结构。该实现也可能是以各种过程形式出现，每个过程实现了在响应各种可能出现的事件时要采取的动作。在这种**基于事件的编程**（event-based programming）方式中，这些过程要么被协议栈中的其他过程调用，要么作为一次中断的结果。在发送方，这些事件包括：①来自上层实体的调用去调用 rdt_send()；②定时器中断；③报文到达时，来自下层的调用去调用 rdt_rcv()。本章后面的编程作业会使你有机会在一个模拟网络环境中实际实现这些例程，但该环境却是真实的。

这里我们注意到，GBN 协议中综合了我们将在 3.5 节中学习 TCP 可靠数据传输构件时遇到的所有技术。这些技术包括使用序号、累积确认、检验和以及超时/重传操作。

3.4.4　选择重传

在图 3-17 中，GBN 协议潜在地允许发送方用多个分组“填充流水线”，因此避免了停等协议中所提到的信道利用率问题。然而，GBN 本身也有一些情况存在着性能问题。尤其是当窗口长度和带宽时延积都很大时，在流水线中会有很多分组更是如此。单个分组的差错就能够引起 GBN 重传大量分组，许多分组根本没有必要重传。随着信道差错率的增加，流水线可能会被这些不必要重传的分组所充斥。想象一下，在我们口述消息的例子中，如果每次有一个单词含糊不清，其前后 1000 个单词（例如，窗口长度为 1000 个单词）不得不被重传的情况。此次口述会由于这些反复述说的单词而变慢。

顾名思义，选择重传（SR）协议通过让发送方仅重传那些它怀疑在接收方出错（即丢失或受损）的分组而避免了不必要的重传。这种个别的、按需的重传要求接收方逐个地确认正确接收的分组。再次用窗口长度 N 来限制流水线中未完成、未被确认的分组数。然而，与 GBN 不同的是，发送方已经收到了对窗口中某些分组的 ACK。图 3-23 显示了 SR 发送方看到的序号空间。图 3-24 详细描述了 SR 发送方所采取的动作。

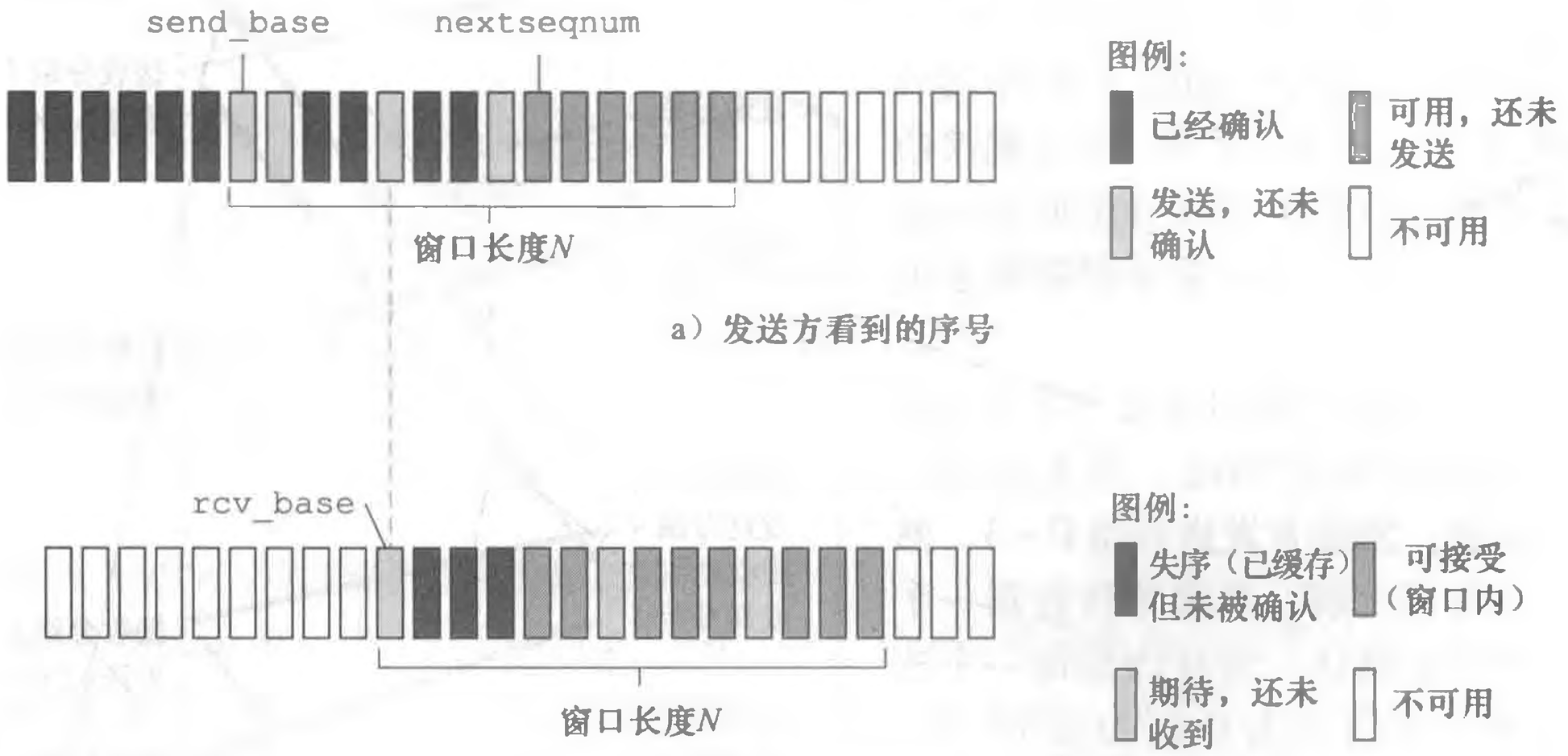

图 3-23　选择重传（SR）发送方与接收方的序号空间

SR 接收方将确认一个正确接收的分组而不管其是否按序。失序的分组将被缓存直到所有丢失分组（即序号更小的分组）皆被收到为止，这时才可以将一批分组按序交付给上层。图 3-25 详细列出了 SR 接收方所采用的各种动作。图 3-26 给出了一个例子以说明出现丢包时 SR 的操作。值得注意的是，在图 3-26 中接收方初始时缓存了分组 3、4、5，并

在最终收到分组 2 时，才将它们一并交付给上层。

1. 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送；否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。

2. 超时。定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。可以使用单个硬件定时器模拟多个逻辑定时器的操作［Varghese 1997］。

3. 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。如果该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

图 3-24 SR 发送方的事件与动作

1. 序号在［rcv_base，rcv_base + N - 1］内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收窗口的基序号（图 3-23 中的 rcv_base），则该分组以及以前缓存的序号连续的（起始于 rcv_base 的）分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。举例子来说，考虑一下图 3-26。当收到一个序号为 rcv_base = 2的分组时，该分组及分组 3、4、5 可被交付给上层。

2. 序号在［rcv_base - N，rcv_base - 1］内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前已确认过的分组。

3. 其他情况。忽略该分组。

图 3-25 SR 接收方的事件与动作

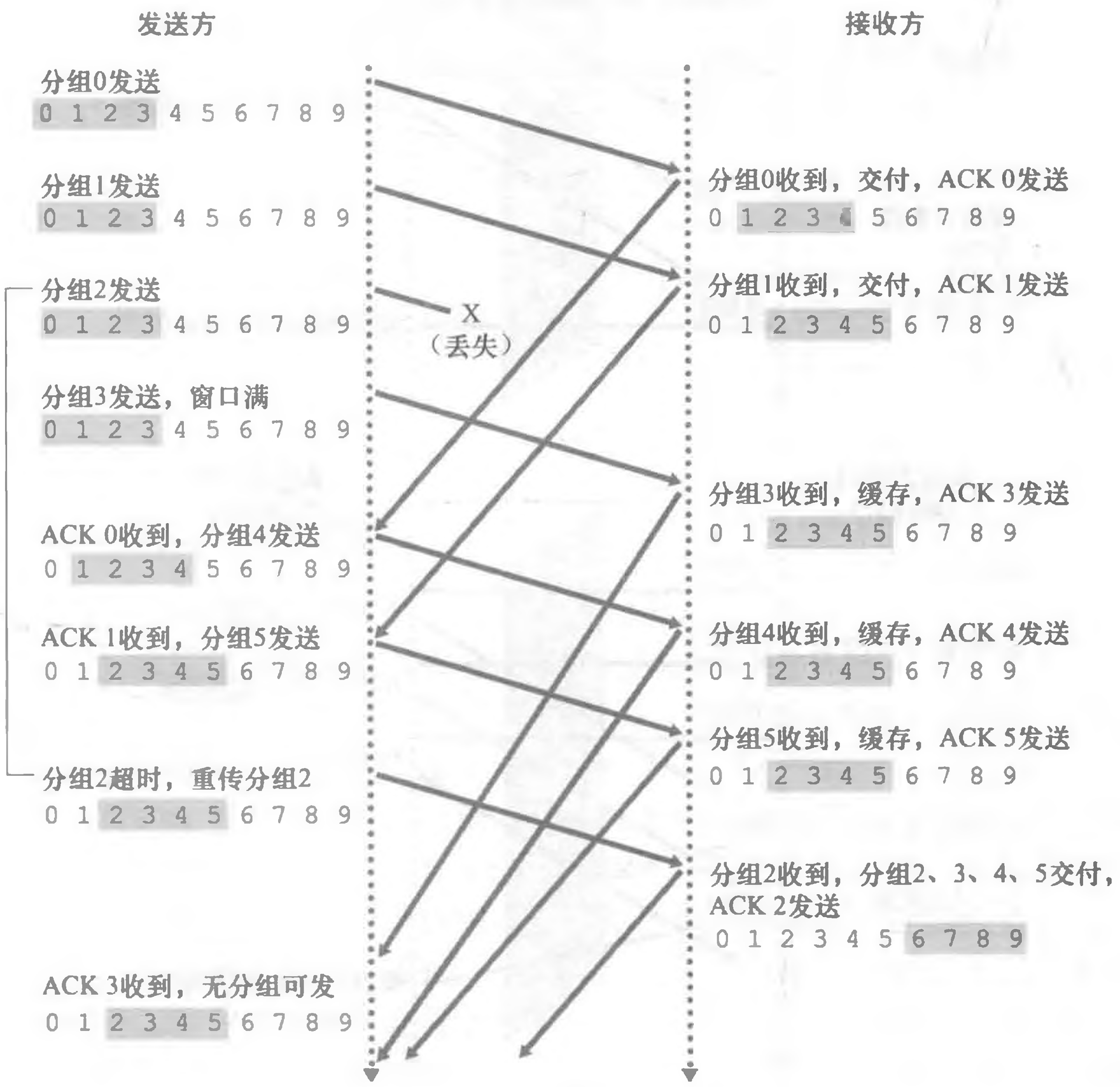

图 3-26 SR 操作

注意到图3-25中的第二步很重要，接收方重新确认（而不是忽略）已收到过的那些序号小于当前窗口基序号的分组。你应该理解这种重新确认确实是需要的。例如，给定在图3-23中所示的发送方和接收方的序号空间，如果分组send_base的ACK没有从接收方传播回发送方，则发送方最终将重传分组send_base，即使显然（对我们而不是对发送方来说!）接收方已经收到了该分组。如果接收方不确认该分组，则发送方窗口将永远不能向前滑动！这个例子说明了SR协议（和很多其他协议一样）的一个重要方面。对于哪些分组已经被正确接收，哪些没有，发送方和接收方并不总是能看到相同的结果。对SR协议而言，这就意味着发送方和接收方的窗口并不总是一致。

当我们面对有限序号范围的现实时，发送方和接收方窗口间缺乏同步会产生严重的后果。考虑下面例子中可能发生的情况，该例有包括4个分组序号0、1、2、3的有限序号范围且窗口长度为3。假定发送了分组0至2，并在接收方被正确接收且确认了。此时，接收方窗口落在第4、5、6个分组上，其序号分别为3、0、1。现在考虑两种情况。在第一种情况下，如图3-27a所示，对前3个分组的ACK丢失，因此发送方重传这些分组。因此，接收方下一步要接收序号为0的分组，即第一个发送分组的副本。

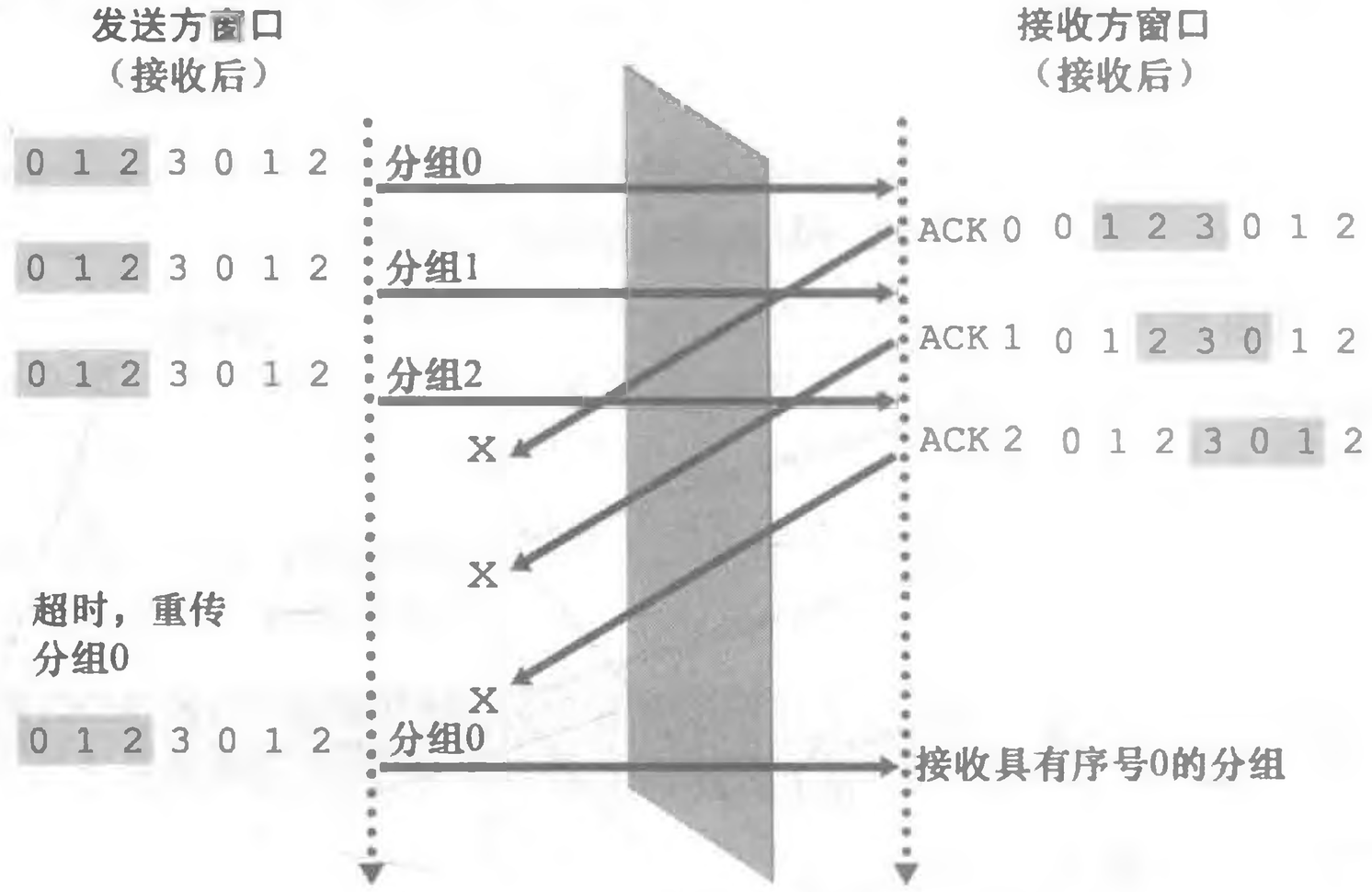

图3-27　SR接收方窗口太大的困境：是一个新分组还是一次重传

在第二种情况下，如图3-27b所示，对前3个分组的ACK都被正确交付。因此发送方向前移动窗口并发送第4、5、6个分组，其序号分别为3、0、1。序号为3的分组丢失，但序号为0的分组到达（一个包含新数据的分组）。

现在考虑一下图3-27中接收方的观点，在发送方和接收方之间有一个假想的帘子，因为接收方不能“看见”发送方采取的动作。接收方所能观察到的是它从信道中收到的以及它向信道中发出报文序列。就其所关注的而言，图3-27中的两种情况是等同的。没有办法区分是第1个分组的重传还是第5个分组的初次传输。显然，窗口长度比序号空间小1时协议无法工作。但窗口必须多小呢？本章后面的一道习题请你说明为何对于SR协议而言，窗口长度必须小于或等于序号空间大小的一半。

在本书配套的网站上，可以找到一个模仿SR协议运行的Java小程序。尝试进行你以前对GBN Java小程序所进行的相同的实验。这些结果与你期望的一致吗？

至此我们结束了对可靠数据传输协议的讨论。我们已涵盖许多基础知识，并介绍了多种机制，这些机制可一起提供可靠数据传输。表3-1总结这些机制。既然我们已经学习了所有这些运行中的机制，并能看到“全景”，我们建议你再复习一遍本节内容，看看这些机制是怎样逐步被添加进来，以涵盖复杂性渐增的（现实的）连接发送方与接收方的各种信道模型的，或者如何改善协议性能的。

表3-1 可靠数据传输机制及其用途的总结

机制	用途和说明
检验和	用于检测在一个传输分组中的比特错误
定时器	用于超时/重传一个分组，可能因为该分组（或其ACK）在信道中丢失了。由于当一个分组延时但未丢失（过早超时），或当一个分组已被接收方收到但从接收方到发送方的ACK丢失时，可能产生超时事件，所以接收方可能会收到一个分组的多个冗余副本
序号	用于为从发送方流向接收方的数据分组按顺序编号。所接收分组的序号间的空隙可使接收方检测出丢失的分组。具有相同序号的分组可使接收方检测出一个分组的冗余副本
确认	接收方用于告诉发送方一个分组或一组分组已被正确地接收到了。确认报文通常携带着被确认的分组或多个分组的序号。确认可以是逐个的或累积的，这取决于协议
否定确认	接收方用于告诉发送方某个分组未被正确地接收。否定确认报文通常携带着未被正确接收的分组的序号
窗口、流水线	发送方也许被限制仅发送那些序号落在一个指定范围内的分组。通过允许一次发送多个分组但未被确认，发送方的利用率可在停等操作模式的基础上得到增加。我们很快将会看到，窗口长度可根据接收方接收和缓存报文的能力、网络中的拥塞程度或两者情况来进行设置

我们通过考虑在底层信道模型中的一个遗留假设来结束对可靠数据传输协议的讨论。前面讲过，我们曾假定分组在发送方与接收方之间的信道中不能被重新排序。这在发送方与接收方由单段物理线路相连的情况下，通常是一个合理的假设。然而，当连接两端的“信道”是一个网络时，分组重新排序是可能会发生的。分组重新排序的一个表现就是，一个具有序号或确认号 x 的分组的旧副本可能会出现，即使发送方或接收方的窗口中都没有包含 x。对于分组重新排序，信道可被看成基本上是在缓存分组，并在将来任意时刻自然地释放出这些分组。由于序号可以被重新使用，那么必须小心，以免出现这样的冗余分组。实际应用中采用的方法是，确保一个序号不被重新使用，直到发送方“确信”任何先前发送的序号为 x 的分组都不再在网络中为止。通过假定一个分组在网络中的“存活”时间不会超过某个固定最大时间量来做到这一点。在高速网络的TCP扩展中，最长的分组寿命被假定为大约3分钟［RFC 1323］。［Sunshine 1978］描述了一种使用序号的方法，它能

够完全避免重新排序问题。

3.5　面向连接的运输：TCP

既然我们已经学习了可靠数据传输的基本原理，我们就可以转而学习 TCP 了。TCP 是因特网运输层的面向连接的可靠的运输协议。我们在本节中将看到，为了提供可靠数据传输，TCP 依赖于前一节所讨论的许多基本原理，其中包括差错检测、重传、累积确认、定时器以及用于序号和确认号的首部字段。TCP 定义在 RFC 793、RFC 1122、RFC 1323、RFC 2018 以及 RFC 2581 中。

3.5.1　TCP 连接

TCP 被称为是**面向连接的**（connection-oriented），这是因为在一个应用进程可以开始向另一个应用进程发送数据之前，这两个进程必须先相互“握手”，即它们必须相互发送某些预备报文段，以建立确保数据传输的参数。作为 TCP 连接建立的一部分，连接的双方都将初始化与 TCP 连接相关的许多 TCP 状态变量（其中的许多状态变量将在本节和 3.7 节中讨论）。

历史事件

Vinton Cerf 和 Robert Kahn 与 TCP/IP

在 20 世纪 70 年代早期，分组交换网开始飞速增长，而因特网的前身 ARPAnet 也只是当时众多分组交换网中的一个。这些网络都有它们各自的协议。两个研究人员 Vinton Cerf 和 Robert Kahn 认识到互联这些网络的重要性，发明了沟通网络的 TCP/IP 协议，该协议代表**传输控制协议/网际协议**（Transmission Control Protocol/Internet Protocol）。虽然 Cerf 和 Kahn 开始时把该协议看成是单一的实体，但是后来将它分成单独运行的两个部分：TCP 和 IP。Cerf 和 Kahn 在 1974 年 5 月的《IEEE Transactions on Communications Technology》杂志上发表了一篇关于 TCP/IP 的论文［Cerf 1974］。

TCP/IP 协议是当今因特网的支柱性协议，但它的发明先于 PC、工作站、智能手机和平板电脑，先于以太网、电缆、DSL、WiFi 和其他接入网技术的激增，先于 Web、社交媒体和流式视频等。Cerf 和 Kahn 预见到了对于联网协议的需求，一方面为行将定义的应用提供广泛的支持，另一方面允许任何主机与链路层协议互操作。

2004 年，Cerf 和 Kahn 由于“联网方面的开创性工作（包括因特网的基本通信协议 TCP/IP 的设计和实现）以及联网方面富有才能的领导”而获得 ACM 图灵奖，该奖项被认为是“计算机界的诺贝尔奖”。

这种 TCP“连接”不是一条像在电路交换网络中的端到端 TDM 或 FDM 电路。相反，该“连接”是一条逻辑连接，其共同状态仅保留在两个通信端系统的 TCP 程序中。前面讲过，由于 TCP 协议只在端系统中运行，而不在中间的网络元素（路由器和链路层交换机）中运行，所以中间的网络元素不会维持 TCP 连接状态。事实上，中间路由器对 TCP 连接完全视而不见，它们看到的是数据报，而不是连接。

台主机上的进程 B 存在一条 TCP 连接，那么应用层数据就可在从进程 B 流向进程 A 的同时，也从进程 A 流向进程 B。TCP 连接也总是**点对点**（point-to-point）的，即在单个发送方与单个接收方之间的连接。所谓“多播”（参见本书的在线补充材料），即在一次发送操作中，从一个发送方将数据传送给多个接收方，这种情况对 TCP 来说是不可能的。对于 TCP 而言，两台主机是一对，而 3 台主机则太多了！

我们现在来看看 TCP 连接是怎样建立的。假设运行在某台主机上的一个进程想与另一台主机上的一个进程建立一条连接。前面讲过，发起连接的这个进程被称为客户进程，而另一个进程被称为服务器进程。该客户应用进程首先要通知客户运输层，它想与服务器上的一个进程建立一条连接。2.7.2 节讲过，一个 Python 客户程序通过发出下面的命令来实现此目的。

```
clientSocket.connect((serverName,serverPort))
```

其中 serverName 是服务器的名字，serverPort 标识了服务器上的进程。客户上的 TCP 便开始与服务器上的 TCP 建立一条 TCP 连接。我们将在本节后面更为详细地讨论连接建立的过程。现在知道下列事实就可以了：客户首先发送一个特殊的 TCP 报文段，服务器用另一个特殊的 TCP 报文段来响应，最后，客户再用第三个特殊报文段作为响应。前两个报文段不承载“有效载荷”，也就是不包含应用层数据；而第三个报文段可以承载有效载荷。由于在这两台主机之间发送了 3 个报文段，所以这种连接建立过程常被称为**三次握手**（three-way handshake）。

一旦建立起一条 TCP 连接，两个应用进程之间就可以相互发送数据了。我们考虑一下从客户进程向服务器进程发送数据的情况。如 2.7 节中所述，客户进程通过套接字（该进程之门）传递数据流。数据一旦通过该门，它就由客户中运行的 TCP 控制了。如图 3-28 所示，TCP 将这些数据引导到该连接的**发送缓存**（send buffer）里，发送缓存是发起三次握手期间设置的缓存之一。接下来 TCP 就会不时从发送缓存里取出一块数据，并将数据传递到网络层。有趣的是，在 TCP 规范［RFC 793］中却没提及 TCP 应何时实际发送缓存里的数据，只是描述为“TCP 应该在它方便的时候以报文段的形式发送数据”。TCP 可从缓存中取出并放入报文段中的数据数量受限于**最大报文段长度**（Maximum Segment Size，MSS）。MSS 通常根据最初确定的由本地发送主机发送的最大链路层帧长度（即所谓的**最大传输单元**（Maximum Transmission Unit，MTU））来设置。设置该 MSS 要保证一个 TCP 报文段（当封装在一个 IP 数据报中）加上 TCP/IP 首部长度（通常 40 字节）将适合单个链路层帧。以太网和 PPP 链路层协议都具有 1500 字节的 MTU，因此 MSS 的典型值为 1460 字节。已经提出了多种发现路径 MTU 的方法，并基于路径 MTU 值设置 MSS（路径 MTU 是指能在从源到目的地的所有链路上发送的最大链路层帧［RFC 1191］）。注意到 MSS 是指在报文段里应用层数据的最大长度，而不是指包括首部的 TCP 报文段的最大长度。（该术语很容易混淆，但是我们不得不采用它，因为它已经根深蒂固了。）

TCP 为每块客户数据配上一个 TCP 首部，从而形成多个 **TCP 报文段**（TCP segment）。这些报文段被下传给网络层，网络层将其分别封装在网络层 IP 数据报中。然后这些 IP 数据报被发送到网络中。当 TCP 在另一端接收到一个报文段后，该报文段的数据就被放入该 TCP 连接的接收缓存中，如图 3-28 中所示。应用程序从此缓存中读取数据流。该连接的每一端都有各自的发送缓存和接收缓存。（读者可以参见 http://www.awl.com/kurose-ross 处的在线流控制 Java 小程序，它提供了关于发送缓存和接收缓存的一个动画演示。）

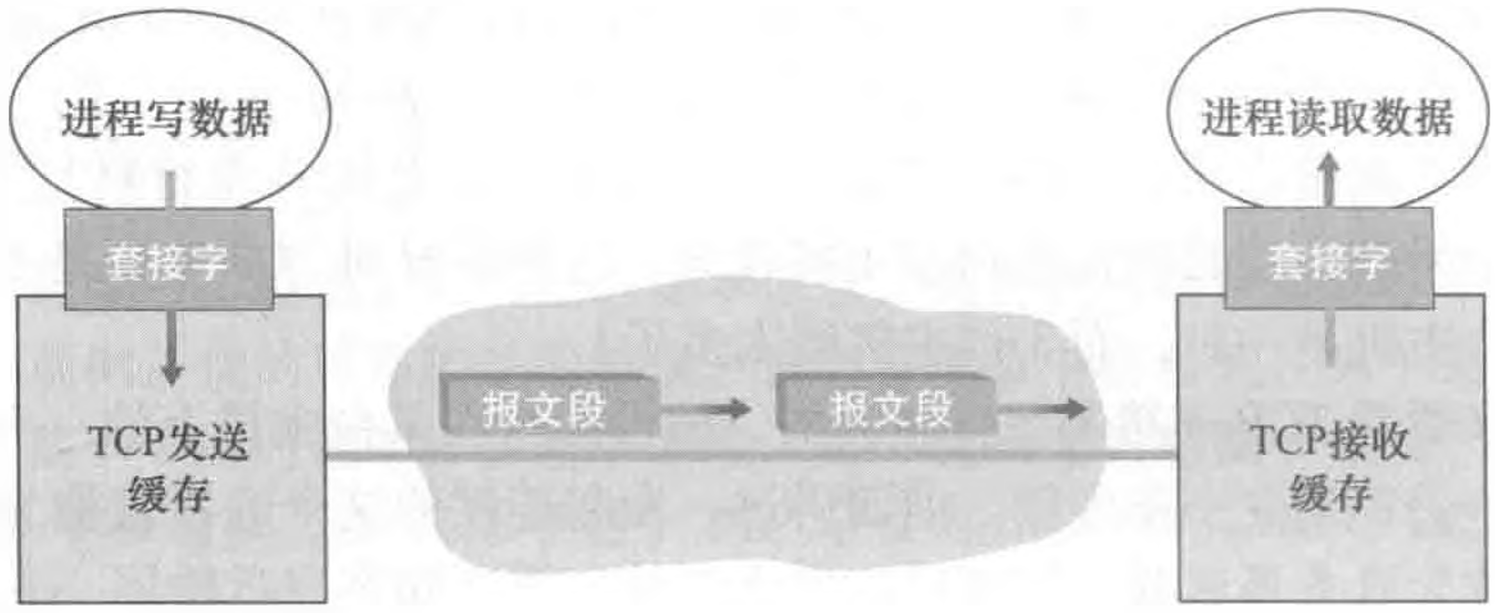

图 3-28　TCP 发送缓存和接收缓存

从以上讨论中我们可以看出，TCP 连接的组成包括：一台主机上的缓存、变量和与进程连接的套接字，以及另一台主机上的另一组缓存、变量和与进程连接的套接字。如前面讲过的那样，在这两台主机之间的网络元素（路由器、交换机和中继器）中，没有为该连接分配任何缓存和变量。

3.5.2　TCP 报文段结构

简要地了解了 TCP 连接后，我们研究一下 TCP 报文段结构。TCP 报文段由首部字段和一个数据字段组成。数据字段包含一块应用数据。如前所述，MSS 限制了报文段数据字段的最大长度。当 TCP 发送一个大文件，例如某 Web 页面上的一个图像时，TCP 通常是将该文件划分成长度为 MSS 的若干块（最后一块除外，它通常小于 MSS）。然而，交互式应用通常传送长度小于 MSS 的数据块。例如，对于像 Telnet 这样的远程登录应用，其 TCP 报文段的数据字段经常只有一个字节。由于 TCP 的首部一般是 20 字节（比 UDP 首部多 12 字节），所以 Telnet 发送的报文段也许只有 21 字节长。

图 3-29 显示了 TCP 报文段的结构。与 UDP 一样，首部包括**源端口号**和**目的端口号**，它被用于多路复用/分解来自或送到上层应用的数据。另外，同 UDP 一样，TCP 首部也包括**检验和字段**（checksum field）。

图 3-29　TCP 报文段结构

TCP 报文段首部还包含下列字段：

- 32 比特的**序号字段**（sequence number field）和 32 比特的**确认号字段**（acknowledgment number field）。这些字段被 TCP 发送方和接收方用来实现可靠数据传输服务，讨论见后。
- 16 比特的**接收窗口字段**（receive window field），该字段用于流量控制。我们很快就会看到，该字段用于指示接收方愿意接受的字节数量。
- 4 比特的**首部长度字段**（header length field），该字段指示了以 32 比特的字为单位的 TCP 首部长度。由于 TCP 选项字段的原因，TCP 首部的长度是可变的。（通常，选项字段为空，所以 TCP 首部的典型长度是 20 字节。）

- 可选与变长的**选项字段**（options field），该字段用于发送方与接收方协商最大报文段长度（MSS）时，或在高速网络环境下用作窗口调节因子时使用。首部字段中还定义了一个时间戳选项。可参见 RFC 854 和 RFC 1323 了解其他细节。
- 6 比特的**标志字段**（flag field）。ACK 比特用于指示确认字段中的值是有效的，即该报文段包括一个对已被成功接收报文段的确认。RST、SYN 和 FIN 比特用于连接建立和拆除，我们将在本节后面讨论该问题。在明确拥塞通告中使用了 CWR 和 ECE 比特，如 3.7.2 节中讨论的那样。当 PSH 比特被置位时，就指示接收方应立即将数据交给上层。最后，URG 比特用来指示报文段里存在着被发送端的上层实体置为“紧急”的数据。紧急数据的最后一个字节由 16 比特的**紧急数据指针字段**（urgent data pointer field）指出。当紧急数据存在并给出指向紧急数据尾指针的时候，TCP 必须通知接收端的上层实体。（在实践中，PSH、URG 和紧急数据指针并没有使用。为了完整性起见，我们才提到这些字段。）

作为教师的经验是，学生有时觉得分组格式的讨论相当枯燥，也许有些乏味。特别是如果你和我们一样都喜爱乐高玩具，有关 TCP 首部的有趣和新颖的讨论请参见［Pomeranz 2010］。

1. 序号和确认号

TCP 报文段首部中两个最重要的字段是序号字段和确认号字段。这两个字段是 TCP 可靠传输服务的关键部分。但是在讨论这两个字段是如何用于提供可靠数据传输之前，我们首先来解释一下 TCP 在这两个字段中究竟放置了什么。

TCP 把数据看成一个无结构的、有序的字节流。我们从 TCP 对序号的使用上可以看出这一点，因为序号是建立在传送的字节流之上，而不是建立在传送的报文段的序列之上。**一个报文段的序号**（sequence number for a segment）因此是该报文段首字节的字节流编号。举例来说，假设主机 A 上的一个进程想通过一条 TCP 连接向主机 B 上的一个进程发送一个数据流。主机 A 中的 TCP 将隐式地对数据流中的每一个字节编号。假定数据流由一个包含 500 000 字节的文件组成，其 MSS 为 1000 字节，数据流的首字节编号是 0。如图 3-30 所示，该 TCP 将为该数据流构建 500 个报文段。给第一个报文段分配序号 0，第二个报文段分配序号 1000，第三个报文段分配序号 2000，以此类推。每一个序号被填入到相应 TCP 报文段首部的序号字段中。

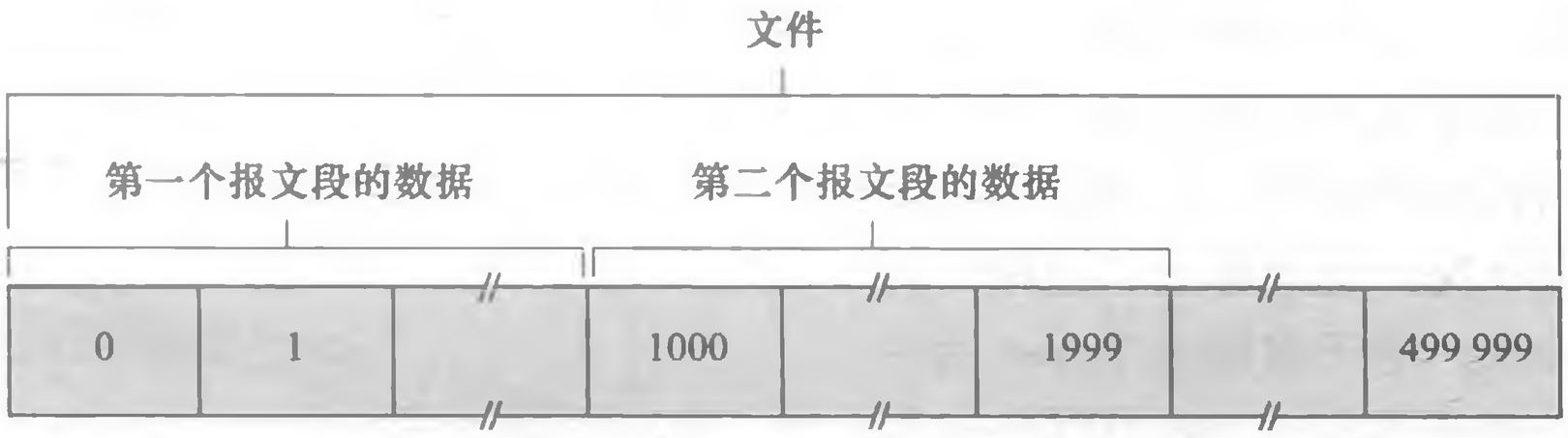

图 3-30 文件数据划分成 TCP 报文段

现在我们考虑一下确认号。确认号要比序号难处理一些。前面讲过，TCP 是全双工的，因此主机 A 在向主机 B 发送数据的同时，也许也接收来自主机 B 的数据（都是同一条 TCP 连接的一部分）。从主机 B 到达的每个报文段中都有一个序号用于从 B 流向 A 的数据。主机 A 填充进报文段的确认号是主机 A 期望从主机 B 收到的下一字节的序号。看一些例子有助于理解实际发生的事情。假设主机 A 已收到了来自主机 B 的编号为 0 ~ 535 的所有字节，同时

假设它打算发送一个报文段给主机 B。主机 A 等待主机 B 的数据流中字节 536 及之后的所有字节。所以主机 A 就会在它发往主机 B 的报文段的确认号字段中填上 536。

再举一个例子，假设主机 A 已收到一个来自主机 B 的包含字节 0 ~ 535 的报文段，以及另一个包含字节 900 ~ 1000 的报文段。由于某种原因，主机 A 还没有收到字节 536 ~ 899 的报文段。在这个例子中，主机 A 为了重新构建主机 B 的数据流，仍在等待字节 536（和其后的字节）。因此，A 到 B 的下一个报文段将在确认号字段中包含 536。因为 TCP 只确认该流中至第一个丢失字节为止的字节，所以 TCP 被称为提供**累积确认**（cumulative acknowledgment）。

最后一个例子也会引发一个重要而微妙的问题。主机 A 在收到第二个报文段（字节 536 ~ 899）之前收到第三个报文段（字节 900 ~ 1000）。因此，第三个报文段失序到达。该微妙的问题是：当主机在一条 TCP 连接中收到失序报文段时该怎么办？有趣的是，TCP RFC 并没有为此明确规定任何规则，而是把这一问题留给实现 TCP 的编程人员去处理。他们有两个基本的选择：①接收方立即丢弃失序报文段（如前所述，这可以简化接收方的设计）；②接收方保留失序的字节，并等待缺少的字节以填补该间隔。显然，后一种选择对网络带宽而言更为有效，是实践中采用的方法。

在图 3-30 中，我们假设初始序号为 0。事实上，一条 TCP 连接的双方均可随机地选择初始序号。这样做可以减少将那些仍在网络中存在的来自两台主机之间先前已终止的连接的报文段，误认为是后来这两台主机之间新建连接所产生的有效报文段的可能性（它碰巧与旧连接使用了相同的端口号）[Sunshine 1978]。

2. Telnet：序号和确认号的一个学习案例

Telnet 由 RFC 854 定义，它现在是一个用于远程登录的流行应用层协议。它运行在 TCP 之上，被设计成可在任意一对主机之间工作。Telnet 与我们第 2 章讨论的批量数据传输应用不同，它是一个交互式应用。我们在此讨论一个 Telnet 例子，因为该例子很好地阐述 TCP 的序号与确认号。我们注意到许多用户现在更愿意采用 SSH 协议而不是 Telnet，因为在 Telnet 连接中发送的数据（包括口令!）是没有加密的，使得 Telnet 易于受到窃听攻击（如在 8.7 节中讨论的那样）。

假设主机 A 发起一个与主机 B 的 Telnet 会话。因为是主机 A 发起该会话，因此它被标记为客户，而主机 B 被标记为服务器。（在客户端的）用户键入的每个字符都会被发送至远程主机；远程主机将回送每个字符的副本给客户，并将这些字符显示在 Telnet 用户的屏幕上。这种“回显”（echo back）用于确保由 Telnet 用户发送的字符已经被远程主机收到并在远程站点上得到处理。因此，在从用户击键到字符被显示在用户屏幕上这段时间内，每个字符在网络中传输了两次。

现在假设用户输入了一个字符 'C'，然后喝起了咖啡。我们考察一下在客户与服务器之间发送的 TCP 报文段。如图 3-31

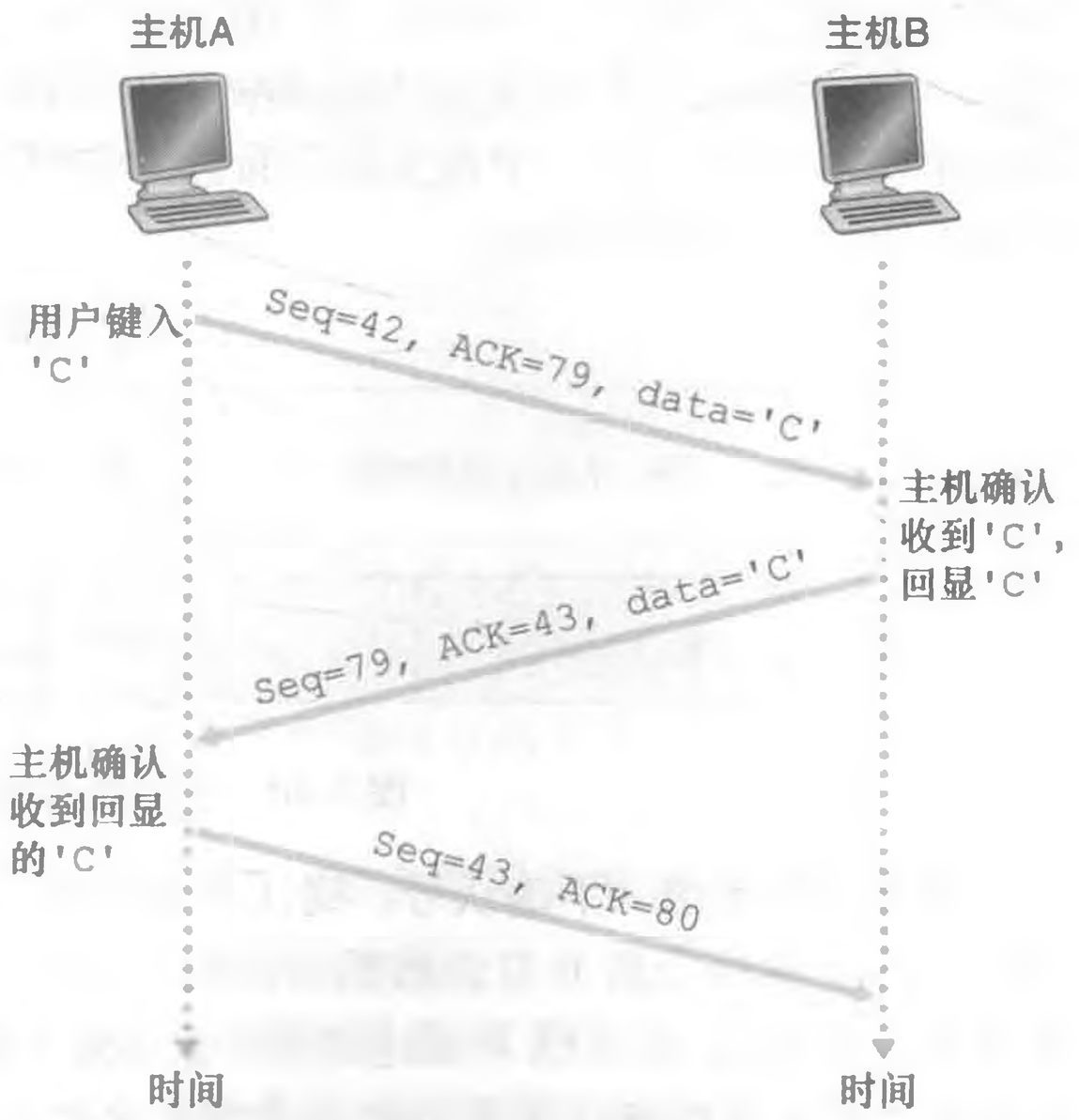

图 3-31　一个经 TCP 的简单 Telnet 应用的确认号和序号

所示，假设客户和服务器的起始序号分别是42和79。前面讲过，一个报文段的序号就是该报文段数据字段首字节的序号。因此，客户发送的第一个报文段的序号为42，服务器发送的第一个报文段的序号为79。前面讲过，确认号就是主机正在等待的数据的下一个字节序号。在TCP连接建立后但没有发送任何数据之前，该客户等待字节79，而该服务器等待字节42。

如图3-31中所示，共发送3个报文段。第一个报文段是由客户发往服务器，在它的数据字段里包含一字节的字符'C'的ASCII码。如我们刚讲到的那样，第一个报文段的序号字段里是42。另外，由于客户还没有接收到来自服务器的任何数据，因此该第一个报文段中的确认号字段中是79。

第二个报文段是由服务器发往客户。它有两个目的：首先它是为该服务器所收到数据提供一个确认。通过在确认号字段中填入43，服务器告诉客户它已经成功地收到字节42及以前的所有字节，现在正等待着字节43的出现。该报文段的第二个目的是回显字符'C'。因此，在第二个报文段的数据字段里填入的是字符'C'的ASCII码。第二个报文段的序号为79，它是该TCP连接上从服务器到客户的数据流的起始序号，这也正是服务器要发送的第一个字节的数据。值得注意的是，对客户到服务器的数据的确认被装载在一个承载服务器到客户的数据的报文段中；这种确认被称为是被**捎带**（piggybacked）在服务器到客户的数据报文段中的。

第三个报文段是从客户发往服务器的。它的唯一目的是确认已从服务器收到的数据。（前面讲过，第二个报文段中包含的数据是字符'C'，是从服务器到客户的。）该报文段的数据字段为空（即确认信息没有被任何从客户到服务器的数据所捎带）。该报文段的确认号字段填入的是80，因为客户已经收到了字节流中序号为79及以前的字节，它现在正等待着字节80的出现。你可能认为这有点奇怪，即使该报文段里没有数据还仍有序号。这是因为TCP存在序号字段，报文段需要填入某个序号。

3.5.3 往返时间的估计与超时

TCP如同前面3.4节所讲的rdt协议一样，它采用超时/重传机制来处理报文段的丢失问题。尽管这在概念上简单，但是当在如TCP这样的实际协议中实现超时/重传机制时还是会产生许多微妙的问题。也许最明显的一个问题就是超时间隔长度的设置。显然，超时间隔必须大于该连接的往返时间（RTT），即从一个报文段发出到它被确认的时间。否则会造成不必要的重传。但是这个时间间隔到底应该是多大呢？刚开始时应如何估计往返时间呢？是否应该为所有未确认的报文段各设一个定时器？问题竟然如此之多！我们在本节中的讨论基于[Jacobson 1988]中有关TCP的工作以及IETF关于管理TCP定时器的建议[RFC 6298]。

1. 估计往返时间

我们开始学习TCP定时器的管理问题，要考虑一下TCP是如何估计发送方与接收方之间往返时间的。这是通过如下方法完成的。报文段的样本RTT（表示为SampleRTT）就是从某报文段被发出（即交给IP）到对该报文段的确认被收到之间的时间量。大多数TCP的实现仅在某个时刻做一次SampleRTT测量，而不是为每个发送的报文段测量一个SampleRTT。这就是说，在任意时刻，仅为一个已发送的但目前尚未被确认的报文段估计SampleRTT，从而产生一个接近每个RTT的新SampleRTT值。另外，TCP决不为已被重传的报文段计算SampleRTT；它仅为传输一次的报文段测量SampleRTT[Kan 1987]。（本章后面

的一个习题请你考虑一下为什么要这么做。）

显然，由于路由器的拥塞和端系统负载的变化，这些报文段的 SampleRTT 值会随之波动。由于这种波动，任何给定的 SampleRTT 值也许都是非典型的。因此，为了估计一个典型的 RTT，自然要采取某种对 SampleRTT 取平均的办法。TCP 维持一个 SampleRTT 均值（称为 EstimatedRTT）。一旦获得一个新 SampleRTT 时，TCP 就会根据下列公式来更新 EstimatedRTT：

$$\text{EstimatedRTT} = (1-\alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

上面的公式是以编程语言的语句方式给出的，即 EstimatedRTT 的新值是由以前的 EstimatedRTT 值与 SampleRTT 新值加权组合而成的。在［RFC 6298］中给出的 α 推荐值是 $\alpha = 0.125$（即 1/8），这时上面的公式变为：

$$\text{EstimatedRTT} = 0.875 \cdot \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT}$$

值得注意的是，EstimatedRTT 是一个 SampleRTT 值的加权平均值。如在本章后面习题中讨论的那样，这个加权平均对最近的样本赋予的权值要大于对旧样本赋予的权值。这是很自然的，因为越近的样本越能更好地反映网络的当前拥塞情况。从统计学观点讲，这种平均被称为**指数加权移动平均**（Exponential Weighted Moving Average，EWMA）。在 EWMA 中的“指数”一词看起来是指一个给定的 SampleRTT 的权值在更新的过程中呈指数型快速衰减。在课后习题中，将要求你推导出 EstimatedRTT 的指数表达形式。

图 3-32 显示了当 $\alpha = 1/8$ 时，在 gaia. cs. umass. edu（在美国马萨诸塞州的 Amherst）与 fantasia. eurecom. fr（在法国南部）之间的一条 TCP 连接上的 SampleRTT 值与 EstimatedRTT 值。显然，SampleRTT 的变化在 EstimatedRTT 的计算中趋于平缓了。

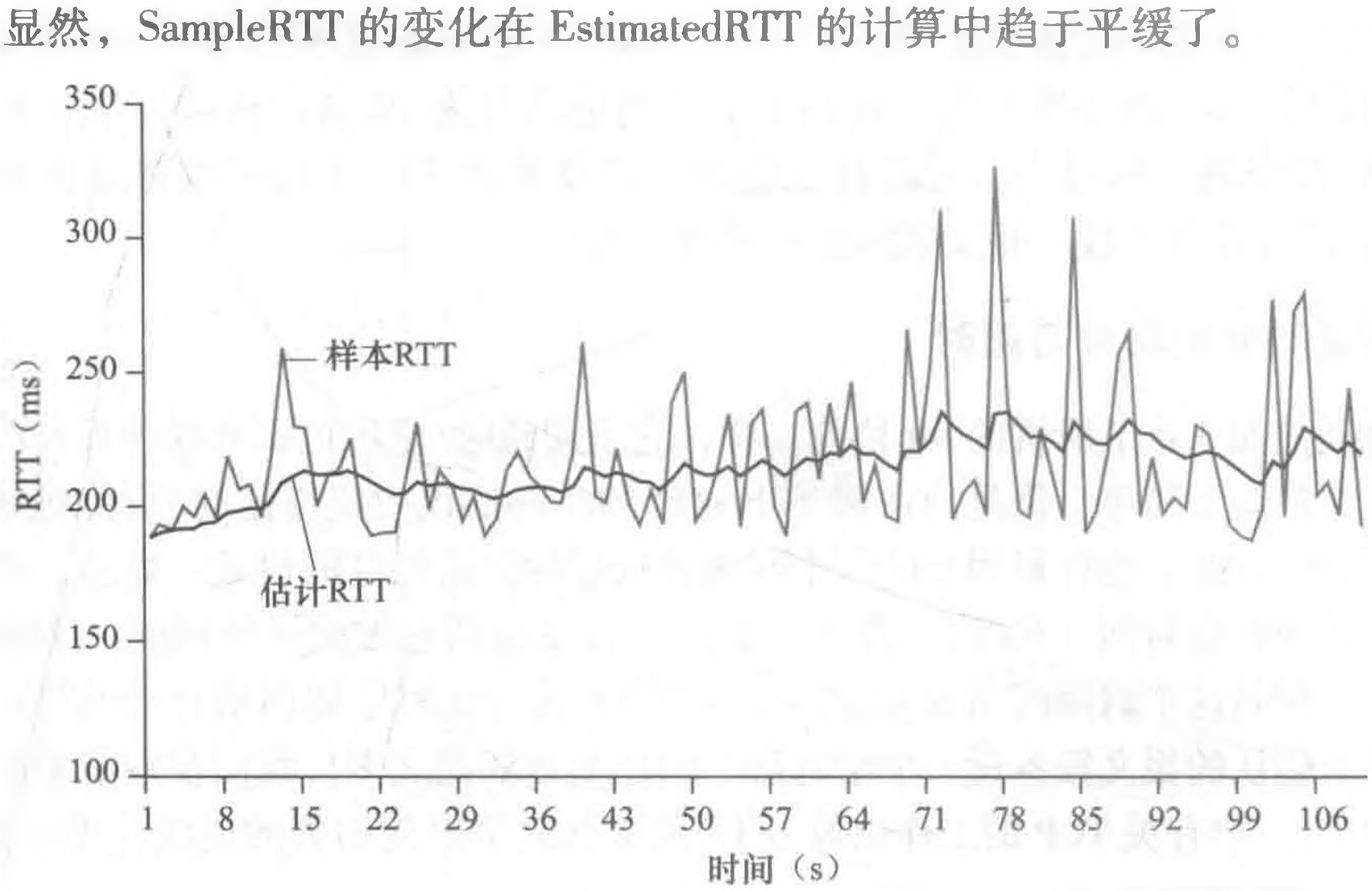

图 3-32　RTT 样本和 RTT 估计

除了估算 RTT 外，测量 RTT 的变化也是有价值的。［RFC 6298］定义了 RTT 偏差 DevRTT，用于估算 SampleRTT 一般会偏离 EstimatedRTT 的程度：

$$\text{DevRTT} = (1-\beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

注意到 DevRTT 是一个 SampleRTT 与 EstimatedRTT 之间差值的 EWMA。如果 SampleRTT 值波动较小，那么 DevRTT 的值就会很小；另一方面，如果波动很大，那么 DevRTT 的值就会很大。β 的推荐值为 0.25。

2. 设置和管理重传超时间隔

假设已经给出了 EstimatedRTT 值和 DevRTT 值，那么 TCP 超时间隔应该用什么值呢？很明显，超时间隔应该大于等于 EstimatedRTT，否则，将造成不必要的重传。但是超时间隔也不应该比 EstimatedRTT 大太多，否则当报文段丢失时，TCP 不能很快地重传该报文段，导致数据传输时延大。因此要求将超时间隔设为 EstimatedRTT 加上一定余量。当 SampleRTT 值波动较大时，这个余量应该大些；当波动较小时，这个余量应该小些。因此，DevRTT 值应该在这里发挥作用了。在 TCP 的确定重传超时间隔的方法中，所有这些因素都考虑到了：

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

推荐的初始 TimeoutInterval 值为 1 秒［RFC 6298］。同时，当出现超时后，TimeoutInterval 值将加倍，以免即将被确认的后继报文段过早出现超时。然而，只要收到报文段并更新 EstimatedRTT，就使用上述公式再次计算 TimeoutInterval。

实 践 原 则

与我们在 3.4 节中所学的方法很像，TCP 通过使用肯定确认与定时器来提供可靠数据传输。TCP 确认正确接收到的数据，而当认为报文段或其确认报文丢失或受损时，TCP 会重传这些报文段。有些版本的 TCP 还有一个隐式 NAK 机制（在 TCP 的快速重传机制下，收到对一个特定报文段的 3 个冗余 ACK 就可作为对后面报文段的一个隐式 NAK，从而在超时之前触发对该报文段的重传）。TCP 使用序号以使接收方能识别丢失或重复的报文段。像可靠数据传输协议 rdt3.0 的情况一样，TCP 自己也无法明确地分辨一个报文段或其 ACK 是丢失了还是受损了，或是时延过长了。在发送方，TCP 的响应是相同的：重传有疑问的报文段。

TCP 也使用流水线，使得发送方在任意时刻都可以有多个已发出但还未被确认的报文段存在。我们在前面已经看到，当报文段长度与往返时延之比很小时，流水线可显著地增加一个会话的吞吐量。一个发送方能够具有的未被确认报文段的具体数量是由 TCP 的流量控制和拥塞控制机制决定的。TCP 流量控制将在本节后面讨论；TCP 拥塞控制将在 3.7 节中讨论。此时我们只需知道 TCP 发送方使用了流水线。

3.5.4 可靠数据传输

前面讲过，因特网的网络层服务（IP 服务）是不可靠的。IP 不保证数据报的交付，不保证数据报的按序交付，也不保证数据报中数据的完整性。对于 IP 服务，数据报能够溢出路由器缓存而永远不能到达目的地，数据报也可能是乱序到达，而且数据报中的比特可能损坏（由 0 变为 1 或者相反）。由于运输层报文段是被 IP 数据报携带着在网络中传输的，所以运输层的报文段也会遇到这些问题。

TCP 在 IP 不可靠的尽力而为服务之上创建了一种**可靠数据传输服务**（reliable data transfer service）。TCP 的可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流；即该字节流与连接的另一方端系统发送出的字节流是完全相同。TCP 提供可靠数据传输的方法涉及我们在 3.4 节中所学的许多原理。

在我们前面研发可靠数据传输技术时，曾假定每一个已发送但未被确认的报文段都与一

个定时器相关联，这在概念上是最简单的。虽然这在理论上很好，但定时器的管理却需要相当大的开销。因此，推荐的定时器管理过程［RFC 6298］仅使用单一的重传定时器，即使有多个已发送但还未被确认的报文段。在本节中描述的 TCP 协议遵循了这种单一定时器的推荐。

我们将以两个递增的步骤来讨论 TCP 是如何提供可靠数据传输的。我们先给出一个 TCP 发送方的高度简化的描述，该发送方只用超时来恢复报文段的丢失；然后再给出一个更全面的描述，该描述中除了使用超时机制外，还使用冗余确认技术。在接下来的讨论中，我们假定数据仅向一个方向发送，即从主机 A 到主机 B，且主机 A 在发送一个大文件。

图 3-33 给出了一个 TCP 发送方高度简化的描述。我们看到在 TCP 发送方有 3 个与发送和重传有关的主要事件：从上层应用程序接收数据；定时器超时和收到 ACK。一旦第一个主要事件发生，TCP 从应用程序接收数据，将数据封装在一个报文段中，并把该报文段交给 IP。注意到每一个报文段都包含一个序号，如 3.5.2 节所讲的那样，这个序号就是该报文段第一个数据字节的字节流编号。还要注意到如果定时器还没有为某些其他报文段而运行，则当报文段被传给 IP 时，TCP 就启动该定时器。（将定时器想象为与最早的未被确认的报文段相关联是有帮助的。）该定时器的过期间隔是 TimeoutInterval，它是由 3.5.3 节中所描述的 EstimatedRTT 和 DevRTT 计算得出的。

```
/* 假设发送方不受TCP流量和拥塞控制的限制，来自上层数据的长度小于MSS，且数据传送只在一个
方向进行。*/

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop   (永远)   {
     switch (事件)

          事件：从上面应用程序接收到数据e
               生成具有序号NextSeqNum的TCP报文段
               if (定时器当前没有运行)
                    启动定时器
               向IP传递报文段
               NextSeqNum=NextSeqNum+length(data)
               break;

          事件：定时器超时
               重传具有最小序号但仍未应答的报文段
               启动定时器
               break;

          事件：收到ACK，具有ACK字段值y
               if (y > SendBase) {
                    SendBase=y
                    if （当前仍无任何应答报文段）
                         启动定时器
                    }
               break;

     } /* 结束永远循环 */
```

图 3-33　简化的 TCP 发送方

第二个主要事件是超时。TCP 通过重传引起超时的报文段来响应超时事件。然后 TCP 重启定时器。

TCP 发送方必须处理的第三个主要事件是，到达一个来自接收方的确认报文段（ACK）（更确切地说，是一个包含了有效 ACK 字段值的报文段）。当该事件发生时，TCP 将 ACK 的值 y 与它的变量 SendBase 进行比较。TCP 状态变量 SendBase 是最早未被确认的字节的序号。（因此 SendBase - 1 是指接收方已正确按序接收到的数据的最后一个字节的序号。）如前面指出的那样，TCP 采用累积确认，所以 y 确认了字节编号在 y 之前的所有字节都已经收到。如果 $y >$ SendBase，则该 ACK 是在确认一个或多个先前未被确认的报文段。因此发送方更新它的 SendBase 变量；如果当前有未被确认的报文段，TCP 还要重新启动定时器。

1. 一些有趣的情况

我们刚刚描述了一个关于 TCP 如何提供可靠数据传输的高度简化的版本。但即使这种高度简化的版本，仍然存在着许多微妙之处。为了较好地感受该协议的工作过程，我们来看几种简单情况。图 3-34 描述了第一种情况，主机 A 向主机 B 发送一个报文段。假设该报文段的序号是 92，而且包含 8 字节数据。在发出该报文段之后，主机 A 等待一个来自主机 B 的确认号为 100 的报文段。虽然 A 发出的报文段在主机 B 上被收到，但从主机 B 发往主机 A 的确认报文丢失了。在这种情况下，超时事件就会发生，主机 A 会重传相同的报文段。当然，当主机 B 收到该重传的报文段时，它将通过序号发现该报文段包含了早已收到的数据。因此，主机 B 中的 TCP 将丢弃该重传的报文段中的这些字节。

在第二种情况中，如图 3-35 所示，主机 A 连续发回了两个报文段。第一个报文段序号是 92，包含 8 字节数据；第二个报文段序号是 100，包含 20 字节数据。假设两个报文段都完好无损地到达主机 B，并且主机 B 为每一个报文段分别发送一个确认。第一个确认报文的确认号是 100，第二个确认报文的确认号是 120。现在假设在超时之前这两个报文段中没有一个确认报文到达主机 A。当超时事件发生时，主机 A 重传序号 92 的第一个报文段，并重启定时器。只要第二个报文段的 ACK 在新的超时发生以前到达，则第二个报文段将不会被重传。

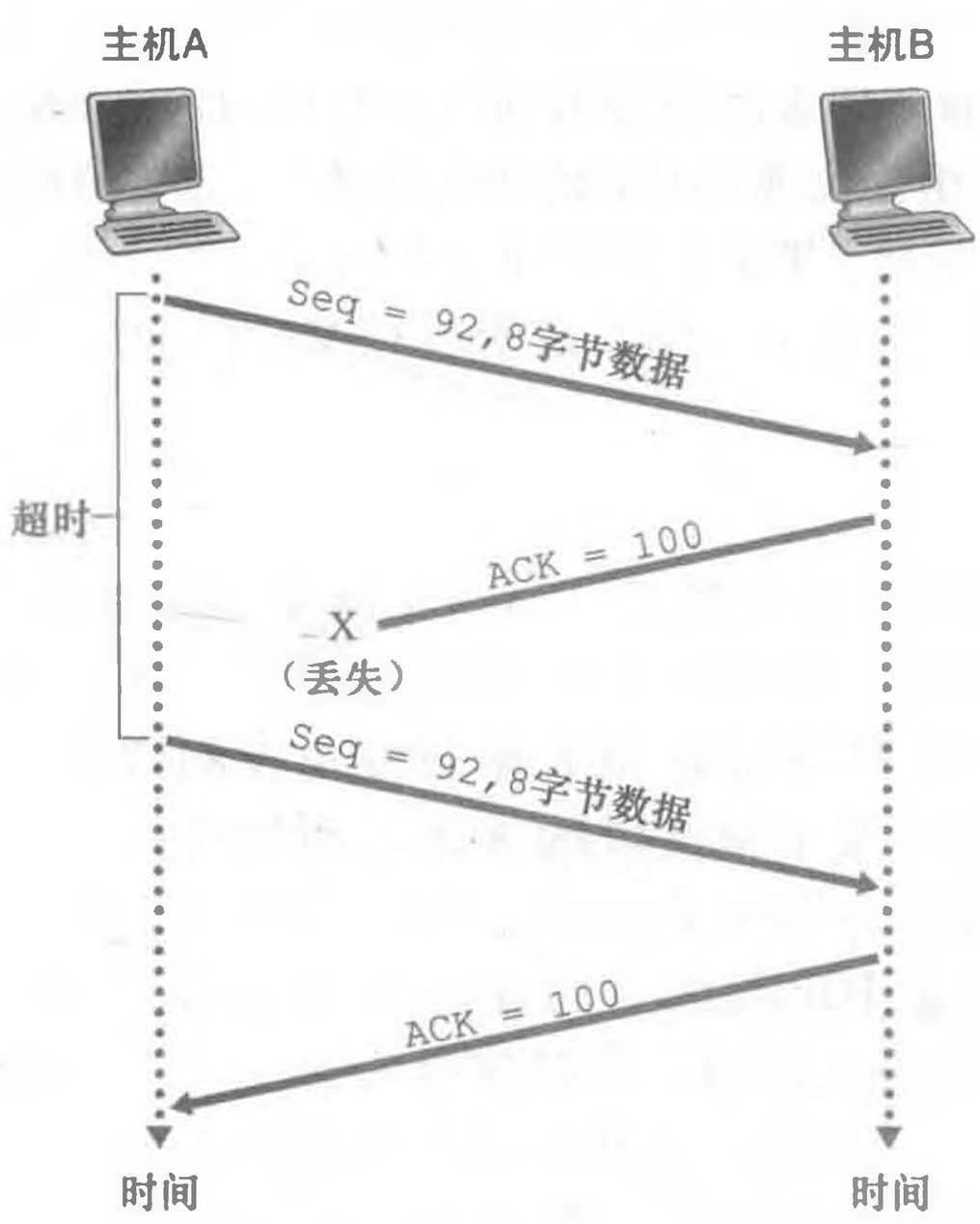

图 3-34 由于确认丢失而重传

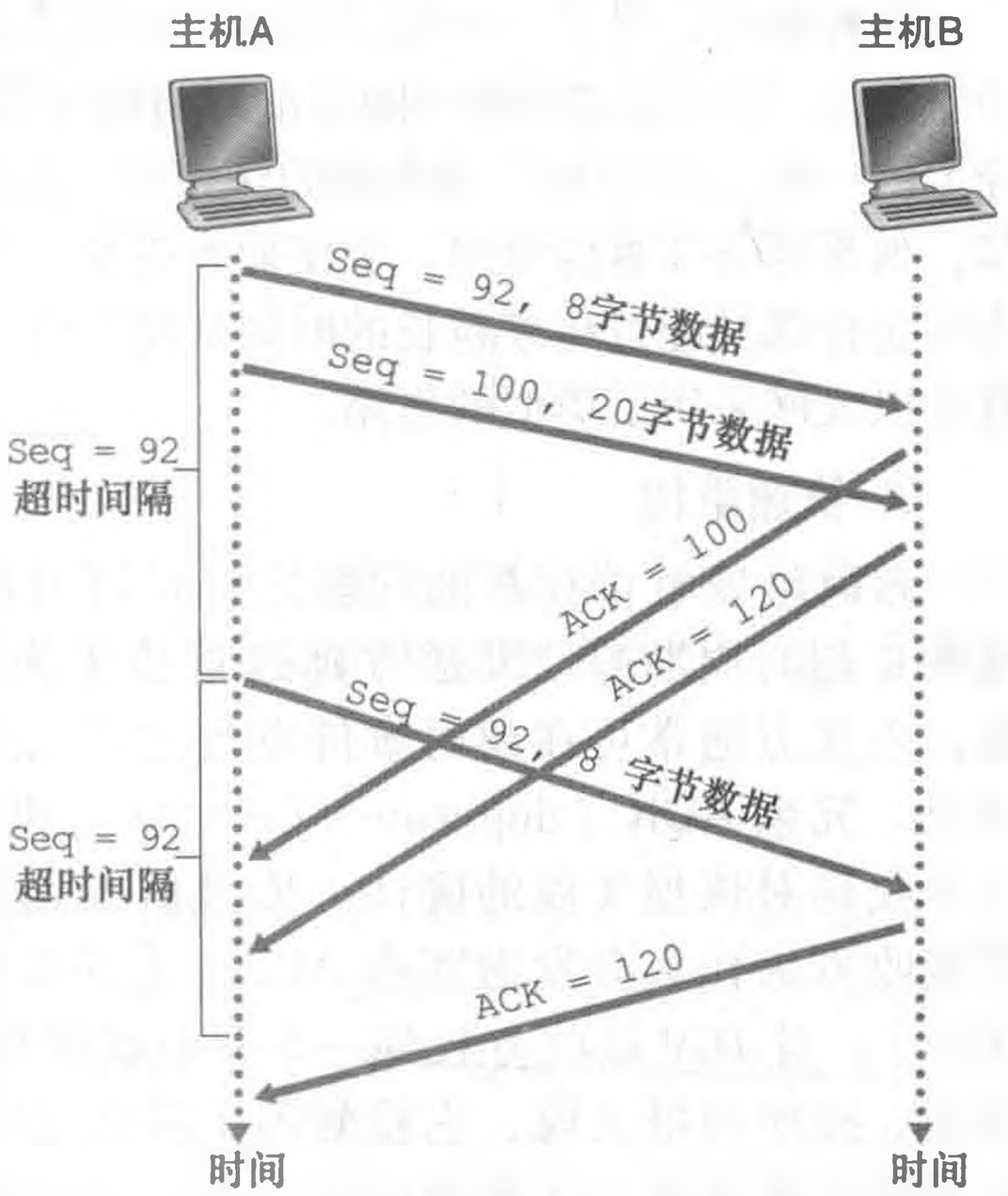

图 3-35 报文段 100 没有重传

在第三种也是最后一种情况中，假设主机 A 与在第二种情况中完全一样，发送两个报

文段。第一个报文段的确认报文在网络丢失，但在超时事件发生之前主机 A 收到一个确认号为 120 的确认报文。主机 A 因而知道主机 B 已经收到了序号为 119 及之前的所有字节；所以主机 A 不会重传这两个报文段中的任何一个。这种情况在图 3-36 中进行了图示。

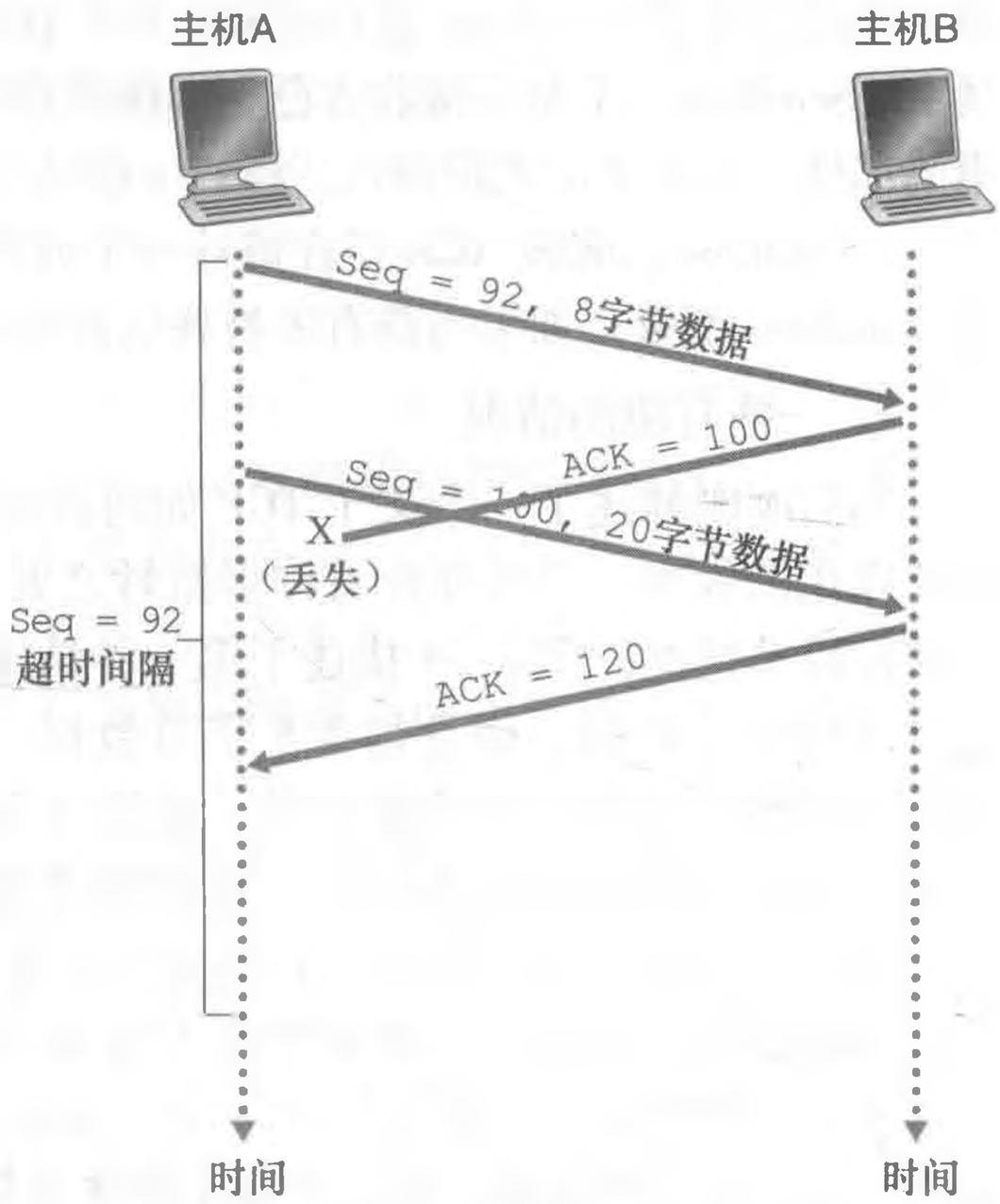

图 3-36　累积确认避免了第一个报文段的重传

2. 超时间隔加倍

我们现在讨论一下在大多数 TCP 实现中所做的一些修改。首先关注的是在定时器时限过期后超时间隔的长度。在这种修改中，每当超时事件发生时，如前所述，TCP 重传具有最小序号的还未被确认的报文段。只是每次 TCP 重传时都会将下一次的超时间隔设为先前值的两倍，而不是用从 EstimatedRTT 和 DevRTT 推算出的值（如在 3.5.3 节中所描述的）。例如，假设当定时器第一次过期时，与最早的未被确认的报文段相关联的 TimeoutInterval 是 0.75 秒。TCP 就会重传该报文段，并把新的过期时间设置为 1.5 秒。如果 1.5 秒后定时器又过期了，则 TCP 将再次重传该报文段，并把过期时间设置为 3.0 秒。因此，超时间隔在每次重传后会呈指数型增长。然而，每当定时器在另两个事件（即收到上层应用的数据和收到 ACK）中的任意一个启动时，TimeoutInterval 由最近的 EstimatedRTT 值与 DevRTT 值推算得到。

这种修改提供了一个形式受限的拥塞控制。（更复杂的 TCP 拥塞控制形式将在 3.7 节中学习。）定时器过期很可能是由网络拥塞引起的，即太多的分组到达源与目的地之间路径上的一台（或多台）路由器的队列中，造成分组丢失或长时间的排队时延。在拥塞的时候，如果源持续重传分组，会使拥塞更加严重。相反，TCP 使用更文雅的方式，每个发送方的重传都是经过越来越长的时间间隔后进行的。当我们在第 6 章学习 CSMA/CD 时，将看到以太网采用了类似的思路。

3. 快速重传

超时触发重传存在的问题之一是超时周期可能相对较长。当一个报文段丢失时，这种长超时周期迫使发送方延迟重传丢失的分组，因而增加了端到端时延。幸运的是，发送方通常可在超时事件发生之前通过注意所谓冗余 ACK 来较好地检测到丢包情况。**冗余 ACK**（duplicate ACK）就是再次确认某个报文段的 ACK，而发送方先前已经收到对该报文段的确认。要理解发送方对冗余 ACK 的响应，我们必须首先看一下接收方为什么会发送冗余 ACK。表 3-2 总结了 TCP 接收方的 ACK 生成策略［RFC 5681］。当 TCP 接收方收到一个具有这样序号的报文段时，即其序号大于下一个所期望的、按序的报文段，它检测到了数据流中的一个间隔，这就是说有报文段丢失。这个间隔可能是由于在网络中报文段丢失或重新排序造成的。因为 TCP 不使用否定确认，所以接收方不能向发送方发回一个显式的否定确认。相反，它只是对已经接收到的最后一个按序字节数据进行重复确认（即产生一个冗余 ACK）即可。（注意到在