者两端都没有。有几个配置参数可以用来控制保活功能的操作。如果在一段时间 (称为保活时间, keepalive time) 内连接处于非活动状态, 开启保活功能



的一端将向对方发送一个保活探测报文。如果发送端没有收到响应报文，那么经过一个已经提前配置好的保活时间间隔 (keepalive interval)，将继续发送保活探测报文，直到发送探测报文的次数达到保活探测数 (keepalive probe)，这时对方主机将被确认为不可到达，连接也将被中断。

保活探测报文为一个空报文段 (或只包含 1 字节)。它的序列号等于对方主机发送的 ACK 报文的最大序列号减 1。因这一序列号的数据段已经被成功接收，所以不会对到达的报文段造成影响，但探测报文返回的响应可以确定连接是否仍在工作。探测及其响应报文都不包含任何新的有效数据 (它是“垃圾”数据)，当它们丢失时也不会进行重传。[RFC1122] 指出，仅凭一个没有被响应的探测报文不能判断连接是否已经停止工作。这就是保活探测数参数需要被提前设置的原因。值得注意的是，一些 TCP 实现 (大部分是早期的 TCP 实现) 不会响应那些不包含“垃圾”数据的保活探测报文。

TCP 保活功能工作过程中，开启该功能的一端会发现对方处于以下四种状态之一：

1. 对方主机仍在工作，并且可以到达。对方的 TCP 响应正常，并且请求端也知道对方在正常工作。请求端将保活计时器重置 (重新设定为保活时间值)。如果在计时器超时之前有应用程序通过该连接传输数据，那么计时器将再次被设定为保活时间值。
2. 对方主机已经崩溃，包括已经关闭或者正在重新启动。这时对方的 TCP 将不会响应。请求端不会接收到响应报文，并在经过保活时间间隔指定的时间后超时。超时前，请求端会持续复送探测报文，一旦发送保活探测数指定次数的探测报文，如果请求端没有收到任何探测报文的响应，那么它将认为对方主机已经关闭，连接也将被断开。
3. 客户主机崩溃并且已重启。在这种情况下，请求端会收到一个对其保活探测报文的响应，但这个响应是一个重置报文段，请求端将会断开连接。
4. 对方主机仍在工作，但是由于某些原因不能到达请求端 (例如网络无法传输，而且可能使用 ICMP 通知也可能不通知对方这一事实)。这种情况与状态 2 相同，因为 TCP 不能区分状态 2 与状态 4，结果都是没有收到探测报文的响应。

请求端不必担心对方主机正常关闭然后重启 (不同于主机崩溃) 的情况。当系统关机时，所有的应用进程也会终止 (即对方的进程)，这会使对方的 TCP 发送一个 FIN。请求端接收到 FIN 后，会向请求端进程报告文件结束，并在检测到该状态后退出。

在第 1 种情况下，请求端的应用层不会觉察到保活探测的进行 (除非请求端应用层激活保活功能)。一切操作均在 TCP 层完成，因此这一过程对应用层是透明的，直至第 2、3、4 种情况中的某种情况发生。在这三种情况中，请求端的应用层将收到一个来自其 TCP 层的差错报告 (通常请求端已经向网络发出了读操作请求，并且等待来自对方的数据。如果保活功能返回了一个差错报告，则该差错报告将作为读操作请求的返回值返回给请求端)。在第 2 种情况下，差错是诸如“连接超时”之类的信息，而在第 3 种情况下则为“连接被对方重置”。第 4 种情况可能是连接超时，也可能是其他的错误信息。在下一节中我们将重点讨论这四种情况。

变量保活时间、保活时间间隔和保活探测数的设置通常是可以变更的。有些系统允许用户在每次建立连接时设置这些变量，还有一些系统规定只有在系统启动时才能设置（有的系统两者皆可）。在 Linux 系统中，这些变量分别对应 `sysctl` 变量 `net.ipv4.tcp_keepalive_time`、`net.ipv4.tcp_keepalive_intvl`、`net.ipv4.tcp_keepalive_probes`，默认设置是 7200 秒（2 小时）、75 秒和 9 次探测。

在 FreeBSD 和 Mac OS X 系统中，前两个变量对应 `sysctl` 变量 `net.inet.tcp.keepidle` 和 `net.inet.tcp.keepintvl`，默认设置力 7200 秒（2 小时）和 75000 毫秒（75 秒）。这两个系统还包含一个名为 `net.inet.tcp.always_keepalive` 的布尔变量。如果这个变量被激活，那么即使应用程序没有请求，所有 TCP 连接的保活功能都会被激活。探测次数被设定为固定值 8（FreeBSD 系统）或 9（Mac OS X 系统）。

在 Windows 系统中，可通过在系统键值下修改注册表项来设置变量：

```
HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
```

`KeepAliveTime` 保活时间默认为 7200 000 毫秒（2 小时），`KeepAliveInterval`（保活时间间隔）默认为 1000 毫秒（1 秒）。如果 10 个保活探测报文都没有响应，Windows 系统将终止连接。

值得注意的是，[RFC1122] 明确给出了用户使用保活功能的限制。保活时间值必须是可配置的，而且默认不能小于 2 小时。此外，除非应用层请求开启保活功能，否则不能使用该功能（而如果 `net.inet.tcp.always_keepalive` 变量被设置时会违反这一限制）。没有经过应用层的请求，Linux 系统不会提供保活功能，但是一个特殊库会被预先载入（即在载人普通共享库之前），从而实现该功能 [LKAJ]。

### 17.2.1 保活功能举例

现在详细讨论上一节提到的第 2、3、4 种情况，我们将在使用保活机制的前提下观察数据包的交换。第 1 种情况的操作将在观察其他几种情况的过程中涉及。

#### 另一端崩溃

我们想了解当服务器主机崩溃且没有重新启动时的过程。为了模拟这种情况，我们需要进行以下几个步骤：

1. 利用 Windows 客户端上的 `regedit` 程序，修改注册表键值，将 `KeepAliveTime` 设置 7000 毫秒（7 秒）。设置新的值，可能需要系统重新启动。
2. 在 Windows 客户端和一个已经开启 TCP 保活功能的 Linux 服务器之间建立 `ssh` 连接。
3. 确保数据可以通过该连接传输。

4. 观察客户端的 TCP 每 7 秒发送一个保活数据包，并且这些数据包都可以被服务器 TCP 接收到。
5. 保持服务器端的网线断开。这时客户端会认为服务器主机已经崩溃。
6. 我们预计，在确认连接断开之前，客户端会发送 10 个间隔为 1 秒的保活探测报文。

这里是客户端的交互输出结果：

```
C: \> ssh -O TCPKeepAlive=yes
10.0.1.1
(password prompt and login continues)
Write ralled:
Connection
reset by peer (about 15 seconds after disconnect)
```

图 17-1 是利用 Wireshark 工具的显示结果。在这个例子中，连接已经被建立。Wireshark 首先输出一个没有被识别的保活报文（数据包 1）。此时，Wireshark 还没有足够的数据包来处理，不能发现数据包 1 中的序列号小于接收端窗口的左边界，因此不能判断数据包 1 是一个保活报文。数据包 2 中包含一个 ACK 号，它可以使 Wireshark 对后续数据包中的序列号进行适当的处理。

这一连接大部分由保活报文和对应的响应报文组成。数据包 1、3、5、7、14、16、18、20 以及 22 31 都是保活报文。数据包 2、4、6、8、15、17、19、21 是这些报文相应的响应。如果保活报文得到响应，那么客户端将每隔 7 秒发送一次。而当保活报文没有被响应时，发送方将会根据 KeepAliveInterval 设定的默认值，转变为每隔 1 秒发送一个保活报文。这一情况发生在 62.120s 时刻，也就是第 23 个数据包发送时。发送方共发送了 10 个没有被响应的保活报文（数据包 22 31）。在这之后，客户端会断开连接，发送最后的重置报文段（数据包 32），但不会收到对方的任何响应。当连接断开时，用户将收到下面一段输出信息：

```
Write failed: Connection reset by peer
```

很明显连接已经断开，但是这种方式并不完全准确。事实上的确是发送端将连接断开的，但是发送端是基于缺少接收端的响应才做出这一判断的。

除了利用了保活机制之外，这一连接还有一些其他有趣的功能，这里做简要说明。首先，服务器使用了 DSACK 机制（第 14 章）。每一个 ACK 包含一个序列号区间，为之前接收到的序列号范围。其次，在 26.09s 时刻，有一个很小比特的数据交换。这个数据只代表了一个键盘键的按下。它被发送给服务器，服务器对其进行确认并回显。由于这一数据被加密，导致该数据包中的用户数据大小为 48 字节（第 18 章）。

有趣的是，回显的数据被发送了两遍。可以看到 11 号包为回显数据包，但是它没有被立即响应。回想一下第 14 章，Linux 系统 RTO 至少为 200 毫秒。这里我们可以看到，Linux 服务器在 200 毫秒之后重传了该数据，这次传输很快得到了客户端的响应。因为网络中无拥塞发生，所以基本不可能出现 11 号包丢失的情况。由于客户端的延迟响应，Linux 服务器进行了假重传。这种情况类似于第 15 章讨论过的 Nagle 算法与延时 ACK 的关系。从结果上看，这里发生了不必要的 200 毫秒延迟。

### 另一端崩溃并已重新启动

在这个例子中，我们需要观察当对方主机崩溃并且重启时会发生什么。这一次我们把 KeepAlive-Time 设置为 120000 毫秒（2 分钟），其他的初始设置与前面的例子相同。我们建立一个连接，然后等待 2 分钟，客户端会发送一个保活消息并成功接收到相应的响应。之后我们将服务器端的网络断开，并将服务器重新启动，最后将它重新连接到网络。我们预计下一个保活探测中，服务器将发出一个重置信息，因为服务器此刻不知道该连接的任何信息。图 17-2 显示了 Wireshark 记录下的整个过程。

在这个例子中，从 0.00s 到 3.46s，连接被建立，并且有少量的数据交换。之后连接进入空闲状态。经过 2 分钟（保活时间值）之后，客户端在 123.47s 时发送第一个保活探测报文，包含小于接收端窗口左边界的“垃圾”字节。该报文被确认，之后服务器被断开网络连接、重新启动、重新连接网络。在 243.47s 的时候，也就是 120s 之后，客户端发送了它的第二个保活探测报文。虽然服务器收到了探测报文，但是它不知道该连接的任何信息，所以它会返回一个重置报文段（数据包 18），通知客户端该连接已经无效，用户也会看到前面已经出现过的“Connection reset by peer”（连接被对方重置）的错误信息。

### 另一端不可达

在这种情况下，服务器没有崩溃，但是在保活探测报文发送间隔内无法到达。原因可能是中间路由器崩溃，或者会话连接出现故障，或者其他类似的情况。为了模拟这种情况，需要利用配置了保活功能的 sock 程序与 Web 服务器之间建立一条连接。我们使用一台 Mac OSx 系统的客户端和一台 LDAP 服务器（端口 389），它运行于网站 ldap.mit.edu。这里我们缩短了客户端的保活时间值（为了方便），然后打开连接，之后断开网络连接，然后来看看会有什么影响。下面是命令行和客户端的输出。

```
Mac# Byectl - net.inet.tcp.keepidle=75000
Mact
sock -k ldap.mit.edu 389
rec error: Operation timed out
about 14 minutes later
```

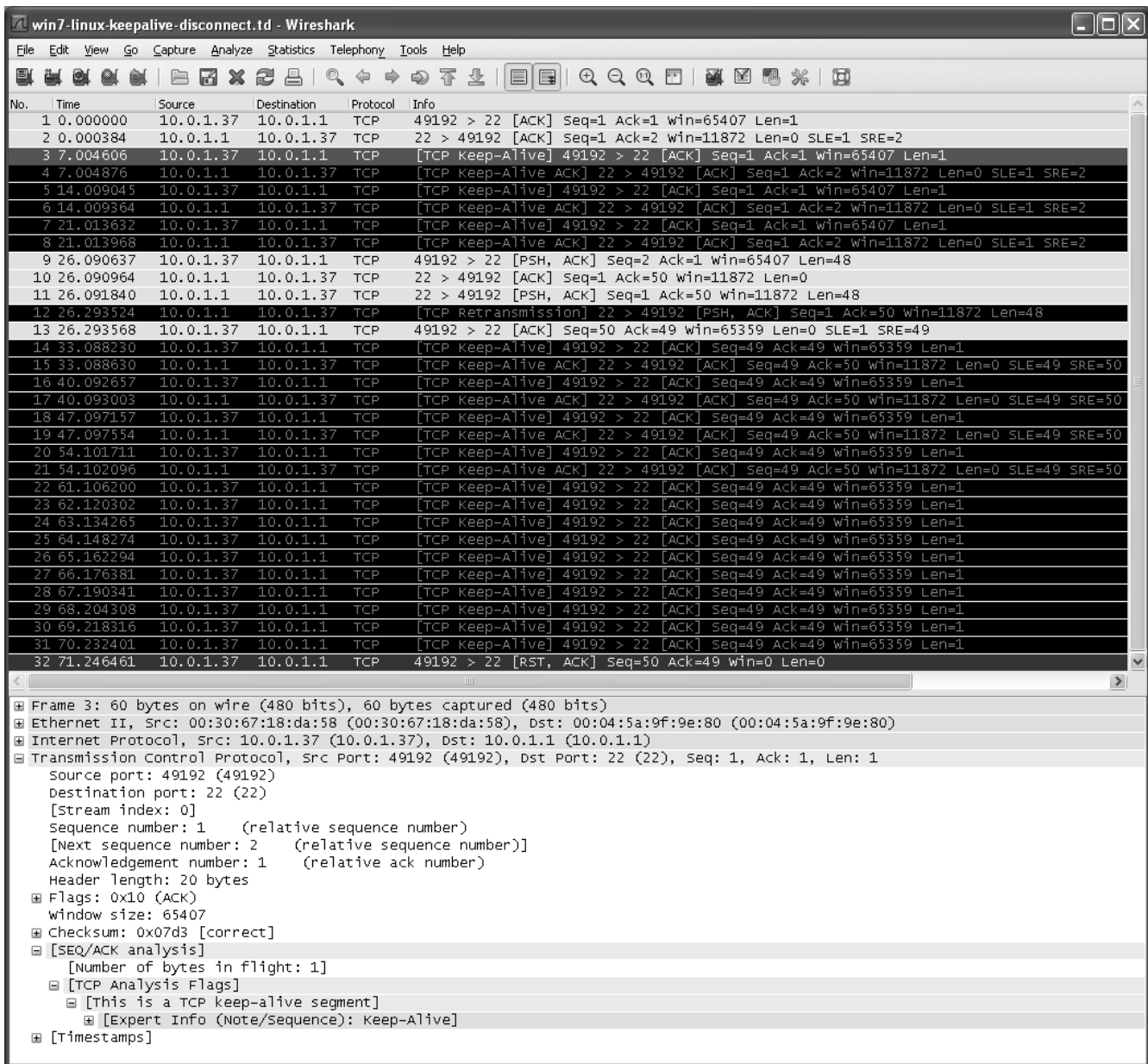


图 17.1: 连接空闲之后, TCP 保活报文每间隔 7 秒发送一次。每一个报文中包含一个小于已被确认数据的序列号。当网线断开 1 分钟后, 后续的保活报文就不会收到响应。客户端会发送 10 次保活报文, 如果都没有响应会将连接断开。断开连接时, 客户端会向服务器发送重置报文段 (服务器不会接收到)。这个例子还说明了服务器使用了 DSACK 机制, 客户端的延迟响应会导致假重传

图 17-3 显示了利用 Wireshark 记录的整个过程。

从图中我们可以看到一个完整的连接过程。在初始的三次握手之后, 连接保持空闲状态。在大约 75 秒时 (数据包 4) 客户端发送一个保活报文并得到确认响应。这个第一次发送的保活报文是由变量 `net.inet.top.keopidle` 的值来决定的。在这之后不久, 网络开始工作。由于连接的两端都没有传输数据, 所以在 150 秒 (75 秒之后, 等于变量 `net.inet.tcp.keepintvl` 的值) 时, 客

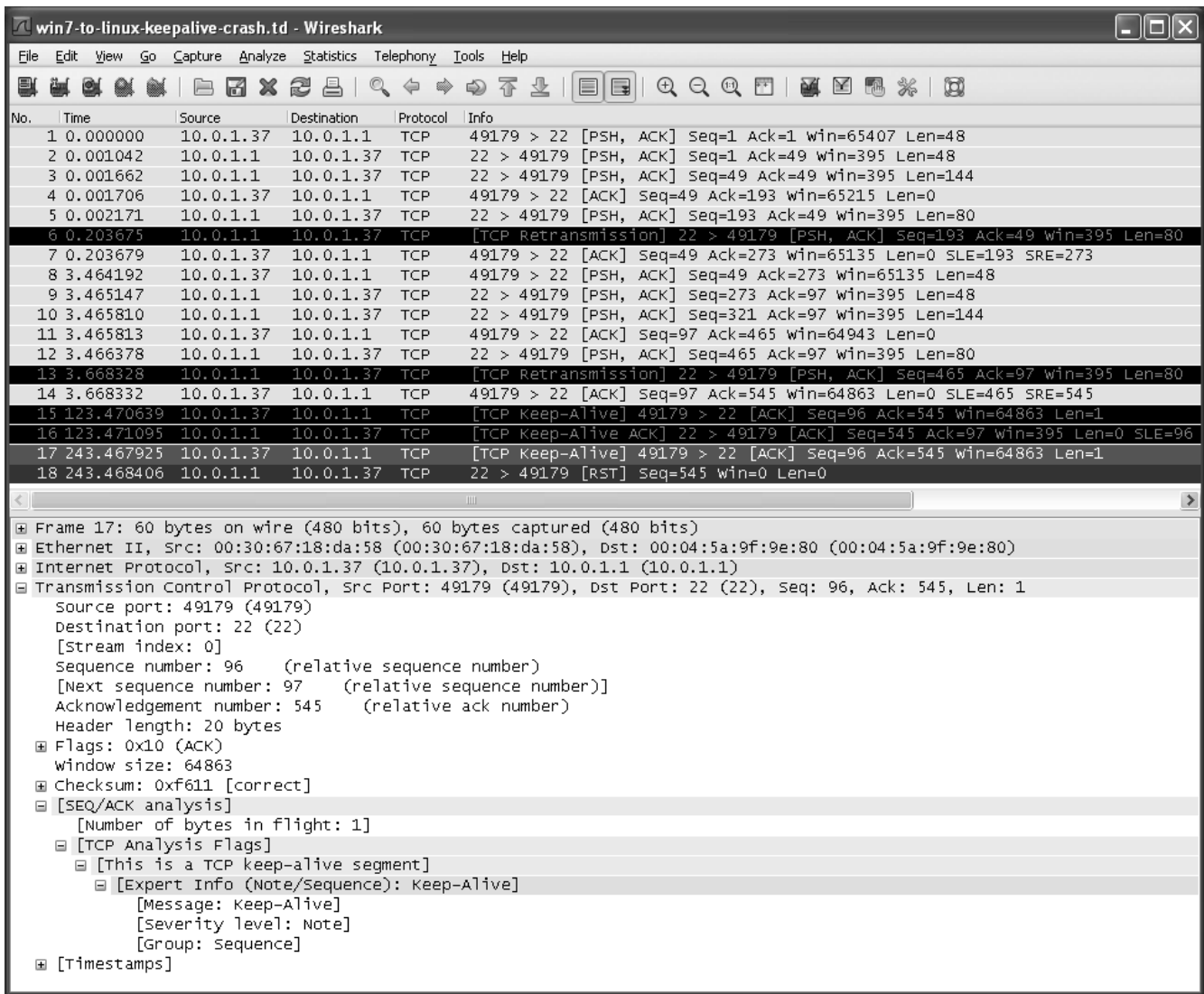


图 17.2: 在客户端发送保活报文的间隔中, 服务器已经重新启动。由于服务器不知道该连接的任何信息, 所以返回一个重置报文段

客户端会发送一个新的保活报文, 如数据包 7 14 所显示的重复操作。虽然服务器开启且正常工作, 但是客户端仍然不能接收到任何响应。最后, 当客户端第 9 次发送保活报文, 并且经过 75 秒之后仍没有收到确认响应时, 客户端结束该连接。连接中断时客户端会向服务器发送一个重置报文段 (数据包 15)。当然, 由于网络是断开的, 所以服务器不能接收到这个数据包。

像上面例子显示的一样, 当客户端的 TCP 不能利用保活报文与对方通信时, 客户端在终止该连接前还会做一定次数的尝试。这基本上与我们前面看到的另一端主机崩溃的情况相同。在大多数情况下, 发送端不能区分这两种状态。也会有一些例外情况, 如通过 ICMP 可以知道目的主机不可达, 或者由于其他的网络原因导致目的主机不可用。但是由于 ICMP 经常被阻塞, 所以很少能区分。因此, TCP 保活机制 (或者一些由应用层实现的相似的机制) 可以用来检测连

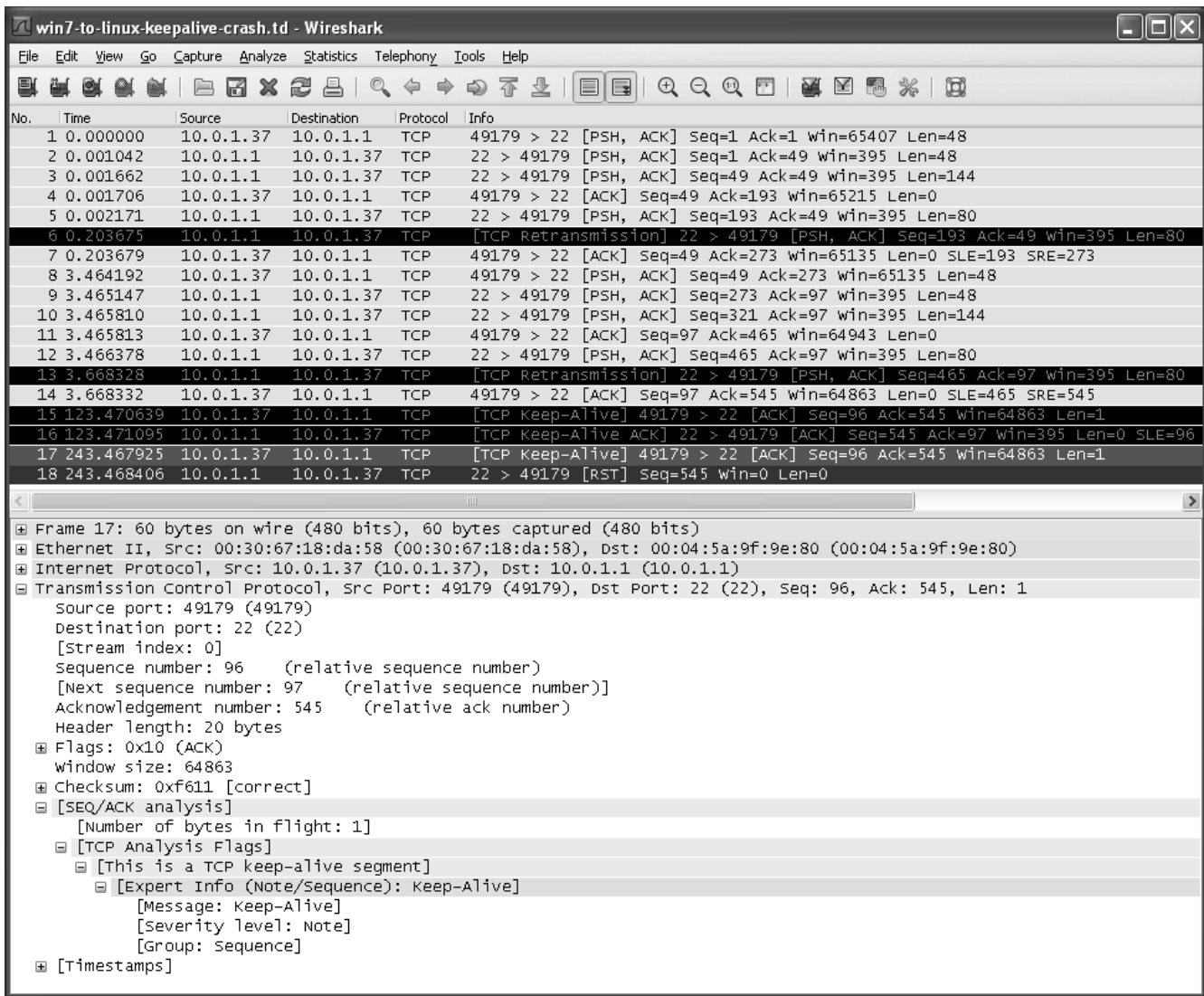


图 17.3: 第一次保活探测报文被确认之后、网络连接断开。客户端每隔 75 秒发送出一个新的探测报文。在发送了。次都没有响应之后，连接被断开。同时客户端向对方发送一个重置信号。对于客户端而言，这种情况与图 17-1 所示的服务器崩溃的情况相同

接断开的周期。

## 17.3 与 TCP 保活机制相关的攻击

之前提到过，ssh（第 2 版）中含有一种应用层的保活机制，称为服务器保活报文和客户端保活报文。与 TCP 保活报文的区别在于，它们是在应用层通过一条加密的链路传输的，而且这些报文中包含数据。TCP 保活报文中不包含任何用户数据，所以它最多只进行有限的加密。因此 TCP 保活机制容易受到欺骗攻击。当受到欺骗攻击时，在相当长的一段时间内，受害主机必

须维护不必要的会话资源。

还有一些相对次要的问题。TCP 保活机制的计时器是由之前提到的不同配置参数决定的，而不是用于数据传输的重传计时器。对于被动的观察者来说，他们能够注意到保活报文的有在，并观察保活报文发送的间隔时间，从而了解系统的配置参数（可能获取发送系统类别信息，称为系统指纹）或者网络的拓扑结构（即下一跳路由器是否能够转发数据流量）。这些问题在某些环境下是非常重要的。

## 17.4 总结

如前所述，保活功能存在一定争议性。协议专家仍然在不断争论该功能是否应该属于传输层，还是全部交由应用层处理。现在所有主流 TCP 版本都实现了保活功能。应用层可以选择是否开启这一功能来建立连接。开启保活功能，即使在没有应用层数据传输的情况下，仍能帮助服务器判断没有响应的客户端，也可以帮助客户端保持连接活跃性（例如保持 NAT 状态活跃）。

若某个连接长时间处于空闲状态（通常这段时间设定为 2 小时），在该连接的一端会发送一个探测数据包（虽然这个数据包可以不含任何数据，但通常情况下会包含“垃圾”字节），从而实现保活功能。可能会发生 4 种不同的情况：另一端仍在工作；另一端崩溃；另一端崩溃并且已经重新启动；另一端当前无法到达。我们分别举了一个例子来观察这 4 种情况。

在前两个例子中，如果没有使用保活功能，而且也没有应用层的计时器或者计时器未被激活，那么 TCP 将不会知道另一端已经崩溃（或已经崩溃但已重新启动）。在最后一个例子中，连接的两端都没有出现差错，而连接最终却被断开了。在使用保活功能的时候，我们必须意识到这一功能的限制，并且考虑这种处理方式是否是我们所期望的。

针对保活机制的攻击主要包括两种：一种是使系统长时间地维护不必要的会话资源，另一种是获得端系统隐藏的一些信息（虽然这些信息对于攻击者而言可能实用性有限）。此外，由于默认情况下 TCP 不会对保活报文进行加密，所以保活探测报文和确认报文都有可能被利用。然而，对于应用层的保活机制（例如 ssh），这些报文都会被加密，所以也就不会出现上述情况。

## 17.5 参考文献

[LKA] <http://libkeepalive.sourceforge.net> [RFC1122] R. Braden, ed., "Requirements for Internet Hosts," Internet RFC 1122, Oct. 1989.