表 3-2 中允许接收方不丢弃失序报文段。）

表 3-2 产生 TCP ACK 的建议［RFC 5681］

| 事件 | TCP 接收方动作 |
| --- | --- |
| 具有所期望序号的按序报文段到达。所有在期望序号及以前的数据都已经被确认 | 延迟的 ACK。对另一个按序报文段的到达最多等待 500ms。如果下一个按序报文段在这个时间间隔内没有到达，则发送一个 ACK |
| 具有所期望序号的按序报文段到达。另一个按序报文段等待 ACK 传输 | 立即发送单个累积 ACK，以确认两个按序报文段 |
| 比期望序号大的失序报文段到达。检测出间隔 | 立即发送冗余 ACK，指示下一个期待字节的序号（其为间隔的低端的序号） |
| 能部分或完全填充接收数据间隔的报文段到达 | 倘若该报文段起始于间隔的低端，则立即发送 ACK |

因为发送方经常一个接一个地发送大量的报文段，如果一个报文段丢失，就很可能引起许多一个接一个的冗余 ACK。如果 TCP 发送方接收到对相同数据的 3 个冗余 ACK，它把这当作一种指示，说明跟在这个已被确认过 3 次的报文段之后的报文段已经丢失。（在课后习题中，我们将考虑为什么发送方等待 3 个冗余 ACK，而不是仅仅等待一个冗余 ACK。）一旦收到 3 个冗余 ACK，TCP 就执行**快速重传**（fast retransmit）［RFC 5681］，即在该报文段的定时器过期之前重传丢失的报文段。对于采用快速重传的 TCP，可用下列代码片段代替图 3-33 中的 ACK 收到事件：

```
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
    }
else {/*对已经确认的报文段的一个冗余ACK*/
    对y收到的冗余ACK数加1
    if (对y==3收到的冗余ACK数)
        /* TCP快速重传 */
        重新发送具有序号y的报文段
    }
break;
```

前面讲过，当在如 TCP 这样一个实际协议中实现超时/重传机制时，会产生许多微妙的问题。上面的过程是在超过 20 年的 TCP 定时器使用经验的基础上演化而来的，读者应当理解实际情况确实是这样的。

#### 4. 是回退 *N* 步还是选择重传

考虑下面这个问题来结束有关 TCP 差错恢复机制的学习：TCP 是一个 GBN 协议还是一个 SR 协议？前面讲过，TCP 确认是累积式的，正确接收但失序的报文段是不会被接收方逐个确认的。因此，如图 3-33 所示（也可参见图 3-19），TCP 发送方仅需维持已发送过但未被确认的字节的最小序号（SendBase）和下一个要发送的字节的序号（NextSeqNum）。在这种意义下，TCP 看起来更像一个 GBN 风格的协议。但是 TCP 和 GBN 协议之间有着一些显著的区别。许多 TCP 实现会将正确接收但失序的报文段缓存起来［Stevens 1994］。另外考虑一下，当发送方发送的一组报文段 1，2，…，*N*，并且所有的

报文段都按序无差错地到达接收方时会发生的情况。进一步假设对分组 $n < N$ 的确认报文丢失，但是其余 $N-1$ 个确认报文在分别超时以前到达发送端，这时又会发生的情况。在该例中，GBN 不仅会重传分组 $n$，还会重传所有后继的分组 $n+1$，$n+2$，…，$N$。在另一方面，TCP 将重传至多一个报文段，即报文段 $n$。此外，如果对报文段 $n+1$ 的确认报文在报文段 $n$ 超时之前到达，TCP 甚至不会重传报文段 $n$。

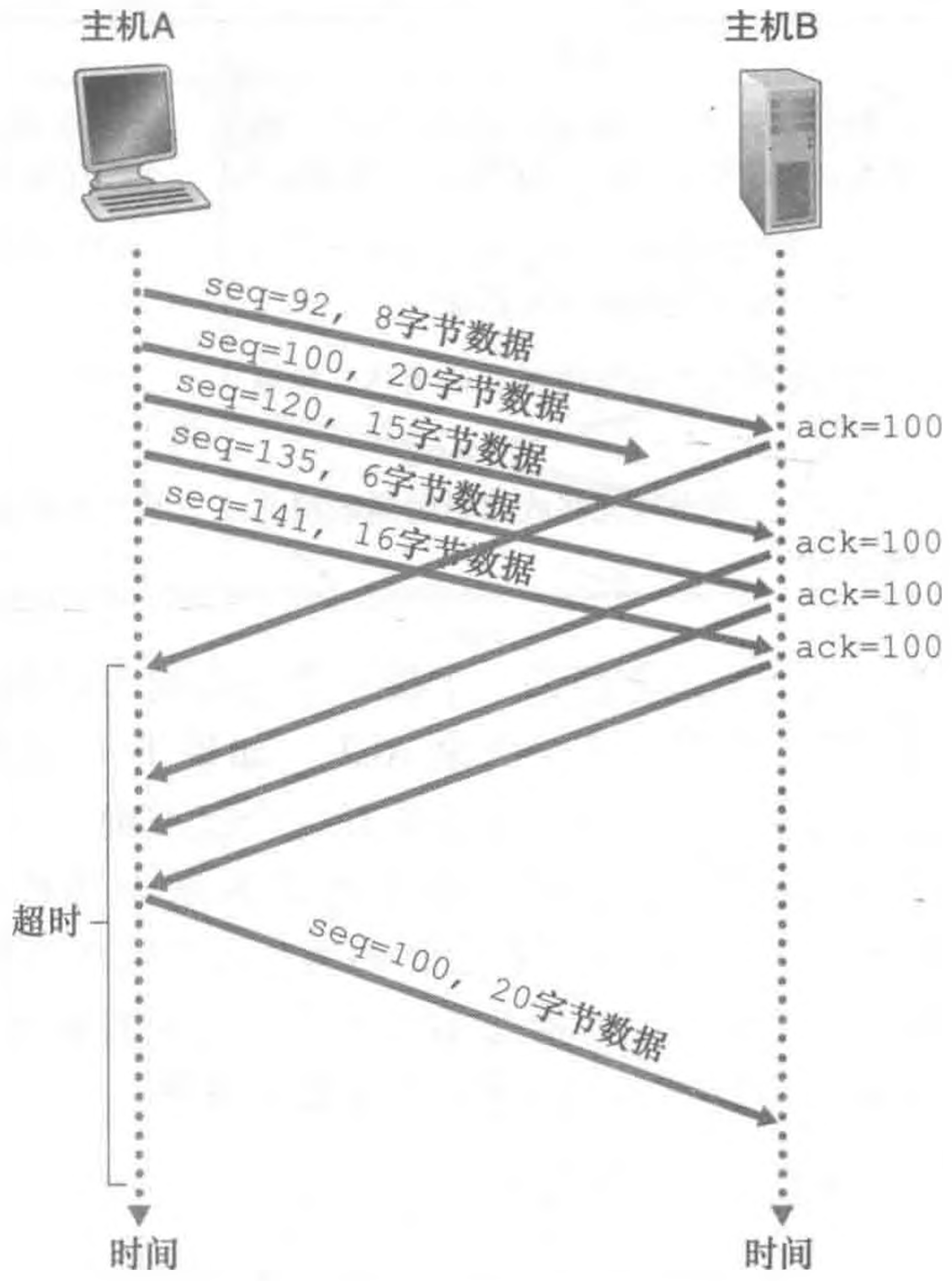


图 3-37　快速重传：在某报文段的定时器过期之前重传丢失的报文段

对 TCP 提出的一种修改意见是所谓的**选择确认**（selective acknowledgment）[RFC 2018]，它允许 TCP 接收方有选择地确认失序报文段，而不是累积地确认最后一个正确接收的有序报文段。当将该机制与选择重传机制结合起来使用时（即跳过重传那些已被接收方选择性地确认过的报文段），TCP 看起来就很像我们通常的 SR 协议。因此，TCP 的差错恢复机制也许最好被分类为 GBN 协议与 SR 协议的混合体。

### 3.5.5　流量控制

前面讲过，一条 TCP 连接的每一侧主机都为该连接设置了接收缓存。当该 TCP 连接收到正确、按序的字节后，它就将数据放入接收缓存。相关联的应用进程会从该缓存中读取数据，但不必是数据刚一到达就立即读取。事实上，接收方应用也许正忙于其他任务，甚至要过很长时间后才去读取该数据。如果某应用程序读取数据时相对缓慢，而发送方发送得太多、太快，发送的数据就会很容易地使该连接的接收缓存溢出。

TCP 为它的应用程序提供了**流量控制服务**（flow-control service）以消除发送方使接收方缓存溢出的可能性。流量控制因此是一个速度匹配服务，即发送方的发送速率与接收方应用程序的读取速率相匹配。前面提到过，TCP 发送方也可能因为 IP 网络的拥塞而被遏制；这种形式的发送方的控制被称为**拥塞控制**（congestion control），我们将在 3.6 节和 3.7 节详细地讨论这个主题。即使流量控制和拥塞控制采取的动作非常相似（对发送方的遏制），但是它们显然是针对完全不同的原因而采取的措施。不幸的是，许多作者把这两个术语混用，理解力强的读者会明智地区分这两种情况。现在我们来讨论 TCP 如何提供流量控制服务的。为了能从整体上看问题，我们在本节都假设 TCP 是这样实现的，即 TCP 接收方丢弃失序的报文段。

TCP 通过让发送方维护一个称为**接收窗口**（receive window）的变量来提供流量控制。通俗地说，接收窗口用于给发送方一个指示——该接收方还有多少可用的缓存空间。因为 TCP 是全双工通信，在连接两端的发送方都各自维护一个接收窗口。我们在文件传输的情况下研究接收窗口。假设主机 A 通过一条 TCP 连接向主机 B 发送一个大文件。主机 B 为该连接分配了一个接收缓存，并用 RcvBuffer 来表示其大小。主机 B 上的应用进程不时地

从该缓存中读取数据。我们定义以下变量：

- LastByteRead：主机 B 上的应用进程从缓存读出的数据流的最后一个字节的编号。
- LastByteRcvd：从网络中到达的并且已放入主机 B 接收缓存中的数据流的最后一个字节的编号。

由于 TCP 不允许已分配的缓存溢出，下式必须成立：

$$\text{LastByteRcvd} - \text{LastByteRead} \leqslant \text{RcvBuffer}$$

接收窗口用 rwnd 表示，根据缓存可用空间的数量来设置：

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

由于该空间是随着时间变化的，所以 rwnd 是动态的。图 3-38 对变量 rwnd 进行了图示。

连接是如何使用变量 rwnd 来提供流量控制服务的呢？主机 B 通过把当前的 rwnd 值放入它发给主机 A 的报文段接收窗口字段中，通知主机 A 它在该连接的缓存中还有多少可用空间。开始时，主机 B 设定 rwnd = RcvBuffer。注意到为了实现这一点，主机 B 必须跟踪几个与连接有关的变量。


图 3-38 接收窗口（rwnd）和接收缓存（RcvBuffer）

主机 A 轮流跟踪两个变量，LastByteSent 和 LastByteAcked，这两个变量的意义很明显。注意到这两个变量之间的差 LastByteSent - LastByteAcked，就是主机 A 发送到连接中但未被确认的数据量。通过将未确认的数据量控制在值 rwnd 以内，就可以保证主机 A 不会使主机 B 的接收缓存溢出。因此，主机 A 在该连接的整个生命周期须保证：

$$\text{LastByteSent} - \text{LastByteAcked} \leqslant \text{rwnd}$$

对于这个方案还存在一个小小的技术问题。为了理解这一点，假设主机 B 的接收缓存已经存满，使得 rwnd = 0。在将 rwnd = 0 通告给主机 A 之后，还要假设主机 B 没有任何数据要发给主机 A。此时，考虑会发生什么情况。因为主机 B 上的应用进程将缓存清空，TCP 并不向主机 A 发送带有 rwnd 新值的新报文段；事实上，TCP 仅当在它有数据或有确认要发时才会发送报文段给主机 A。这样，主机 A 不可能知道主机 B 的接收缓存已经有新的空间了，即主机 A 被阻塞而不能再发送数据！为了解决这个问题，TCP 规范中要求：当主机 B 的接收窗口为 0 时，主机 A 继续发送只有一个字节数据的报文段。这些报文段将会被接收方确认。最终缓存将开始清空，并且确认报文里将包含一个非 0 的 rwnd 值。

位于 http://www.awl.com/kurose-ross 的在线站点为本书提供了一个交互式 Java 小程序，用以说明 TCP 接收窗口的运行情况。

描述了 TCP 的流量控制服务以后，我们在此要简要地提一下 UDP 并不提供流量控制，报文段由于缓存溢出可能在接收方丢失。例如，考虑一下从主机 A 上的一个进程向主机 B 上的一个进程发送一系列 UDP 报文段的情形。对于一个典型的 UDP 实现，UDP 将在一个有限大小的缓存中加上报文段，该缓存在相应套接字（进程的门户）“之前”。进程每次从缓存中读取一个完整的报文段。如果进程从缓存中读取报文段的速度不够快，那么缓存将会溢出，并且将丢失报文段。

### 3.5.6　TCP 连接管理

在本小节中，我们更为仔细地观察如何建立和拆除一条 TCP 连接。尽管这个主题并不特别令人兴奋，但是它很重要，因为 TCP 连接的建立会显著地增加人们感受到的时延（如在 Web 上冲浪时）。此外，许多常见的网络攻击（包括极为流行的 SYN 洪泛攻击）利用了 TCP 连接管理中的弱点。现在我们观察一下一条 TCP 连接是如何建立的。假设运行在一台主机（客户）上的一个进程想与另一台主机（服务器）上的一个进程建立一条连接。客户应用进程首先通知客户 TCP，它想建立一个与服务器上某个进程之间的连接。客户中的 TCP 会用以下方式与服务器中的 TCP 建立一条 TCP 连接：

- 第一步：客户端的 TCP 首先向服务器端的 TCP 发送一个特殊的 TCP 报文段。该报文段中不包含应用层数据。但是在报文段的首部（参见图 3-29）中的一个标志位（即 SYN 比特）被置为 1。因此，这个特殊报文段被称为 SYN 报文段。另外，客户会随机地选择一个初始序号（client_isn），并将此编号放置于该起始的 TCP SYN 报文段的序号字段中。该报文段会被封装在一个 IP 数据报中，并发送给服务器。为了避免某些安全性攻击，在适当地随机化选择 client_isn 方面有着不少有趣的研究［CERT 2001-09］。
- 第二步：一旦包含 TCP SYN 报文段的 IP 数据报到达服务器主机（假定它的确到达了!），服务器会从该数据报中提取出 TCP SYN 报文段，为该 TCP 连接分配 TCP 缓存和变量，并向该客户 TCP 发送允许连接的报文段。（我们将在第 8 章看到，在完成三次握手的第三步之前分配这些缓存和变量，使得 TCP 易于受到称为 SYN 洪泛的拒绝服务攻击。）这个允许连接的报文段也不包含应用层数据。但是，在报文段的首部却包含 3 个重要的信息。首先，SYN 比特被置为 1。其次，该 TCP 报文段首部的确认号字段被置为 client_isn + 1。最后，服务器选择自己的初始序号（server_isn），并将其放置到 TCP 报文段首部的序号字段中。这个允许连接的报文段实际上表明了："我收到了你发起建立连接的 SYN 分组，该分组带有初始序号 client_isn。我同意建立该连接。我自己的初始序号是 server_isn。"该允许连接的报文段被称为 **SYNACK 报文段**（SYNACK segment）。
- 第三步：在收到 SYNACK 报文段后，客户也要给该连接分配缓存和变量。客户主机则向服务器发送另外一个报文段；这最后一个报文段对服务器的允许连接的报文段进行了确认（该客户通过将值 server_isn + 1 放置到 TCP 报文段首部的确认字段中来完成此项工作）。因为连接已经建立了，所以该 SYN 比特被置为 0。该三次握手的第三个阶段可以在报文段负载中携带客户到服务器的数据。

一旦完成这 3 个步骤，客户和服务器主机就可以相互发送包括数据的报文段了。在以后每一个报文段中，SYN 比特都将被置为 0。注意到为了创建该连接，在两台主机之间发送了 3 个分组，如图 3-39 所示。由于这个原因，这种连接创建过程通常被称为 **3 次握手**（three-way handshake）。TCP 3 次握手的几个方面将在课后习题中讨论（为什么需要初始序号？为什么需要 3 次握手，而不是两次握手？）。注意到这样一件事是很有趣的，一个攀岩者和一个保护者（他位于攀岩者的下面，他的任务是处理好攀岩者的安全绳索）就使用了与 TCP 相同的 3 次握手通信协议，以确保在攀岩者开始攀爬前双方都已经准备好了。

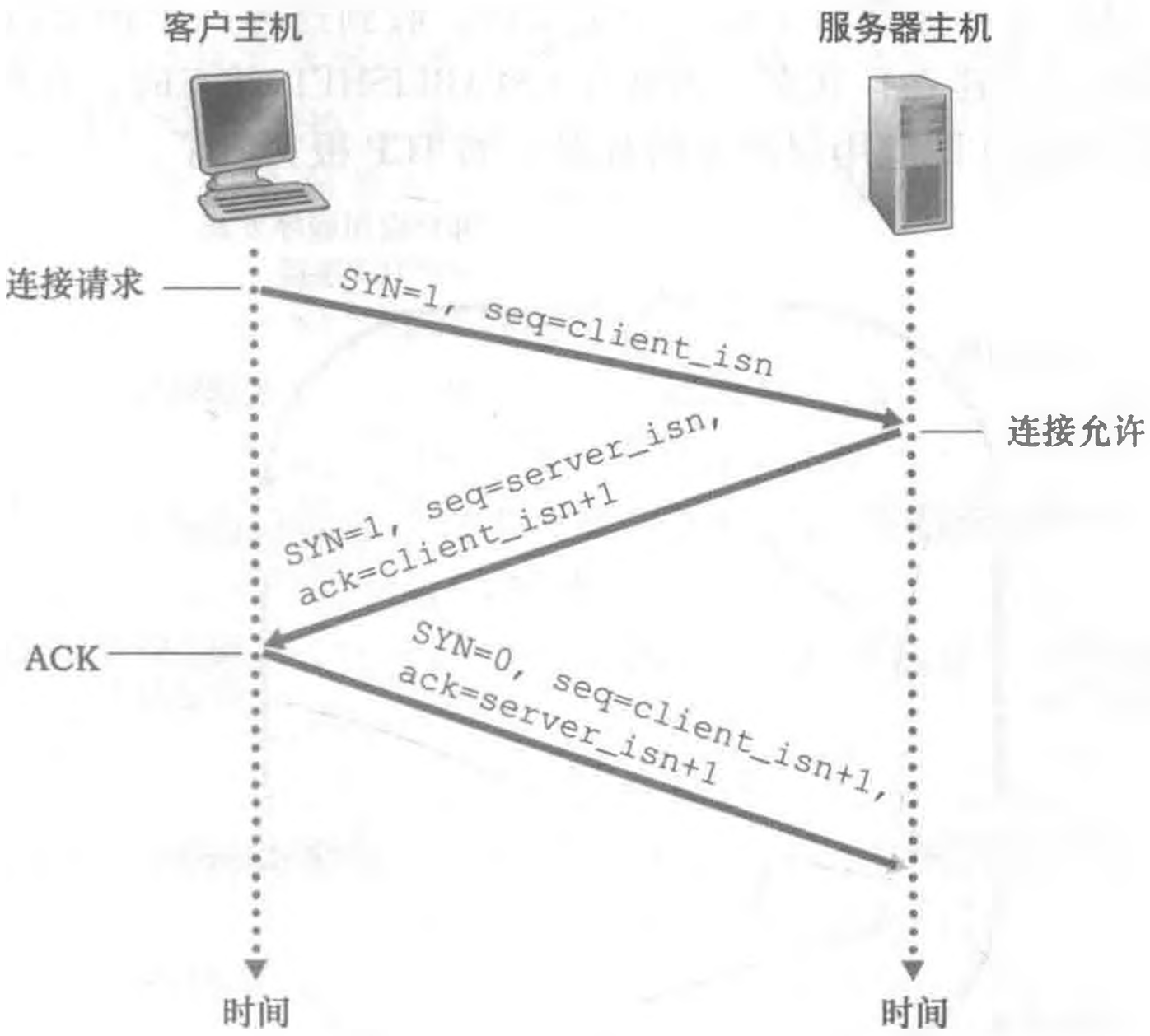


图 3-39 TCP 三次握手：报文段交换

天下没有不散的宴席，对于 TCP 连接也是如此。参与一条 TCP 连接的两个进程中的任何一个都能终止该连接。当连接结束后，主机中的“资源”（即缓存和变量）将被释放。举一个例子，假设某客户打算关闭连接，如图 3-40 所示。客户应用进程发出一个关闭连接命令。这会引起客户 TCP 向服务器进程发送一个特殊的 TCP 报文段。这个特殊的报文段让其首部中的一个标志位即 FIN 比特（参见图 3-29）被设置为 1。当服务器接收到该报文段后，就向发送方回送一个确认报文段。然后，服务器发送它自己的终止报文段，其 FIN 比特被置为 1。最后，该客户对这个服务器的终止报文段进行确认。此时，在两台主机上用于该连接的所有资源都被释放了。

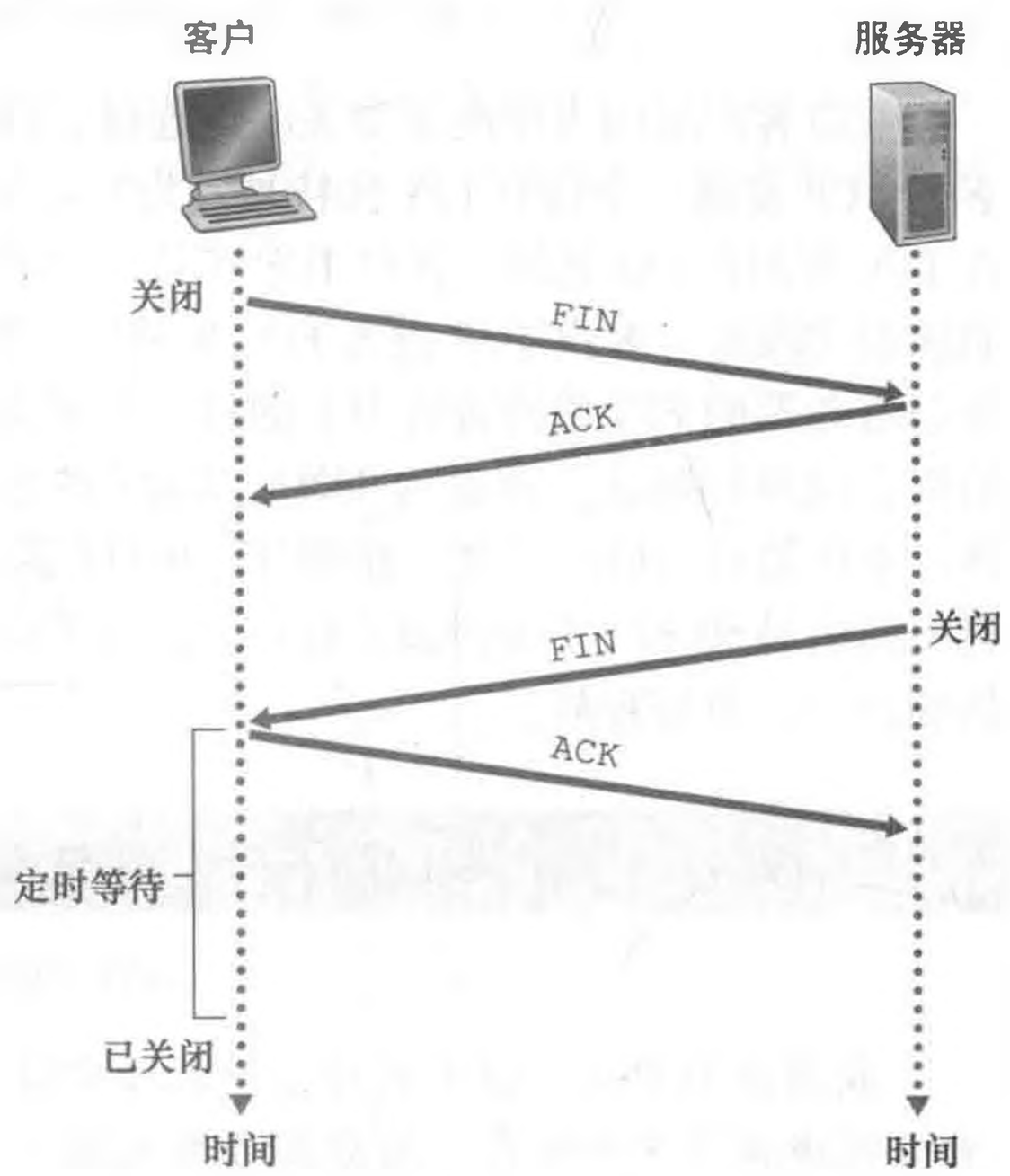


图 3-40 关闭一条 TCP 连接

在一个 TCP 连接的生命周期内，运行在每台主机中的 TCP 协议在各种 TCP 状态（TCP state）之间变迁。图 3-41 说明了客户 TCP 会经历的一系列典型 TCP 状态。

客户 TCP 开始时处于 CLOSED（关闭）状态。客户的应用程序发起一个新的 TCP 连接（可通过在第 2 章讲过的 Python 例子中创建一个 Socket 对象来完成）。这引起客户中的 TCP 向服务器中的 TCP 发送一个 SYN 报文段。在发送过 SYN 报文段后，客户 TCP 进入了 SYN_SENT 状态。当客户 TCP 处在 SYN_SENT 状态时，它等待来自服务器 TCP 的对客户所发报

文段进行确认且 SYN 比特被置为 1 的一个报文段。收到这样一个报文段之后，客户 TCP 进入 ESTABLISHED（已建立）状态。当处在 ESTABLISHED 状态时，TCP 客户就能发送和接收包含有效载荷数据（即应用层产生的数据）的 TCP 报文段了。

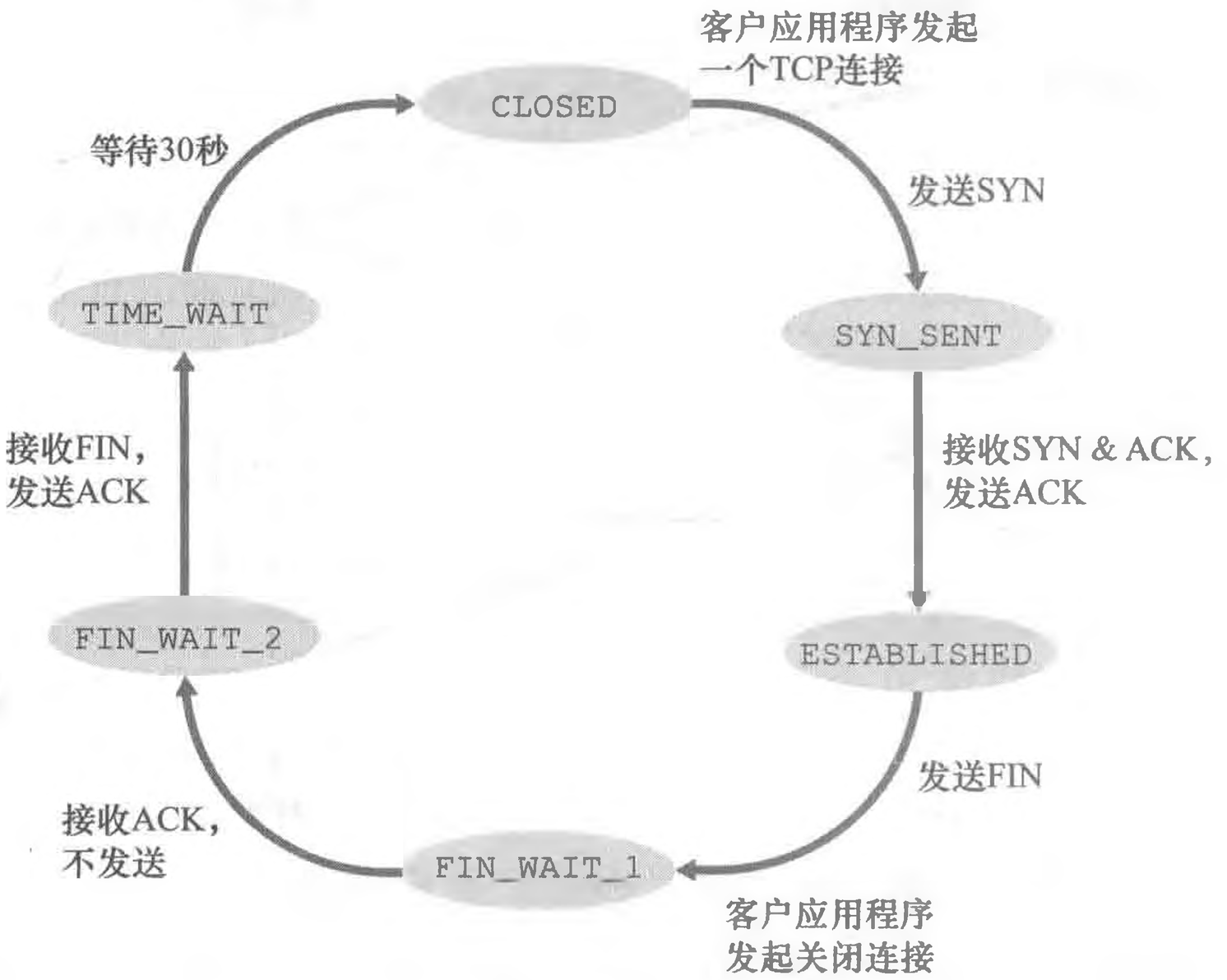


图 3-41　客户 TCP 经历的典型的 TCP 状态序列

假设客户应用程序决定要关闭该连接。（注意到服务器也能选择关闭该连接。）这引起客户 TCP 发送一个带有 FIN 比特被置为 1 的 TCP 报文段，并进入 FIN_WAIT_1 状态。当处在 FIN_WAIT_1 状态时，客户 TCP 等待一个来自服务器的带有确认的 TCP 报文段。当它收到该报文段时，客户 TCP 进入 FIN_WAIT_2 状态。当处在 FIN_WAIT_2 状态时，客户等待来自服务器的 FIN 比特被置为 1 的另一个报文段；当收到该报文段后，客户 TCP 对服务器的报文段进行确认，并进入 TIME_WAIT 状态。假定 ACK 丢失，TIME_WAIT 状态使 TCP 客户重传最后的确认报文。在 TIME_WAIT 状态中所消耗的时间是与具体实现有关的，而典型的值是 30 秒、1 分钟或 2 分钟。经过等待后，连接就正式关闭，客户端所有资源（包括端口号）将被释放。

## 关注安全性

### SYN 洪泛攻击

我们在 TCP 三次握手的讨论中已经看到，服务器为了响应一个收到的 SYN，分配并初始化连接变量和缓存。然后服务器发送一个 SYNACK 进行响应，并等待来自客户的 ACK 报文段。如果某客户不发送 ACK 来完成该三次握手的第三步，最终（通常在一分多钟之后）服务器将终止该半开连接并回收资源。

这种 TCP 连接管理协议为经典的 DoS 攻击即 **SYN 洪泛攻击**（SYN flood attack）提供了环境。在这种攻击中，攻击者发送大量的 TCP SYN 报文段，而不完成第三次握手的步骤。随着这种 SYN 报文段纷至沓来，服务器不断为这些半开连接分配资源（但从未

使用），导致服务器的连接资源被消耗殆尽。这种 SYN 洪泛攻击是被记载的众多 DoS 攻击中的第一种［CERT SYN 1996］。幸运的是，现在有一种有效的防御系统，称为 SYN cookie［RFC 4987］，它们被部署在大多数主流操作系统中。SYN cookie 以下列方式工作：

- 当服务器接收到一个 SYN 报文段时，它并不知道该报文段是来自一个合法的用户，还是一个 SYN 洪泛攻击的一部分。因此服务器不会为该报文段生成一个半开连接。相反，服务器生成一个初始 TCP 序列号，该序列号是 SYN 报文段的源和目的 IP 地址与端口号以及仅有该服务器知道的秘密数的一个复杂函数（散列函数）。这种精心制作的初始序列号被称为“cookie”。服务器则发送具有这种特殊初始序列号的 SYNACK 分组。**重要的是，服务器并不记忆该 cookie 或任何对应于 SYN 的其他状态信息。**
- 如果客户是合法的，则它将返回一个 ACK 报文段。当服务器收到该 ACK，需要验证该 ACK 是与前面发送的某些 SYN 相对应的。如果服务器没有维护有关 SYN 报文段的记忆，这是怎样完成的呢？正如你可能猜测的那样，它是借助于 cookie 来做到的。前面讲过对于一个合法的 ACK，在确认字段中的值等于在 SYNACK 字段（此时为 cookie 值）中的值加 1（参见图 3-39）。服务器则将使用在 SYNACK 报文段中的源和目的地 IP 地址与端口号（它们与初始的 SYN 中的相同）以及秘密数运行相同的散列函数。如果该函数的结果加 1 与在客户的 SYNACK 中的确认（cookie）值相同的话，服务器认为该 ACK 对应于较早的 SYN 报文段，因此它是合法的。服务器则生成一个具有套接字的全开的连接。
- 在另一方面，如果客户没有返回一个 ACK 报文段，则初始的 SYN 并没有对服务器产生危害，因为服务器没有为它分配任何资源。

图 3-42 图示了服务器端的 TCP 通常要经历的一系列状态，其中假设客户开始连接拆

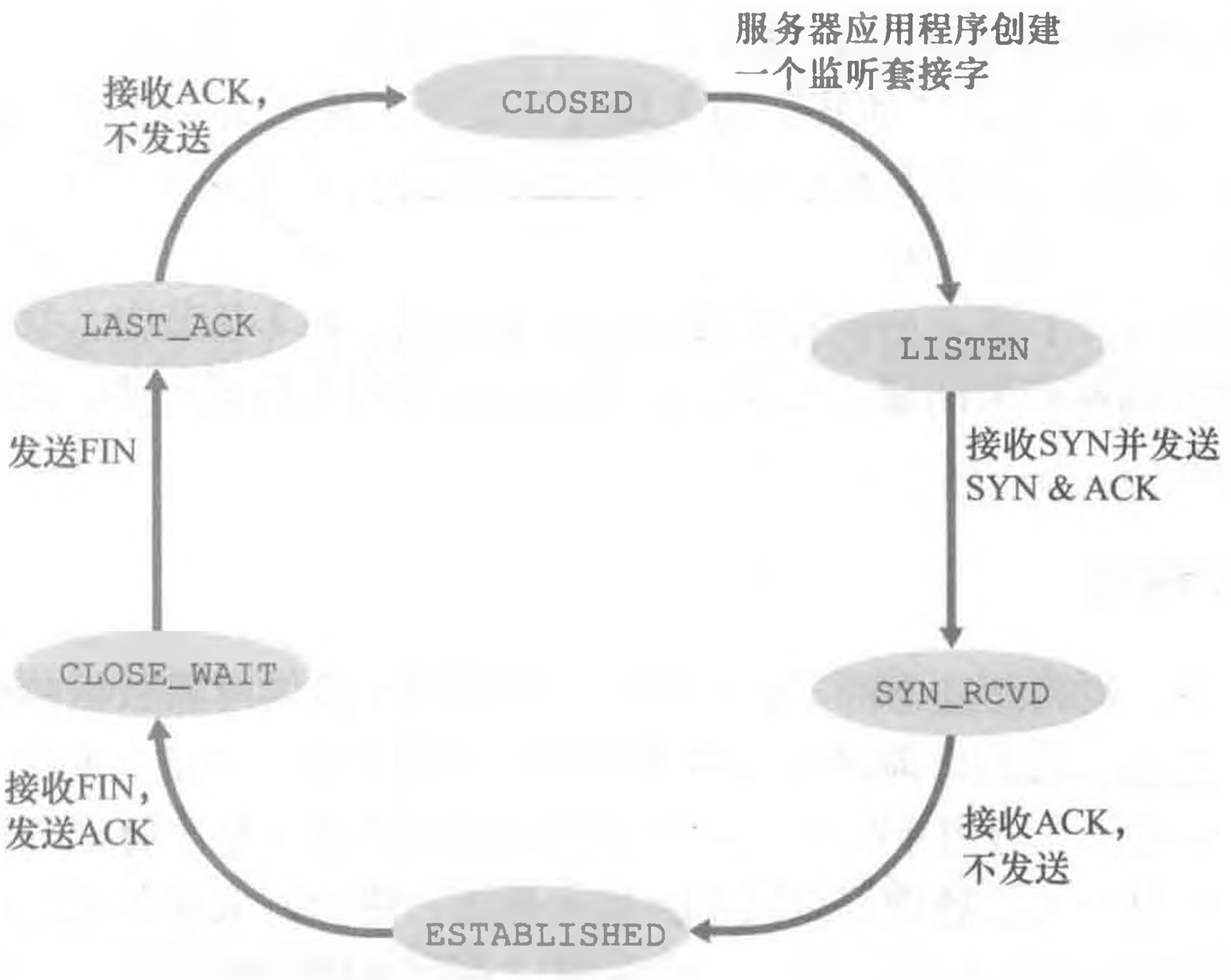


图 3-42 服务器端 TCP 经历的典型的 TCP 状态序列

除。这些状态变迁是自解释的。在这两个状态变迁图中，我们只给出了TCP连接是如何正常地被建立和拆除的。我们没有描述在某些不正常的情况下（例如当连接的双方同时都要发起或终止一条连接时）发生的事情。如果你对此问题及其他与TCP有关的高级问题感兴趣，推荐阅读Stevens的内容更全面的书籍［Stevens 1994］。

我们上面的讨论假定了客户和服务器都准备通信，即服务器正在监听客户发送其SYN报文段的端口。我们来考虑当一台主机接收到一个TCP报文段，其端口号或源IP地址与该主机上进行中的套接字都不匹配的情况。例如，假如一台主机接收了具有目的端口80的一个TCP SYN分组，但该主机在端口80不接受连接（即它不在端口80上运行Web服务器）。则该主机将向源发送一个特殊重置报文段。该TCP报文段将RST标志位（参见3.5.2节）置为1。因此，当主机发送一个重置报文段时，它告诉该源“我没有那个报文段的套接字。请不要再发送该报文段了”。当一台主机接收一个UDP分组，它的目的端口与进行中的UDP套接字不匹配，该主机发送一个特殊的ICMP数据报，这将在第4章中讨论。

既然我们已经对TCP连接管理有了深入的了解，我们再次回顾nmap端口扫描工具，并更为详细地研究它的工作原理。为了探索目标主机上的一个特定的TCP端口，如端口6789，nmap将对那台主机的目的端口6789发送一个特殊的TCP SYN报文段。有3种可能的输出：

- 源主机从目标主机接收到一个TCP SYNACK报文段。因为这意味着在目标主机上一个应用程序使用TCP端口6789运行，nmap返回“打开”。
- 源主机从目标主机接收到一个TCP RST报文段。这意味着该SYN报文段到达了目标主机，但目标主机没有运行一个使用TCP端口6789的应用程序。但攻击者至少知道发向该主机端口6789的报文段没有被源和目标主机之间的任何防火墙所阻挡。（将在第8章中讨论防火墙。）
- 源什么也没有收到。这很可能表明该SYN报文段被中间的防火墙所阻挡，无法到达目标主机。

nmap是一个功能强大的工具，该工具不仅能“侦察”打开的TCP端口，也能“侦察”打开的UDP端口，还能“侦察”防火墙及其配置，甚至能“侦察”应用程序的版本和操作系统。其中的大多数都能通过操作TCP连接管理报文段完成［Skoudis 2006］。读者能够从www.nmap.org下载nmap。

到此，我们介绍完了TCP中的差错控制和流量控制。在3.7节中，我们将回到TCP并更深入地研究TCP拥塞控制问题。然而，在此之前，我们先后退一步，在更广泛环境中讨论拥塞控制问题。

## 3.6　拥塞控制原理

在前面几节中，我们已经分析了面临分组丢失时用于提供可靠数据传输服务的基本原理及特定的TCP机制。我们以前讲过，在实践中，这种丢包一般是当网络变得拥塞时由于路由器缓存溢出引起的。分组重传因此作为网络拥塞的征兆（某个特定的运输层报文段的丢失）来对待，但是却无法处理导致网络拥塞的原因，因为有太多的源想以过高的速率发送数据。为了处理网络拥塞原因，需要一些机制以在面临网络拥塞时遏制发送方。

在本节中，我们考虑一般情况下的拥塞控制问题，试图理解为什么网络拥塞是一件坏

事情，网络拥塞是如何在上层应用得到的服务性能中明确地显露出来的？如何可用各种方法来避免网络拥塞或对它做出反应？这种对拥塞控制的更一般研究是恰当的，因为就像可靠数据传输一样，它在组网技术中的前 10 个基础性重要问题清单中位居前列。下面一节详细研究 TCP 的拥塞控制算法。

### 3.6.1 拥塞原因与代价

我们通过分析 3 个复杂性越来越高的发生拥塞的情况，开始对拥塞控制的一般性研究。在每种情况下，我们首先将看看出现拥塞的原因以及拥塞的代价（根据资源未被充分利用以及端系统得到的低劣服务性能来评价）。我们暂不关注如何对拥塞做出反应或避免拥塞，而是重点理解一个较为简单的问题，即随着主机增加其发送速率并使网络变得拥塞，这时会发生的情况。

#### 1. 情况 1：两个发送方和一台具有无穷大缓存的路由器

我们先考虑也许是最简单的拥塞情况：两台主机（A 和 B）都有一条连接，且这两条连接共享源与目的地之间的单跳路由，如图 3-43 所示。

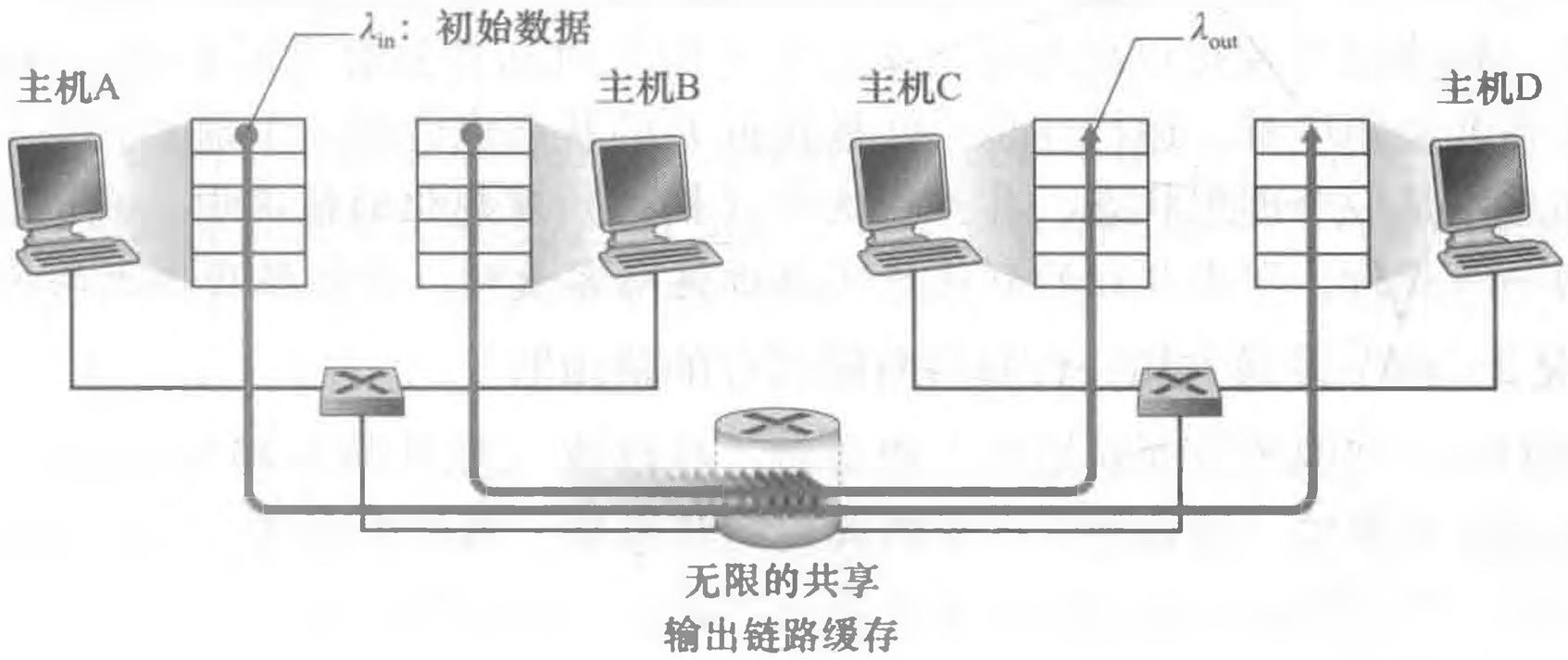


图 3-43 拥塞情况 1：两条连接共享具有无限大缓存的单跳路由

我们假设主机 A 中的应用程序以 $\lambda_{in}$ 字节/秒的平均速率将数据发送到连接中（例如，通过一个套接字将数据传递给运输层协议）。这些数据是初始数据，这意味着每个数据单元仅向套接字中发送一次。下面的运输层协议是一个简单的协议。数据被封装并发送；不执行差错恢复（如重传）、流量控制或拥塞控制。忽略由于添加运输层和较低层首部信息产生的额外开销，在第一种情况下，主机 A 向路由器提供流量的速率是 $\lambda_{in}$ 字节/秒。主机 B 也以同样的方式运行，为了简化问题，我们假设它也是以速率 $\lambda_{in}$ 字节/秒发送数据。来自主机 A 和主机 B 的分组通过一台路由器，在一段容量为 $R$ 的共享式输出链路上传输。该路由器带有缓存，可用于当分组到达速率超过该输出链路的容量时存储“入分组”。在此第一种情况下，我们将假设路由器有无限大的缓存空间。

图 3-44 描绘出了第一种情况下主机 A 的连接性能。左边的图形描绘了**每连接的吞吐量**（per-connection throughput）（接收方每秒接收的字节数）与该连接发送速率之间的函数关系。当发送速率在 0 ~ $R/2$ 之间时，接收方的吞吐量等于发送方的发送速率，即发送方发送的所有数据经有限时延后到达接收方。然而当发送速率超过 $R/2$ 时，它的吞吐量只能达 $R/2$。这个吞吐量上限是由两条连接之间共享链路容量造成的。链路完全不能以超过 $R/2$ 的稳定状态速率向接收方交付分组。无论主机 A 和主机 B 将其发送速率设置为多高，它们都不会看到超过 $R/2$ 的吞吐量。

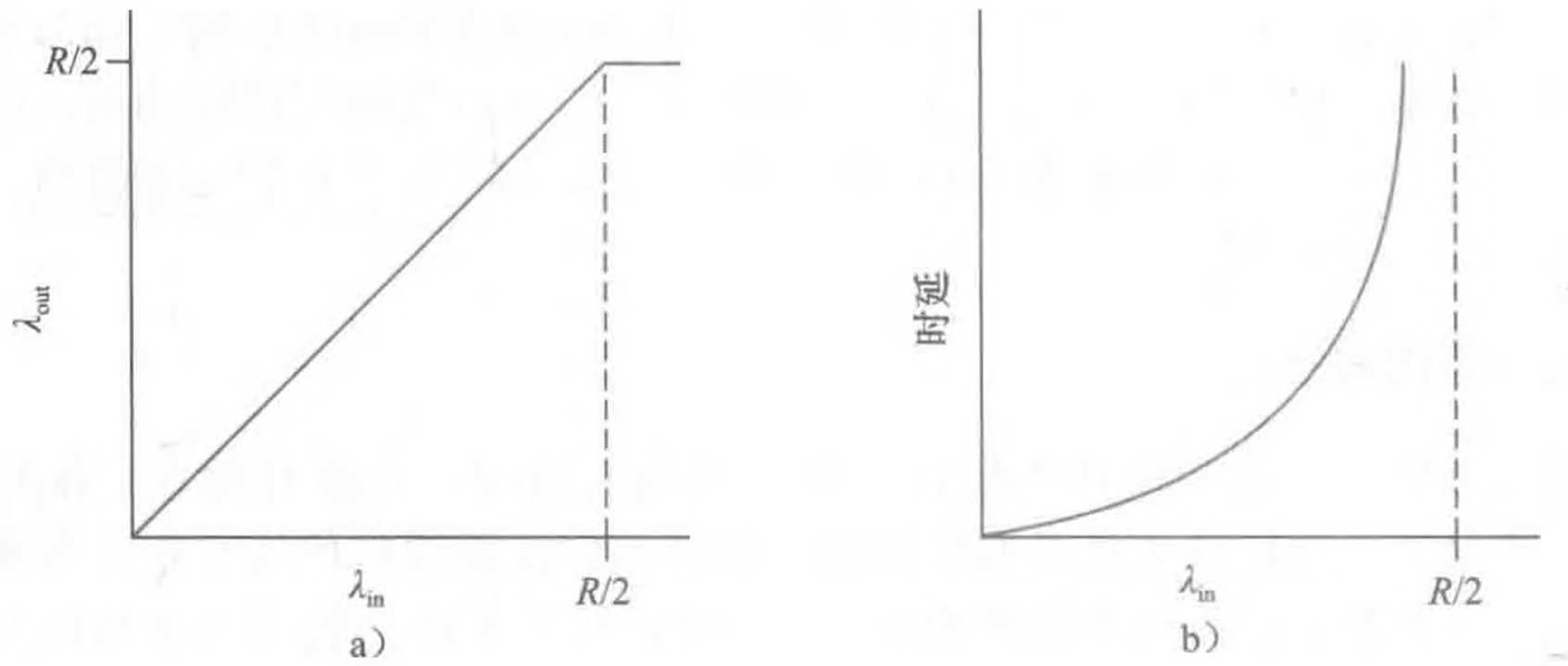


图 3-44　拥塞情况 1：吞吐量、时延与主机发送速率的函数关系

取得每连接 $R/2$ 的吞吐量实际上看起来可能是件好事，因为在将分组交付到目的地的过程中链路被充分利用了。但是，图 3-44b 的图形却显示了以接近链路容量的速率运行时产生的后果。当发送速率接近 $R/2$ 时（从左至右），平均时延就会越来越大。当发送速率超过 $R/2$ 时，路由器中的平均排队分组数就会无限增长，源与目的地之间的平均时延也会变成无穷大（假设这些连接以此发送速率运行无限长时间并且有无限量的缓存可用）。因此，虽然从吞吐量角度看，运行在总吞吐量接近 $R$ 的状态也许是一个理想状态，但从时延角度看，却远不是一个理想状态。甚至在这种（极端）理想化的情况中，我们已经发现了拥塞网络的一种代价，即当分组的到达速率接近链路容量时，分组经历巨大的排队时延。

### 2. 情况 2：两个发送方和一台具有有限缓存的路由器

现在我们从下列两个方面对情况 1 稍微做一些修改（参见图 3-45）。首先，假定路由器缓存的容量是有限的。这种现实世界的假设的结果是，当分组到达一个已满的缓存时会被丢弃。其次，我们假定每条连接都是可靠的。如果一个包含有运输层报文段的分组在路由器中被丢弃，那么它终将被发送方重传。由于分组可以被重传，所以我们现在必须更小心地使用发送速率这个术语。特别是我们再次以 $\lambda_{in}$ 字节/秒表示应用程序将初始数据发送到套接字中的速率。运输层向网络中发送报文段（含有初始数据或重传数据）的速率用 $\lambda'_{in}$ 字节/秒表示。$\lambda'_{in}$ 有时被称为网络的**供给载荷**（offered load）。

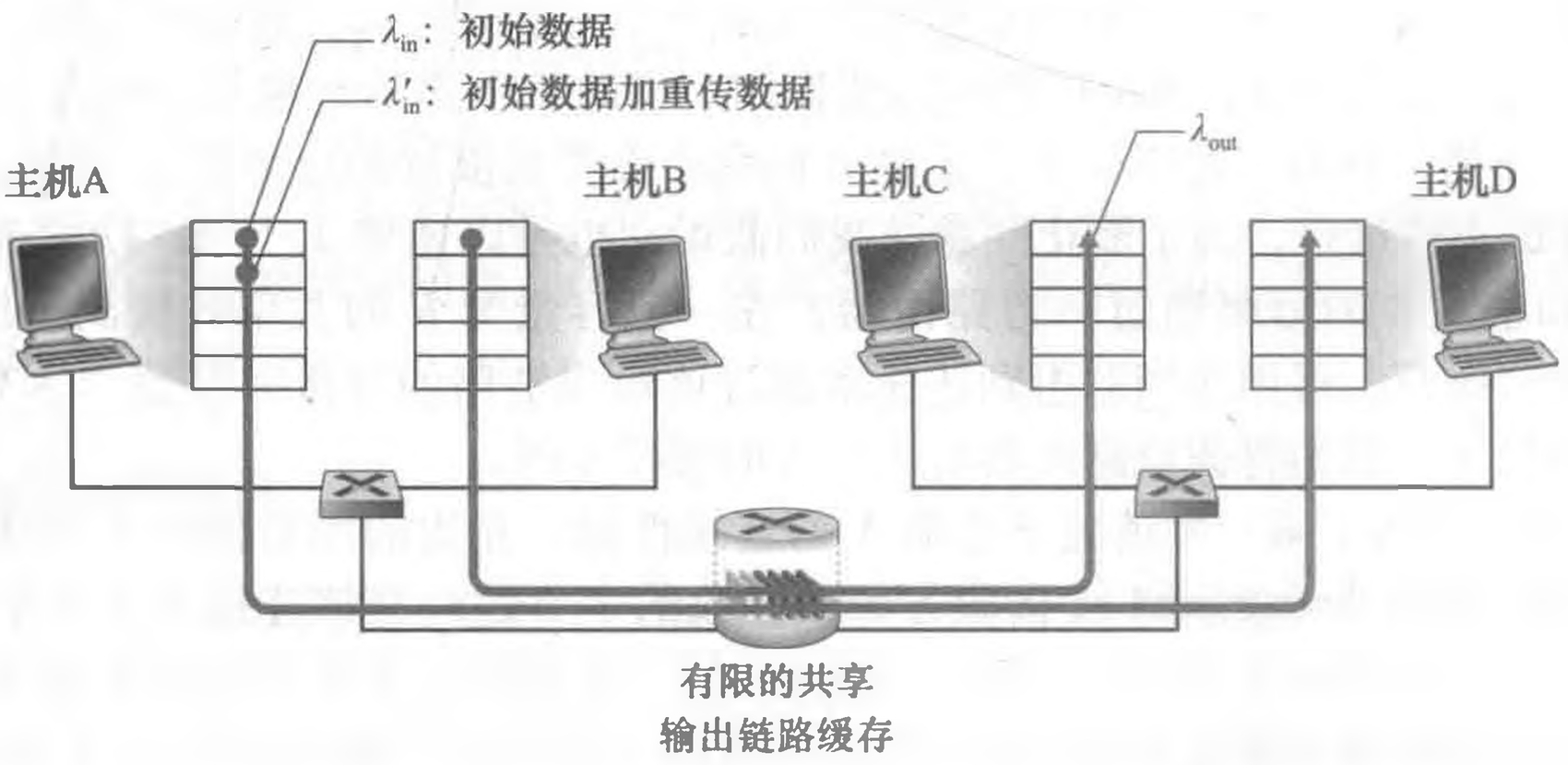


图 3-45　情况 2：（有重传的）两台主机与一台拥有有限缓存的路由器

在情况 2 下实现的性能强烈地依赖于重传的方式。首先，考虑一种不真实的情况，即

主机 A 能够以某种方式（不可思议地!）确定路由器中的缓存是否空闲，因而仅当缓存空闲时才发送一个分组。在这种情况下，将不会产生丢包，$\lambda_{in}$与 $\lambda'_{in}$相等，并且连接的吞吐量就等于 $\lambda_{in}$。图 3-46a 中描述了这种情况。从吞吐量的角度看，性能是理想的，即发送的每个分组都被接收到。注意到在这种情况下，平均主机发送速率不能超过 $R/2$，因为假定不会发生分组丢失。

接下来考虑一种更为真实的情况，发送方仅当在确定了一个分组已经丢失时才重传。（同样，所做的假设有一些弹性。然而，发送主机有可能将超时时间设置得足够长，以无形中使其确信一个还没有被确认的分组已经丢失。）在这种情况下，性能就可能与图 3-46b 所示的情况相似。为了理解这时发生的情况，考虑一下供给载荷 $\lambda'_{in}$（初始数据传输加上重传的总速率）等于 $R/2$ 的情况。根据图 3-46b，在这一供给载荷值时，数据被交付给接收方应用程序的速率是 $R/3$。因此，在所发送的 $0.5R$ 单位数据当中，从平均的角度说，$0.333R$ 字节/秒是初始数据，而 $0.166R$ 字节/秒是重传数据。我们在此看到了另一种网络拥塞的代价，即发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组。

最后，我们考虑下面一种情况：发送方也许会提前发生超时并重传在队列中已被推迟但还未丢失的分组。在这种情况下，初始数据分组和重传分组都可能到达接收方。当然，接收方只需要一份这样的分组副本就行了，重传分组将被丢弃。在这种情况下，路由器转发重传的初始分组副本是在做无用功，因为接收方已收到了该分组的初始版本。而路由器本可以利用链路的传输能力去发送另一个分组。这里，我们又看到了网络拥塞的另一种代价，即发送方在遇到大时延时所进行的不必要重传会引起路由器利用其链路带宽来转发不必要的分组副本。图 3-46c 显示了当假定每个分组被路由器转发（平均）两次时，吞吐量与供给载荷的对比情况。由于每个分组被转发两次，当其供给载荷接近 $R/2$ 时，其吞吐量将渐近 $R/4$。

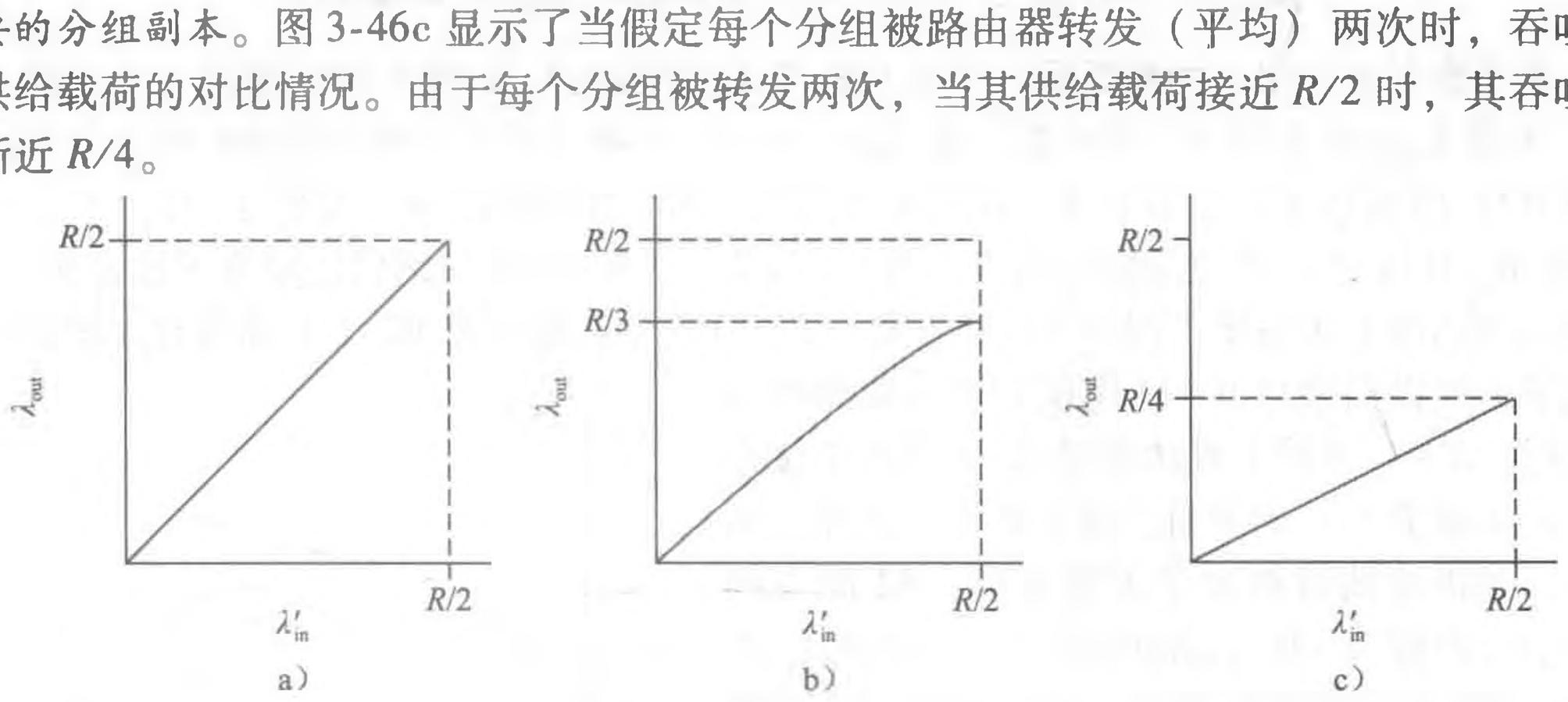


图 3-46 具有有限缓存时情况 2 的性能

### 3. 情况 3：4 个发送方和具有有限缓存的多台路由器及多跳路径

在最后一种拥塞情况中，有 4 台主机发送分组，每台都通过交叠的两跳路径传输，如图 3-47 所示。我们再次假设每台主机都采用超时/重传机制来实现可靠数据传输服务，所有的主机都有相同的 $\lambda_{in}$值，所有路由器的链路容量都是 $R$ 字节/秒。

我们考虑从主机 A 到主机 C 的连接，该连接经过路由器 R1 和 R2。A－C 连接与 D－B 连接共享路由器 R1，并与 B－D 连接共享路由器 R2。对极小的 $\lambda_{in}$值，路由器缓存的溢出是很少见的（与拥塞情况 1、拥塞情况 2 中的一样），吞吐量大致接近供给载荷。对稍大的 $\lambda_{in}$值，对应的吞吐量也更大，因为有更多的初始数据被发送到网络中并交付到目的地，溢出仍然很少。因此，对于较小的 $\lambda_{in}$，$\lambda_{in}$的增大会导致 $\lambda_{out}$的增大。

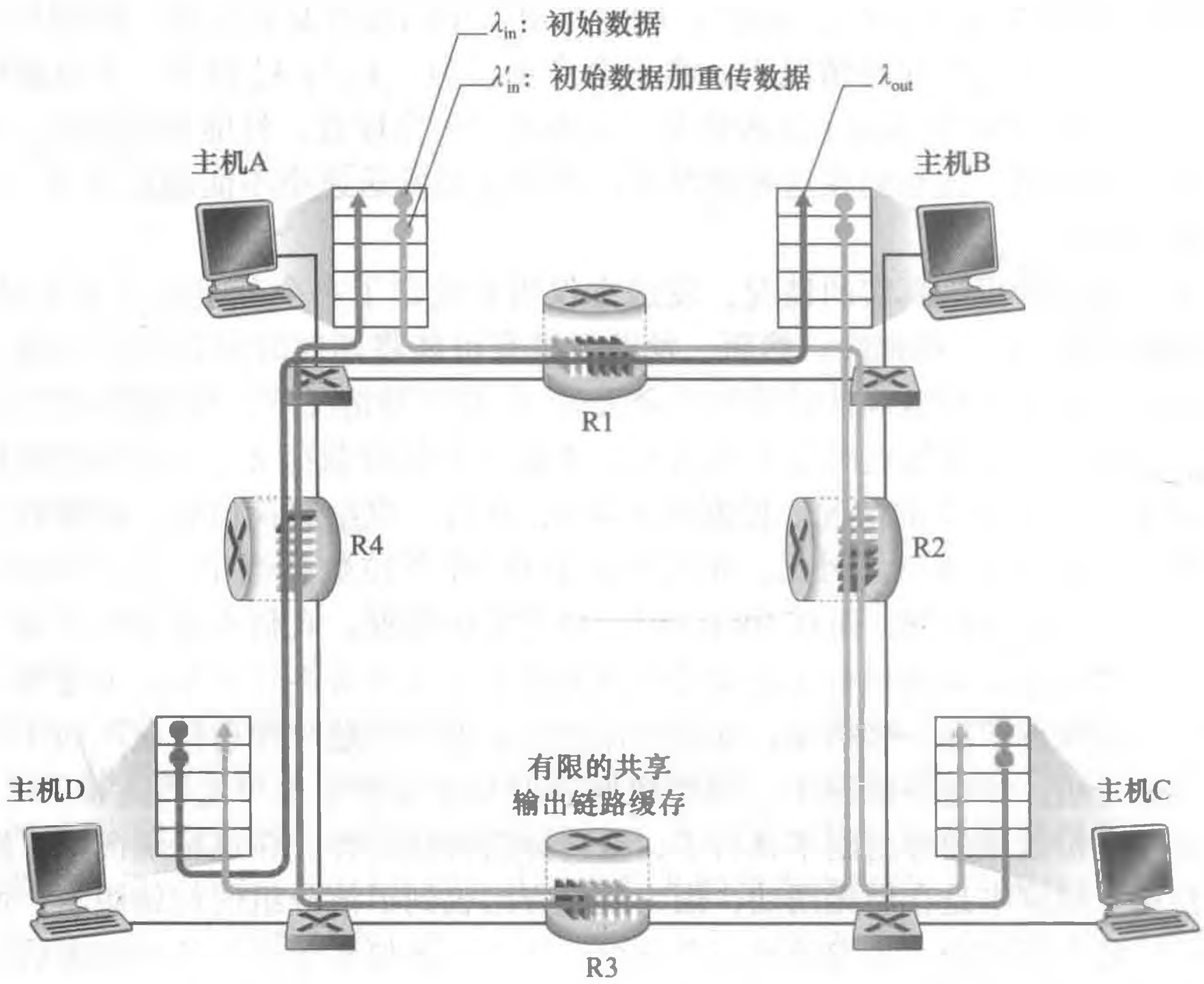


图 3-47　四个发送方和具有有限缓存的多台路由器及多跳路径

在考虑了流量很小的情况后，下面分析当 $\lambda_{in}$（因此 $\lambda'_{in}$）很大时的情况。考虑路由器 R2。不管 $\lambda_{in}$的值是多大，到达路由器 R2 的 A－C 流量（在经过路由器 R1 转发后到达路由器 R2）的到达速率至多是 $R$，也就是从 R1 到 R2 的链路容量。如果 $\lambda'_{in}$对于所有连接（包括 B－D 连接）来说是极大的值，那么在 R2 上，B－D 流量的到达速率可能会比 A－C 流量的到达速率大得多。因为 A－C 流量与 B－D 流量在路由器 R2 上必须为有限缓存空间而竞争，所以当来自 B－D 连接的供给载荷越来越大时，A－C 连接上成功通过 R2（即由于缓存溢出而未被丢失）的流量会越来越小。在极限情况下，当供给载荷趋近于无穷大时，R2 的空闲缓存会立即被 B－D 连接的分组占满，因而A－C 连接在 R2 上的吞吐量趋近于 0。这又一次说明在重载的极限情况下，A－C 端到端吞吐量将趋近于 0。这些考虑引发了供给载荷与吞吐量之间的权衡，如图 3-48 所示。

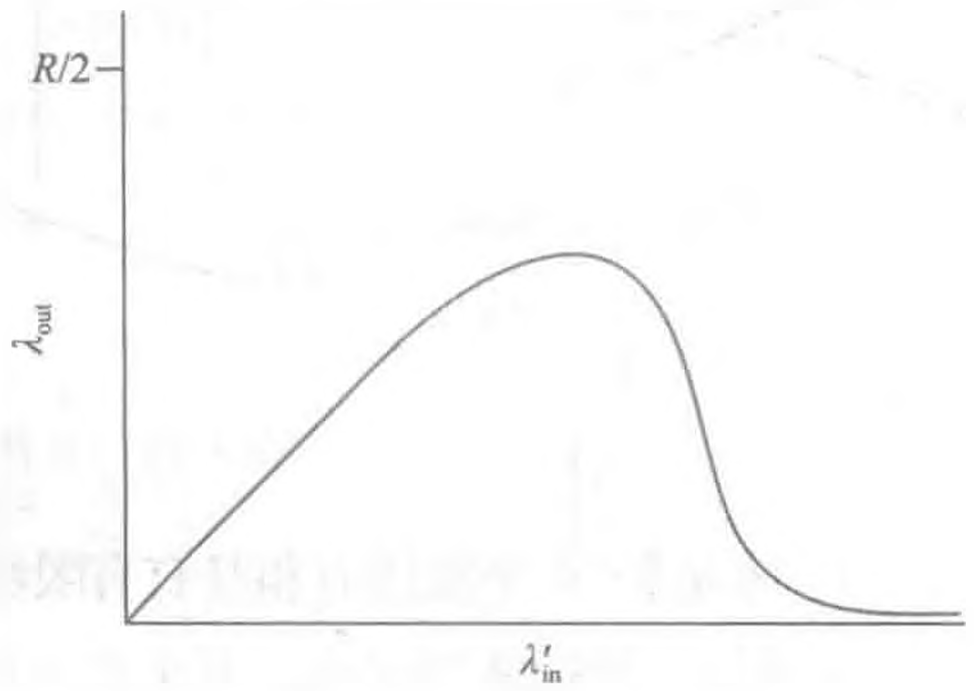


图 3-48　具有有限缓存和多跳路径时的情况 3 性能

当考虑由网络所做的浪费掉的工作量时，随着供给载荷的增加而使吞吐量最终减少的原因是明显的。在上面提到的大流量的情况中，每当有一个分组在第二跳路由器上被丢弃时，第一跳路由器所做的将分组转发到第二跳路由器的工作就是“劳而无功”的。如果第一跳路由器只是丢弃该分组并保持空闲，则网络中的情况是幸运的（更准确地说是糟糕的）。需要指出的是，第一跳路由器所使用的将分组转发到第二跳路由器的传输容量用来传送不同的分组可能更有效

益。（例如，当选择一个分组发送时，路由器最好优先考虑那些已经历过一定数量的上游路由器的分组。）所以，我们在此又看到了由于拥塞而丢弃分组的另一种代价，即当一个分组沿一条路径被丢弃时，每个上游路由器用于转发该分组到丢弃该分组而使用的传输容量最终被浪费掉了。

### 3.6.2 拥塞控制方法

在3.7节中，我们将详细研究TCP用于拥塞控制的特定方法。这里，我们指出在实践中所采用的两种主要拥塞控制方法，讨论特定的网络体系结构和具体使用这些方法的拥塞控制协议。

在最为宽泛的级别上，我们可根据网络层是否为运输层拥塞控制提供了显式帮助，来区分拥塞控制方法。

- 端到端拥塞控制。在端到端拥塞控制方法中，网络层没有为运输层拥塞控制提供显式支持。即使网络中存在拥塞，端系统也必须通过对网络行为的观察（如分组丢失与时延）来推断之。我们将在3.7.1节中将看到，TCP采用端到端的方法解决拥塞控制，因为IP层不会向端系统提供有关网络拥塞的反馈信息。TCP报文段的丢失（通过超时或3次冗余确认而得知）被认为是网络拥塞的一个迹象，TCP会相应地减小其窗口长度。我们还将看到关于TCP拥塞控制的一些最新建议，即使用增加的往返时延值作为网络拥塞程度增加的指示。
- 网络辅助的拥塞控制。在网络辅助的拥塞控制中，路由器向发送方提供关于网络中拥塞状态的显式反馈信息。这种反馈可以简单地用一个比特来指示链路中的拥塞情况。该方法在早期的IBM SNA［Schwartz 1982］、DEC DECnet［Jain 1989；Ramakrishnan 1990］和ATM［Black 1995］等体系结构中被采用。更复杂的网络反馈也是可能的。例如，在ATM **可用比特率**（Available Bite Rate，ABR）拥塞控制中，路由器显式地通知发送方它（路由器）能在输出链路上支持的最大主机发送速率。如上面所提到的，默认因特网版本的IP和TCP采用端到端拥塞控制方法。然而，我们在3.7.2节中将看到，最近IP和TCP也能够选择性地实现网络辅助拥塞控制。

对于网络辅助的拥塞控制，拥塞信息从网络反馈到发送方通常有两种方式，如图3-49

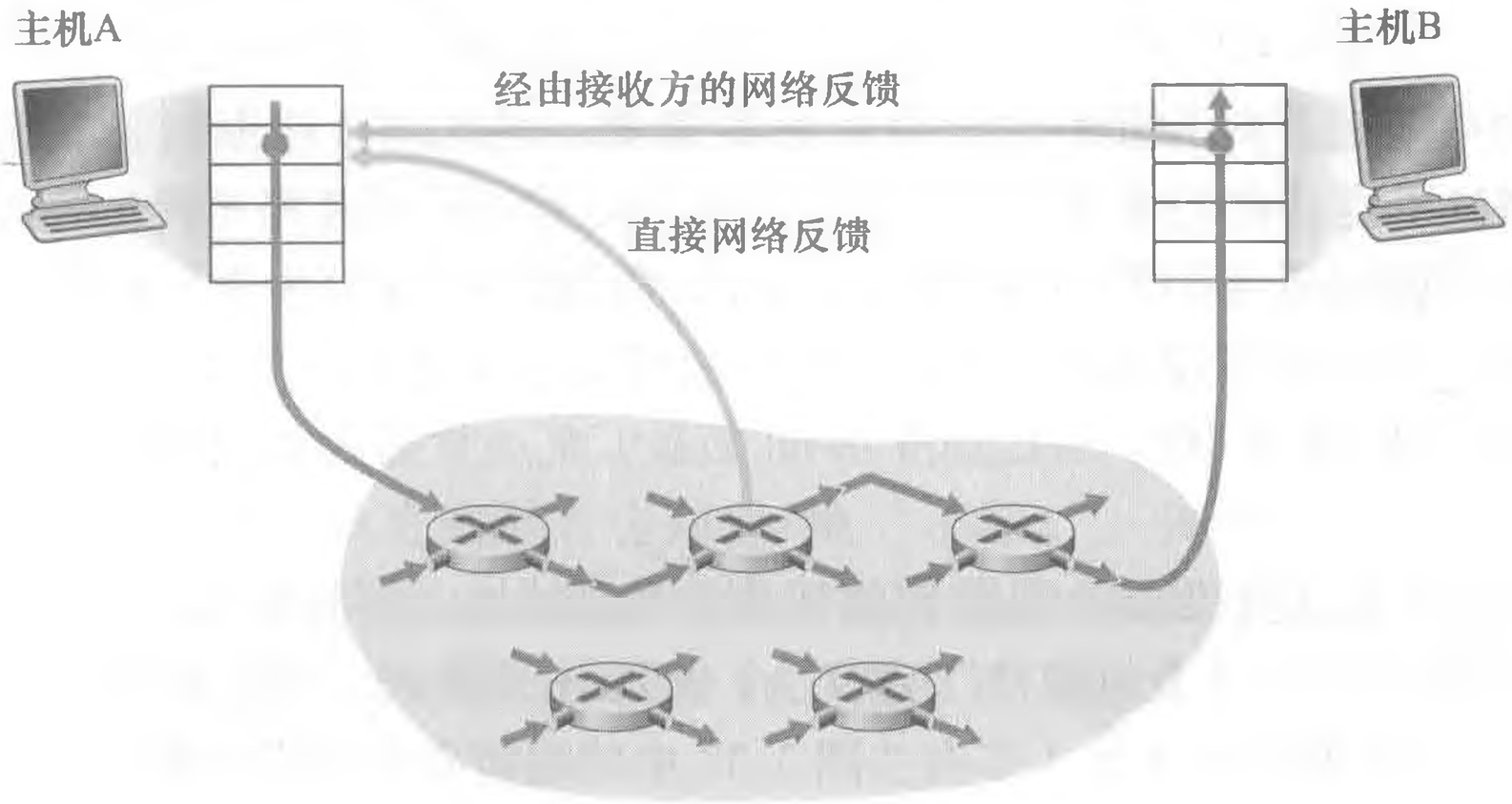


图3-49 网络指示拥塞信息的两种反馈路径

所示。直接反馈信息可以由网络路由器发给发送方。这种方式的通知通常采用了一种**阻塞分组**（choke packet）的形式（主要是说："我拥塞了！"）。更为通用的第二种形式的通知是，路由器标记或更新从发送方流向接收方的分组中的某个字段来指示拥塞的产生。一旦收到一个标记的分组后，接收方就会向发送方通知该网络拥塞指示。注意到后一种形式的通知至少要经过一个完整的往返时间。

## 3.7　TCP 拥塞控制

在本节中，我们再次来学习 TCP。如我们在 3.5 节所见，TCP 为运行在不同主机上的两个进程之间提供了可靠传输服务。TCP 的另一个关键部分就是其拥塞控制机制。如在前一节所指出，TCP 必须使用端到端拥塞控制而不是使网络辅助的拥塞控制，因为 IP 层不向端系统提供显式的网络拥塞反馈。

TCP 所采用的方法是让每一个发送方根据所感知到的网络拥塞程度来限制其能向连接发送流量的速率。如果一个 TCP 发送方感知从它到目的地之间的路径上没什么拥塞，则 TCP 发送方增加其发送速率；如果发送方感知沿着该路径有拥塞，则发送方就会降低其发送速率。但是这种方法提出了三个问题。第一，一个 TCP 发送方如何限制它向其连接发送流量的速率呢？第二，一个 TCP 发送方如何感知从它到目的地之间的路径上存在拥塞呢？第三，当发送方感知到端到端的拥塞时，采用何种算法来改变其发送速率呢？

我们首先分析一下 TCP 发送方是如何限制向其连接发送流量的。在 3.5 节中我们看到，TCP 连接的每一端都是由一个接收缓存、一个发送缓存和几个变量（LastByteRead、rwnd 等）组成。运行在发送方的 TCP 拥塞控制机制跟踪一个额外的变量，即**拥塞窗口**（congestion window）。拥塞窗口表示为 cwnd，它对一个 TCP 发送方能向网络中发送流量的速率进行了限制。特别是，在一个发送方中未被确认的数据量不会超过 cwnd 与 rwnd 中的最小值，即

$$\text{LastByteSent} - \text{LastByteAcked} \leqslant \min\{\text{cwnd}, \text{rwnd}\}$$

为了关注拥塞控制（与流量控制形成对比），我们后面假设 TCP 接收缓存足够大，以至可以忽略接收窗口的限制；因此在发送方中未被确认的数据量仅受限于 cwnd。我们还假设发送方总是有数据要发送，即在拥塞窗口中的所有报文段要被发送。

上面的约束限制了发送方中未被确认的数据量，因此间接地限制了发送方的发送速率。为了理解这一点，我们来考虑一个丢包和发送时延均可以忽略不计的连接。因此粗略地讲，在每个往返时间（RTT）的起始点，上面的限制条件允许发送方向该连接发送 cwnd 个字节的数据，在该 RTT 结束时发送方接收对数据的确认报文。因此，该发送方的发送速率大概是 cwnd/RTT 字节/秒。通过调节 cwnd 的值，发送方因此能调整它向连接发送数据的速率。

我们接下来考虑 TCP 发送方是如何感知在它与目的地之间的路径上出现了拥塞的。我们将一个 TCP 发送方的"丢包事件"定义为：要么出现超时，要么收到来自接收方的 3 个冗余 ACK。（回想我们在 3.5.4 节有关图 3-33 中的超时事件的讨论和收到 3 个冗余 ACK 后包括快速重传的后继修改。）当出现过度的拥塞时，在沿着这条路径上的一台（或多台）路由器的缓存会溢出，引起一个数据报（包含一个 TCP 报文段）被丢弃。丢弃的数

据报接着会引起发送方的丢包事件（要么超时或收到 3 个冗余 ACK），发送方就认为在发送方到接收方的路径上出现了拥塞的指示。

考虑了拥塞检测问题后，我们接下来考虑网络没有拥塞这种更为乐观的情况，即没有出现丢包事件的情况。在此情况下，在 TCP 的发送方将收到对于以前未确认报文段的确认。如我们将看到的那样，TCP 将这些确认的到达作为一切正常的指示，即在网络上传输的报文段正被成功地交付给目的地，并使用确认来增加窗口的长度（及其传输速率）。注意到如果确认以相当慢的速率到达（例如，如果该端到端路径具有高时延或包含一段低带宽链路），则该拥塞窗口将以相当慢的速率增加。在另一方面，如果确认以高速率到达，则该拥塞窗口将会更为迅速地增大。因为 TCP 使用确认来触发（或计时）增大它的拥塞窗口长度，TCP 被说成是**自计时**（self-clocking）的。

给定调节 cwnd 值以控制发送速率的机制，关键的问题依然存在：TCP 发送方怎样确定它应当发送的速率呢？如果众多 TCP 发送方总体上发送太快，它们能够拥塞网络，导致我们在图 3-48 中看到的拥塞崩溃。事实上，为了应对在较早 TCP 版本下观察到的因特网拥塞崩溃［Jacobson 1988］，研发了该版本的 TCP（我们马上将学习它）。然而，如果 TCP 发送方过于谨慎，发送太慢，它们不能充分利用网络的带宽；这就是说，TCP 发送方能够以更高的速率发送而不会使网络拥塞。那么 TCP 发送方如何确定它们的发送速率，既使得网络不会拥塞，与此同时又能充分利用所有可用的带宽？TCP 发送方是显式地协作，或存在一种分布式方法使 TCP 发送方能够仅基于本地信息设置它们的发送速率？TCP 使用下列指导性原则回答这些问题：

- 一个丢失的报文段表意味着拥塞，因此当丢失报文段时应当降低 TCP 发送方的速率。回想在 3.5.4 节中的讨论，对于给定报文段，一个超时事件或四个确认（一个初始 ACK 和其后的三个冗余 ACK）被解释为跟随该四个 ACK 的报文段的“丢包事件”的一种隐含的指示。从拥塞控制的观点看，该问题是 TCP 发送方应当如何减小它的拥塞窗口长度，即减小其发送速率，以应对这种推测的丢包事件。
- 一个确认报文段指示该网络正在向接收方交付发送方的报文段，因此，当对先前未确认报文段的确认到达时，能够增加发送方的速率。确认的到达被认为是一切顺利的隐含指示，即报文段正从发送方成功地交付给接收方，因此该网络不拥塞。拥塞窗口长度因此能够增加。
- 带宽探测。给定 ACK 指示源到目的地路径无拥塞，而丢包事件指示路径拥塞，TCP 调节其传输速率的策略是增加其速率以响应到达的 ACK，除非出现丢包事件，此时才减小传输速率。因此，为探测拥塞开始出现的速率，TCP 发送方增加它的传输速率，从该速率后退，进而再次开始探测，看看拥塞开始速率是否发生了变化。TCP 发送方的行为也许类似于要求（并得到）越来越多糖果的孩子，直到最后告知他/她“不行!”，孩子后退一点，然后过一会儿再次开始提出请求。注意到网络中没有明确的拥塞状态信令，即 ACK 和丢包事件充当了隐式信号，并且每个 TCP 发送方根据异步于其他 TCP 发送方的本地信息而行动。

概述了 TCP 拥塞控制后，现在是我们考虑广受赞誉的 **TCP 拥塞控制算法**（TCP congestion control algorithm）细节的时候了，该算法首先在［Jacobson 1988］中描述并