且在［RFC 5681］中标准化。该算法包括3个主要部分：①慢启动；②拥塞避免；③快速恢复。慢启动和拥塞避免是TCP的强制部分，两者的差异在于对收到的ACK做出反应时增加cwnd长度的方式。我们很快将会看到慢启动比拥塞避免能更快地增加cwnd的长度（不要被名称所迷惑!）。快速恢复是推荐部分，对TCP发送方并非是必需的。

1. 慢启动

当一条TCP连接开始时，cwnd的值通常初始置为一个MSS的较小值［RFC 3390］，这就使得初始发送速率大约为MSS/RTT。例如，如果MSS=500字节且RTT=200ms，则得到的初始发送速率大约只有20kbps。由于对TCP发送方而言，可用带宽可能比MSS/RTT大得多，TCP发送方希望迅速找到可用带宽的数量。因此，在**慢启动**（slow-start）状态，cwnd的值以1个MSS开始并且每当传输的报文段首次被确认就增加1个MSS。在图3-50所示的例子中，TCP向网络发送第一个报文段并等待一个确认。当该确认到达时，TCP发送方将拥塞窗口增加一个MSS，并发送出两个最大长度的报文段。这两个报文段被确认，则发送方对每个确认报文段将拥塞窗口增加一个MSS，使得拥塞窗口变为4个MSS，并这样下去。这一过程每过一个RTT，发送速率就翻番。因此，TCP发送速率起始慢，但在慢启动阶段以指数增长。

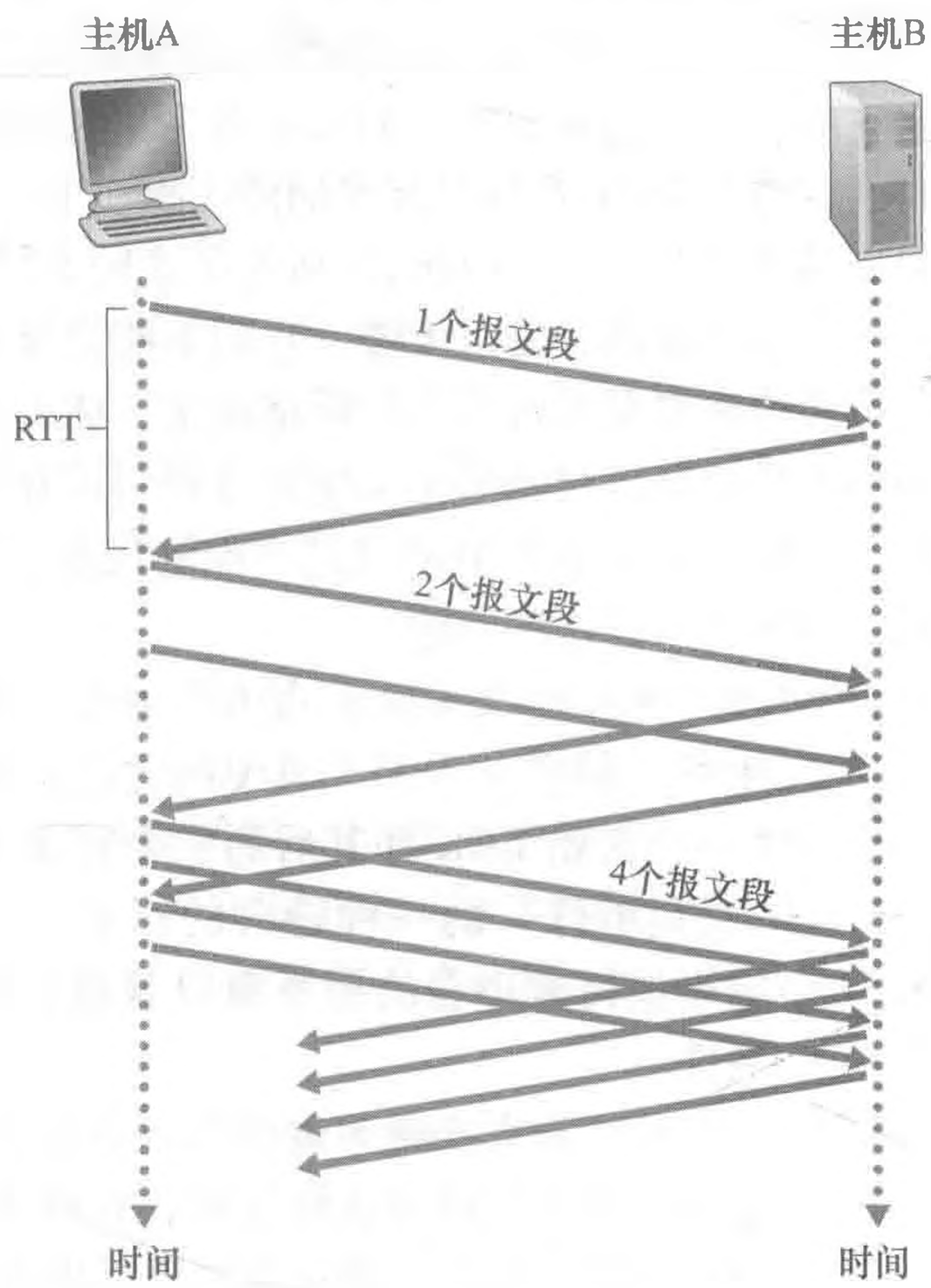

图3-50　TCP慢启动

但是，何时结束这种指数增长呢？慢启动对这个问题提供了几种答案。首先，如果存在一个由超时指示的丢包事件（即拥塞），TCP发送方将cwnd设置为1并重新开始慢启动过程。它还将第二个状态变量的值ssthresh（“慢启动阈值”的速记）设置为cwnd/2，即当检测到拥塞时将ssthresh置为拥塞窗口值的一半。慢启动结束的第二种方式是直接与ssthresh的值相关联。因为当检测到拥塞时ssthresh设为cwnd的值一半，当到达或超过ssthresh的值时，继续使cwnd翻番可能有些鲁莽。因此，当cwnd的值等于ssthresh时，结束慢启动并且TCP转移到拥塞避免模式。我们将会看到，当进入拥塞避免模式时，TCP更为谨慎地增加cwnd。最后一种结束慢启动的方式是，如果检测到3个冗余ACK，这时TCP执行一种快速重传（参见3.5.4节）并进入快速恢复状态，后面将讨论相关内容。慢启动中的TCP行为总结在图3-51中的TCP拥塞控制的FSM描述中。慢启动算法最早源于［Jacobson 1988］；在［Jain 1986］中独立地提出了一种类似于慢启动的方法。

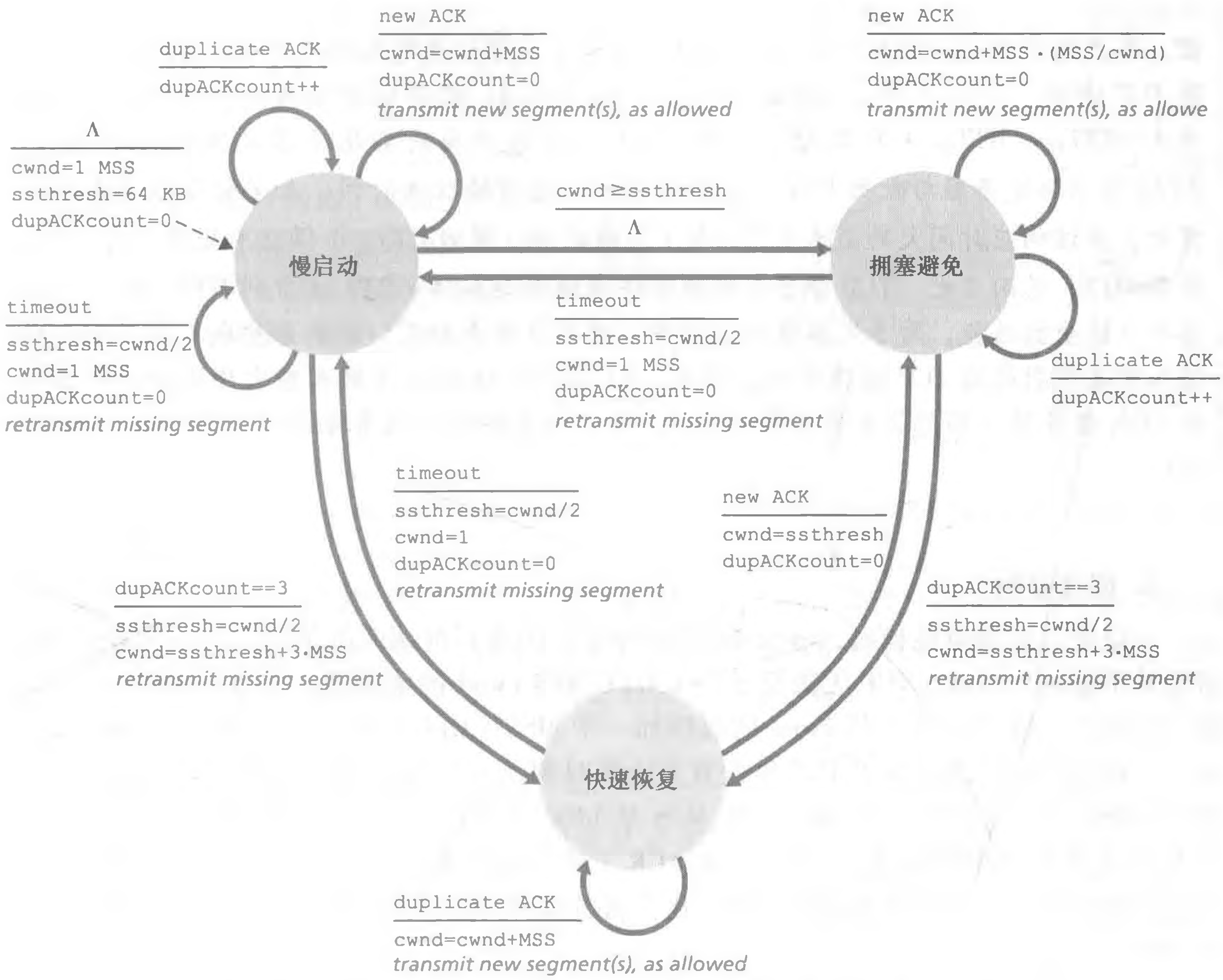

图 3-51 TCP 拥塞控制的 FSM 描述

实 践 原 则

TCP 分岔：优化云服务的性能

对于诸如搜索、电子邮件和社交网络等云服务，非常希望提供高水平的响应性，给用户一种完美的印象，即这些服务正运行在它们自己的端系统（包括其智能手机）中。因为用户经常位于远离数据中心的地方，而这些数据中心负责为云服务关联的动态内容提供服务。实际上，如果端系统远离数据中心，则 RTT 将会很大，会由于 TCP 慢启动潜在地导致低劣的响应时间性能。

作为一个学习案例，考虑接收对某搜索问题的响应中的时延。通常，服务器在慢启动期间交付响应要求三个 TCP 窗口［Pathak 2010］。所以从某端系统发起一条 TCP 连接到它收到该响应的最后一个分组的时间粗略是 4 * RTT（用于建立 TCP 连接的一个 RTT 加上用于 3 个数据窗口的 3 个 RTT），再加上在数据中心中处理的时间。对于一个相当小的查询来说，这些 RTT 时延导致其返回搜索结果中显而易见的时延。此外，在接入网中可能有较大的丢包，导致 TCP 重传甚至较大的时延。

缓解这个问题和改善用户感受到的性能的一个途径是：①部署邻近用户的前端服务器；②在该前端服务器利用 **TCP 分岔**（TCP splitting）来分裂 TCP 连接。借助于 TCP 分

岔，客户向邻近前端连接一条 TCP 连接，并且该前端以非常大的窗口向数据中心维护一条 TCP 连接［Tariq 2008，Pathak 2010，Chen 2011］。使用这种方法，响应时间大致变为 $4 * RTT_{FE} + RTT_{BE}$ + 处理时间，其中 RTT_{FE} 是客户与前端服务器之间的往返时间，RTT_{BE} 是前端服务器与数据中心（后端服务器）之间的往返时间。如果前端服务器邻近客户，则该响应时间大约变为 RTT_{BE} 加上处理时间，因为 RTT_{FE} 小得微不足道并且 RTT_{BE} 约为 RTT。总而言之，TCP 分岔大约能够将网络时延从 4 * RTT 减少到 RTT，极大地改善用户感受的性能，对于远离最近数据中心的用户更是如此。TCP 分岔也有助于减少因接入网丢包引起的 TCP 重传时延。今天，Google 和 Akamai 在接入网中广泛利用了它们的 CDN 服务器（回想2.6 节中的讨论），为它们支持的云服务来执行 TCP 分岔［Chen 2011］。

2. 拥塞避免

一旦进入拥塞避免状态，cwnd 的值大约是上次遇到拥塞时的值的一半，即距离拥塞可能并不遥远！因此，TCP 无法每过一个 RTT 再将 cwnd 的值翻番，而是采用了一种较为保守的方法，每个 RTT 只将 cwnd 的值增加一个 MSS［RFC 5681］。这能够以几种方式完成。一种通用的方法是对于 TCP 发送方无论何时到达一个新的确认，就将 cwnd 增加一个 MSS（MSS/cwnd）字节。例如，如果 MSS 是 1460 字节并且 cwnd 是 14 600 字节，则在一个 RTT 内发送 10 个报文段。每个到达 ACK（假定每个报文段一个 ACK）增加 1/10MSS 的拥塞窗口长度，因此在收到对所有 10 个报文段的确认后，拥塞窗口的值将增加了一个 MSS。

但是何时应当结束拥塞避免的线性增长（每 RTT 1MSS）呢？当出现超时时，TCP 的拥塞避免算法行为相同。与慢启动的情况一样，cwnd 的值被设置为 1 个 MSS，当丢包事件出现时，ssthresh 的值被更新为 cwnd 值的一半。然而，前面讲过丢包事件也能由一个三个冗余 ACK 事件触发。在这种情况下，网络继续从发送方向接收方交付报文段（就像由收到冗余 ACK 所指示的那样）。因此 TCP 对这种丢包事件的行为，相比于超时指示的丢包，应当不那么剧烈：TCP 将 cwnd 的值减半（为使测量结果更好，计及已收到的 3 个冗余的 ACK 要加上 3 个 MSS），并且当收到 3 个冗余的 ACK，将 ssthresh 的值记录为 cwnd 的值的一半。接下来进入快速恢复状态。

3. 快速恢复

在快速恢复中，对于引起 TCP 进入快速恢复状态的缺失报文段，对收到的每个冗余的 ACK，cwnd 的值增加一个 MSS。最终，当对丢失报文段的一个 ACK 到达时，TCP 在降低 cwnd 后进入拥塞避免状态。如果出现超时事件，快速恢复在执行如同在慢启动和拥塞避免中相同的动作后，迁移到慢启动状态：当丢包事件出现时，cwnd 的值被设置为 1 个 MSS，并且 ssthresh 的值设置为 cwnd 值的一半。

快速恢复是 TCP 推荐的而非必需的构件［RFC 5681］。有趣的是，一种称为 TCP Tahoe 的 TCP 早期版本，不管是发生超时指示的丢包事件，还是发生 3 个冗余 ACK 指示的丢包事件，都无条件地将其拥塞窗口减至 1 个 MSS，并进入慢启动阶段。TCP 的较新版本 TCP Reno，则综合了快速恢复。

图 3-52 图示了 Reno 版 TCP 与 Tahoe 版 TCP 的拥塞控制窗口的演化情况。在该图中，阈值初始等于 8 个 MSS。在前 8 个传输回合，Tahoe 和 Reno 采取了相同的动作。拥塞窗口在慢启动阶段以指数速度快速爬升，在第 4 轮传输时到达了阈值。然后拥塞窗口以线性速度爬升，直到在第 8 轮传输后出现3 个冗余 ACK。注意到当该丢包事件发生时，拥塞窗口值为 12 × MSS。于是 ssthresh 的值被设置为 0.5 × cwnd = 6 × MSS。在 TCP Reno 下，拥塞窗口被设置为 cwnd = 9MSS，然后线性地增长。在 TCP Tahoe 下，拥塞窗口被设置为 1 个 MSS，然后呈指数增长，直至到达 ssthresh 值为止，在这个点它开始线性增长。

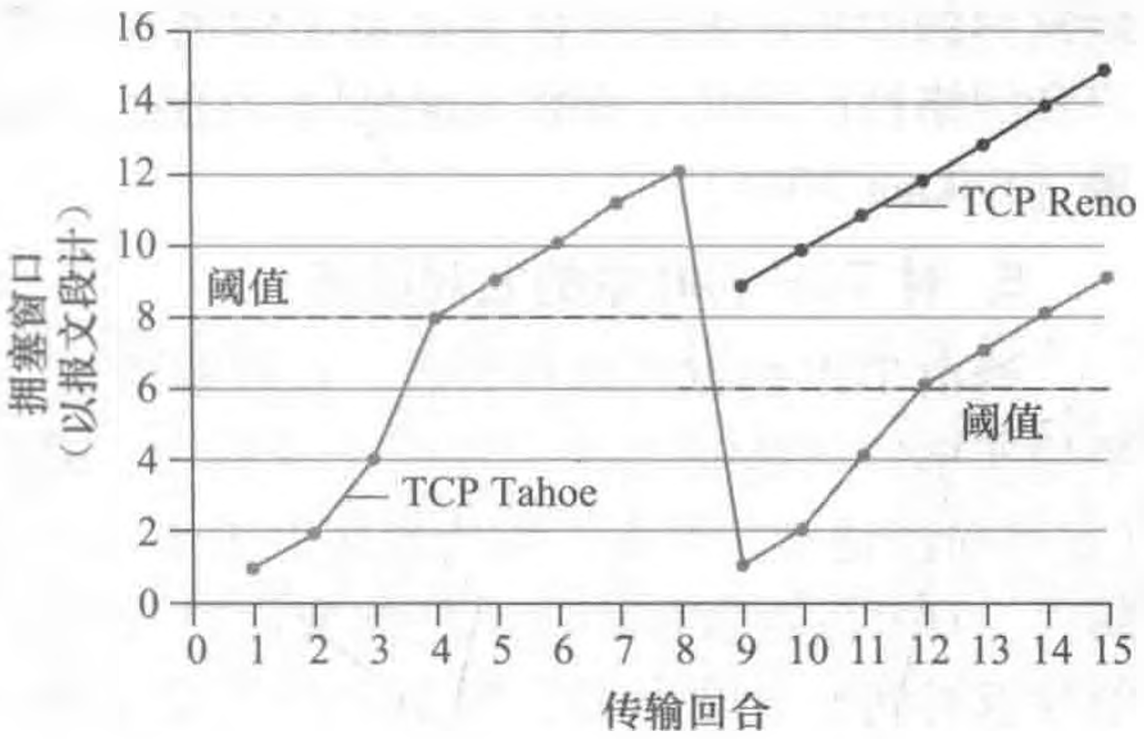

图 3-52 TCP 拥塞窗口的演化（Tahoe 和 Reno）

图 3-51 表示了 TCP 拥塞控制算法（即慢启动、拥塞避免和快速恢复）的完整 FSM 描述。该图也指示了新报文段的传输或重传的报文段可能出现的位置。尽管区分 TCP 差错控制/重传与 TCP 拥塞控制非常重要，但是注意到 TCP 这两个方面交织链接的方式也很重要。

4. TCP 拥塞控制：回顾

在深入了解慢启动、拥塞避免和快速恢复的细节后，现在有必要退回来回顾一下全局。忽略一条连接开始时初始的慢启动阶段，假定丢包由 3 个冗余的 ACK 而不是超时指示，TCP 的拥塞控制是：每个 RTT 内 cwnd 线性（加性）增加 1MSS，然后出现 3 个冗余 ACK 事件时 cwnd 减半（乘性减）。因此，TCP 拥塞控制常常被称为**加性增、乘性减**（Additive-Increase，Multiplicative-Decrease，AIMD）拥塞控制方式。AIMD 拥塞控制引发了在图 3-53 中所示的“锯齿”行为，这也很好地图示了我们前面 TCP 检测带宽时的直觉，即 TCP 线性地增加它的拥塞窗口长度（因此增加其传输速率），直到出现 3 个冗余 ACK 事件。然后以 2 个因子来减少它的拥塞窗口长度，然后又开始了线性增长，探测是否还有另外的可用带宽。

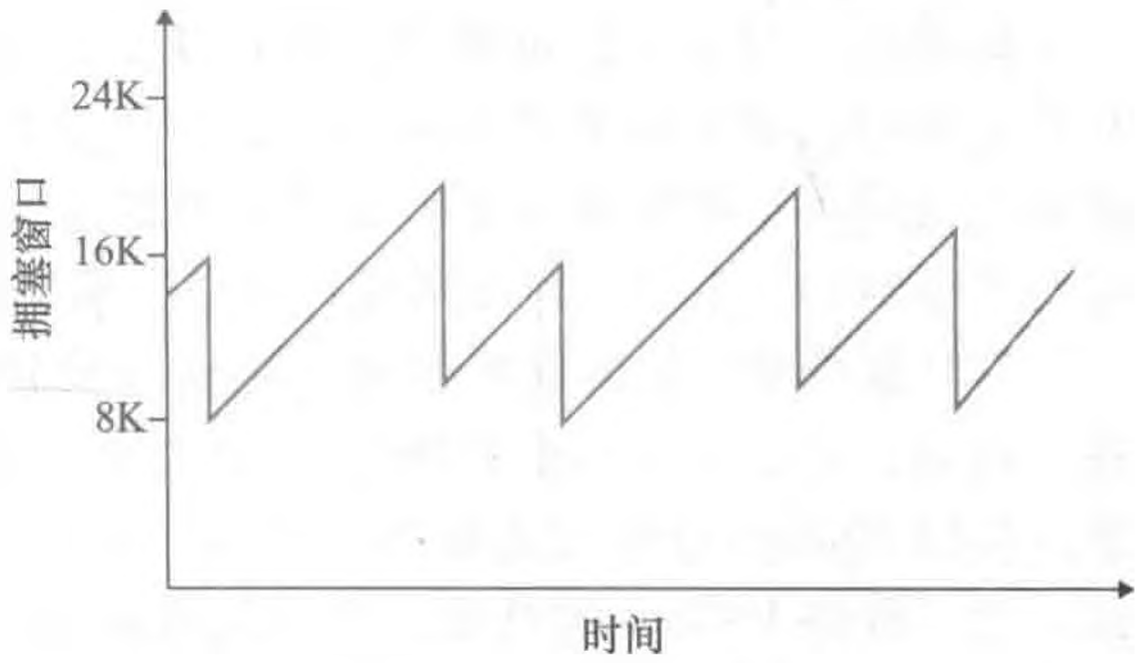

图 3-53 加性增、乘性减的拥塞控制

如前所述，许多 TCP 实现采用了 Reno 算法［Padhye 2001］。Reno 算法的许多变种已被提出［RFC 3782；RFC 2018］。TCP Vegas 算法［Brakmo 1995；Ahn 1995］试图在维持较好的吞吐量的同时避免拥塞。Vegas 的基本思想是：①在分组丢失发生之前，在源与目的地之间检测路由器中的拥塞；②当检测出快要发生的分组丢失时，线性地降低发送速率。快要发生的分组丢失是通过观察 RTT 来预测的。分组的 RTT 越长，路由器中的拥塞越严重。到 2015 年年底，TCP 的 Ubuntu Linux 实现默认提供了慢启动、拥塞避免、快速恢复、快速重传和 SACK，也提供了诸如 TCP Vegas 和 BIC［Xu 2004］等其他拥塞控制算法。对于许多特色 TCP 的综述参见［Afanasyev 2010］。

TCP AIMD 算法基于大量的工程见解和在运行网络中的拥塞控制经验而开发。在 TCP 研发后的十年，理论分析显示 TCP 的拥塞控制算法用做一种分布式异步优化算法，使得用户和网络性能的几个重要方面被同时优化［Kelly 1998］。拥塞控制的丰富理论已经得到发展［Srikant 2004］。

5. 对 TCP 吞吐量的宏观描述

给出 TCP 的锯齿状行为后，自然要考虑一个长存活期的 TCP 连接的平均吞吐量（即平均速率）可能是多少。在这个分析中，我们将忽略在超时事件后出现的慢启动阶段。（这些阶段通常非常短，因为发送方很快就以指数增长离开该阶段。）在一个特定的往返间隔内，TCP 发送数据的速率是拥塞窗口与当前 RTT 的函数。当窗口长度是 w 字节，且当前往返时间是 RTT 秒时，则 TCP 的发送速率大约是 w/RTT。于是，TCP 通过每经过 1 个 RTT 将 w 增加 1 个 MSS 探测出额外的带宽，直到一个丢包事件发生为止。当一个丢包事件发生时，用 W 表示 w 的值。假设在连接持续期间 RTT 和 W 几乎不变，那么 TCP 的传输速率在 $W/(2 \times \text{RTT})$ 到 W/RTT 之间变化。

这些假设导出了 TCP 稳态行为的一个高度简化的宏观模型。当速率增长至 W/RTT 时，网络丢弃来自连接的分组；然后发送速率就会减半，进而每过一个 RTT 就发送速率增加 MSS/RTT，直到再次达到 W/RTT 为止。这一过程不断地自我重复。因为 TCP 吞吐量（即速率）在两个极值之间线性增长，所以我们有

$$\text{一条连接的平均吞吐量} = \frac{0.75 \times W}{\text{RTT}}$$

通过这个高度理想化的 TCP 稳态动态性模型，我们可以推出一个将连接的丢包率与可用带宽联系起来的有趣表达式［Mahdavi 1997］。这个推导将在课后习题中概要给出。一个根据经验建立的并与测量数据一致的更复杂模型参见［Padhye 2000］。

6. 经高带宽路径的 TCP

认识到下列事实是重要的：TCP 拥塞控制已经演化了多年并仍在继续演化。对当前 TCP 变量的总结和 TCP 演化的讨论，参见［Floyd 2001；RFC 5681；Afanasyev 2010］。以往对因特网有益的东西（那时大量的 TCP 连接承载的是 SMTP、FTP 和 Telnet 流量），不一定对当今 HTTP 主宰的因特网或具有难以想象的服务的未来因特网还是有益的。

TCP 继续演化的需求能够通过考虑网格和云计算应用所需要的高速 TCP 连接加以阐述。例如，考虑一条具有 1500 字节报文段和 100ms RTT 的 TCP 连接，假定我们要通过这条连接以 10Gbps 速率发送数据。根据［RFC 3649］，我们注意到使用上述 TCP 吞吐量公式，为了取得 10Gbps 吞吐量，平均拥塞窗口长度将需要是 83 333 个报文段。对如此大量的报文段，使我们相当关注这 83 333 个传输中的报文段也许会丢失。在丢失的情况下，将会出现什么情况呢？或者以另一种方式说，这些传输的报文段能以何种比例丢失，使得在图 3-52 中列出的 TCP 拥塞控制算法仍能取得所希望的 10Gbps 速率？在本章的课后习题中，要求读者推导出一条 TCP 连接的吞吐量公式，该公式作为丢包率（L）、往返时间（RTT）和最大报文段长度（MSS）的函数：

$$\text{一条连接的平均吞吐量} = \frac{1.22 \times \text{MSS}}{\text{RTT}\sqrt{L}}$$

使用该公式，我们能够看到，为了取得 10Gbps 的吞吐量，今天的 TCP 拥塞控制算法

仅能容忍 2×10^{-10} 的报文段丢失概率（或等价地说，对每 5 000 000 000 个报文段有一个丢包），这是一个非常低的值。这种观察导致许多研究人员为这种高速环境特别设计新版 TCP，对这些努力的讨论参见［Jin 2004；Kelly 2003；Ha 2008；RFC 7323］。

3.7.1 公平性

考虑 K 条 TCP 连接，每条都有不同的端到端路径，但是都经过一段传输速率为 R bps 的瓶颈链路。（所谓瓶颈链路，是指对于每条连接，沿着该连接路径上的所有其他段链路都不拥塞，而且与该瓶颈链路的传输容量相比，它们都有充足的传输容量。）假设每条连接都在传输一个大文件，而且无 UDP 流量通过该段瓶颈链路。如果每条连接的平均传输速率接近 R/K，即每条连接都得到相同份额的链路带宽，则认为该拥塞控制机制是公平的。

TCP 的 AIMD 算法公平吗？尤其是假定可在不同时间启动并因此在某个给定的时间点可能具有不同的窗口长度情况下，对这些不同的 TCP 连接还是公平的吗？TCP 趋于在竞争的多条 TCP 连接之间提供对一段瓶颈链路带宽的平等分享，其理由［Chiu 1989］给出了一个极好的、直观的解释。

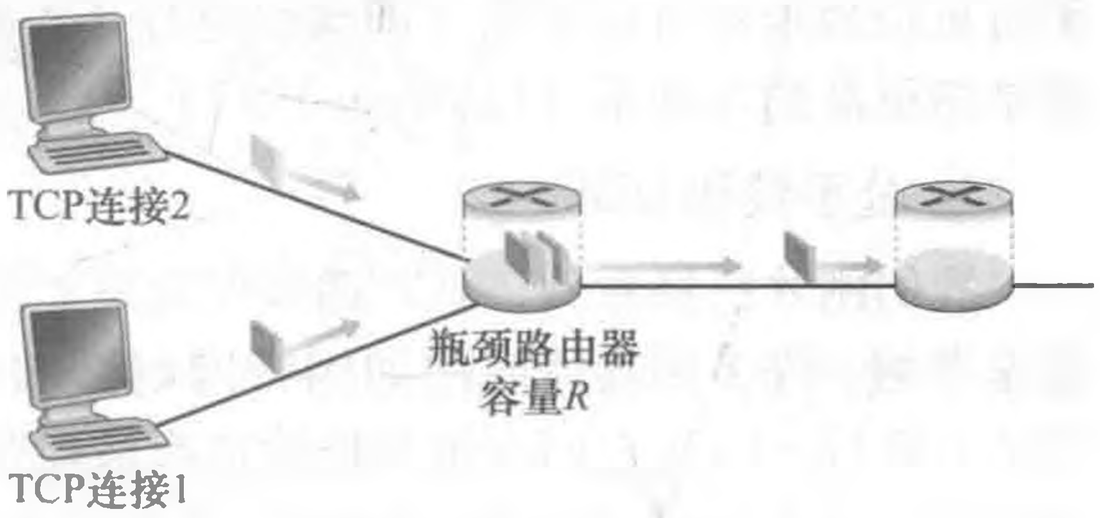

图 3-54 两条 TCP 连接共享同一条瓶颈链路

我们考虑有两条 TCP 连接共享一段传输速率为 R 的链路的简单例子，如图 3-54 中所示。我们将假设这两条连接有相同的 MSS 和 RTT（这样如果它们有相同的拥塞窗口长度，就会有相同的吞吐量），它们有大量的数据要发送，且没有其他 TCP 连接或 UDP 数据报穿越该段共享链路。我们还将忽略 TCP 的慢启动阶段，并假设 TCP 连接一直按 CA 模式（AIMD）运行。

图 3-55 描绘了两条 TCP 连接实现的吞吐量情况。如果 TCP 要在这两条 TCP 连接之间平等地共享链路带宽，那么实现的吞吐量曲线应当是从原点沿 45°方向的箭头向外辐射（平等带宽共享）。理想情况是，两个吞吐量的和应等于 R。（当然，每条连接得到相同但容量为 0 的共享链路容量并非我们所期望的情况！）所以我们的目标应该是使取得的吞吐量落在图 3-55 中平等带宽共享曲线与全带宽利用曲线的交叉点附近的某处。

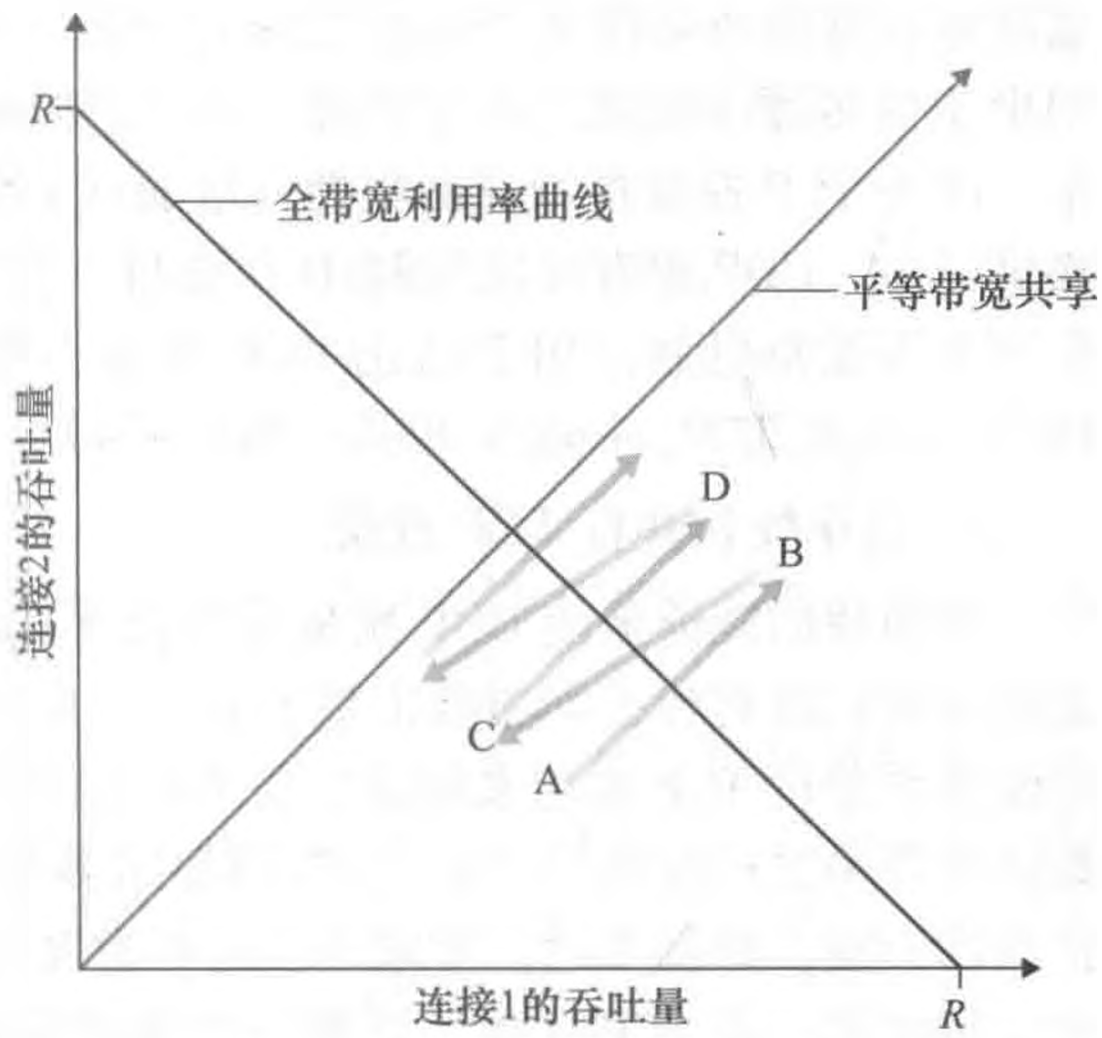

图 3-55 TCP 连接 1 和连接 2 实现的吞吐量

假定 TCP 窗口长度是这样的，即在某给定时刻，连接 1 和连接 2 实现了由图 3-55 中 A 点所指明的吞吐量。因为这两条连接共同消耗的链路带宽量小于 R，所以无丢包事件发生，根据 TCP 的拥塞避免算法的结果，这两条连接每过一个 RTT 都要将其窗口增加 1 个 MSS。因此，这两条连接的总吞吐量就会从 A 点开始沿 45°线前行（两条连接都有相同的增长）。最终，这两条连接共同消耗的带宽将超过 R，最终将发生分组丢失。假设连接 1

和连接2实现B点指明的吞吐量时，它们都经历了分组丢失。连接1和连接2于是就按二分之一减小其窗口。所产生的结果实现了C点指明的吞吐量，它正好位于始于B点止于原点的一个向量的中间。因为在C点，共同消耗的带宽小于R，所以这两条连接再次沿着始于C点的45°线增加其吞吐量。最终，再次发生丢包事件，如在D点，这两条连接再次将其窗口长度减半，如此等等。你应当搞清楚这两条连接实现的带宽最终将沿着平等带宽共享曲线在波动。还应该搞清楚无论这两条连接位于二维空间的何处，它们最终都会收敛到该状态！虽然此时我们做了许多理想化的假设，但是它仍然能对解释为什么TCP会导致在多条连接之间的平等共享带宽这个问题提供一个直观的感觉。

在理想化情形中，我们假设仅有TCP连接穿过瓶颈链路，所有的连接具有相同的RTT值，且对于一个主机－目的地对而言只有一条TCP连接与之相关联。实践中，这些条件通常是得不到满足的，客户－服务器应用因此能获得非常不平等的链路带宽份额。特别是，已经表明当多条连接共享一个共同的瓶颈链路时，那些具有较小RTT的连接能够在链路空闲时更快地抢到可用带宽（即较快地打开其拥塞窗口），因而将比那些具有较大RTT的连接享用更高的吞吐量［Laksman 1997］。

1. 公平性和UDP

我们刚才已经看到，TCP拥塞控制是如何通过拥塞窗口机制来调节一个应用程序的传输速率的。许多多媒体应用如因特网电话和视频会议，经常就因为这种特定原因而不在TCP上运行，因为它们不想其传输速率被扼制，即使在网络非常拥塞的情况下。相反，这些应用宁可在UDP上运行，UDP是没有内置的拥塞控制的。当运行在UDP上时，这些应用能够以恒定的速率将其音频和视频数据注入网络之中并且偶尔会丢失分组，而不愿在拥塞时将其发送速率降至“公平”级别并且不丢失任何分组。从TCP的观点来看，运行在UDP上的多媒体应用是不公平的，因为它们不与其他连接合作，也不适时地调整其传输速率。因为TCP拥塞控制在面临拥塞增加（丢包）时，将降低其传输速率，而UDP源则不必这样做，UDP源有可能压制TCP流量。当今的一个主要研究领域就是开发一种因特网中的拥塞控制机制，用于阻止UDP流量不断压制直至中断因特网吞吐量的情况［Floyd 1999；Floyd 2000；Kohler 2006；RFC 4340］。

2. 公平性和并行TCP连接

即使我们能够迫使UDP流量具有公平的行为，但公平性问题仍然没有完全解决。这是因为我们没有什么办法阻止基于TCP的应用使用多个并行连接。例如，Web浏览器通常使用多个并行TCP连接来传送一个Web页中的多个对象。（多条连接的确切数目可以在多数浏览器中进行配置。）当一个应用使用多条并行连接时，它占用了一条拥塞链路中较大比例的带宽。举例来说，考虑一段速率为R且支持9个在线客户－服务器应用的链路，每个应用使用一条TCP连接。如果一个新的应用加入进来，也使用一条TCP连接，则每个应用得到差不多相同的传输速率$R/10$。但是如果这个新的应用这次使用了11个并行TCP连接，则这个新应用就不公平地分到超过$R/2$的带宽。Web流量在因特网中是非常普遍的，所以多条并行连接并非不常见。

3.7.2　明确拥塞通告：网络辅助拥塞控制

自20世纪80年代后期慢启动和拥塞避免开始标准化以来［RFC 1122］，TCP已经实现了端到端拥塞控制的形式，我们在3.7.1节中对此进行了学习：一个TCP发送方不会收

到来自网络层的明确拥塞指示，而是通过观察分组丢失来推断拥塞。最近，对于 IP 和 TCP 的扩展方案［RFC 3168］已经提出并已经实现和部署，该方案允许网络明确向 TCP 发送方和接收方发出拥塞信号。这种形式的网络辅助拥塞控制称为**明确拥塞通告**（Explicit Congestion Notification，ECN）。如图 3-56 所示，涉及了 TCP 和 IP 协议。

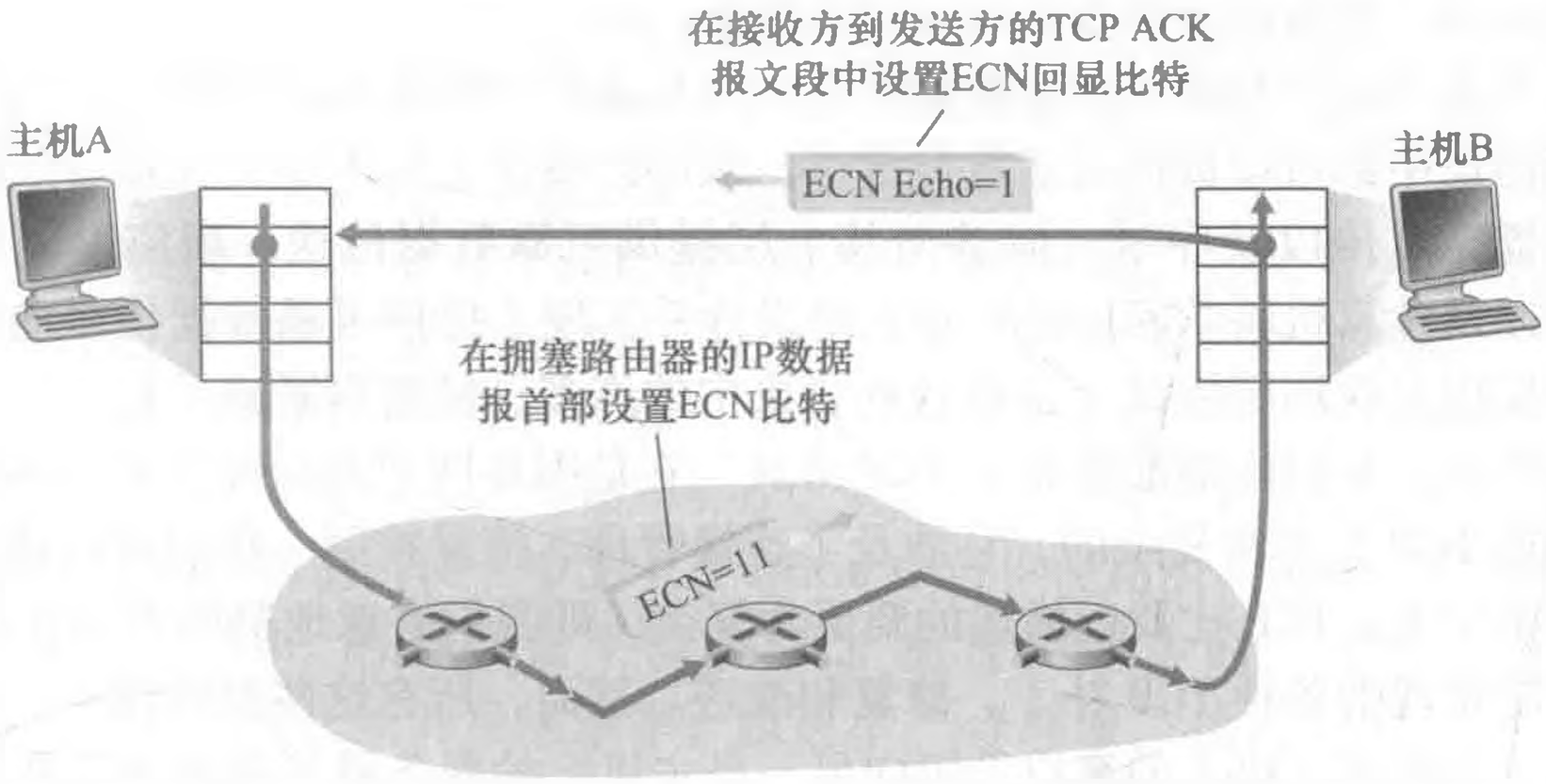

图 3-56 明确拥塞通告：网络辅助拥塞控制

在网络层，IP 数据报首部的服务类型字段中的两个比特（总的说来，有四种可能的值）被用于 ECN。路由器所使用的一种 ECN 比特设置指示该路由器正在历经拥塞。该拥塞指示则由被标记的 IP 数据报所携带，送给目的主机，再由目的主机通知发送主机，如图 3-56 所示。RFC 3168 没有提供路由器拥塞时的定义；该判断是由路由器厂商所做的配置选择，并且由网络操作员决定。然而，RFC 3168 推荐仅当拥塞持续不断存在时才设置 ECN 比特。发送主机所使用的另一种 ECN 比特设置通知路由器发送方和接收方是 ECN 使能的，因此能够对于 ECN 指示的网络拥塞采取行动。

如图 3-56 所示，当接收主机中的 TCP 通过一个接收到的数据报收到了一个 ECN 拥塞指示时，接收主机中的 TCP 通过在接收方到发送方的 TCP ACK 报文段中设置 ECE（明确拥塞通告回显）比特（参见图 3-29），通知发送主机中的 TCP 收到拥塞指示。接下来，TCP 发送方通过减半拥塞窗口对一个具有 ECE 拥塞指示的 ACK 做出反应，就像它对丢失报文段使用快速重传做出反应一样，并且在下一个传输的 TCP 发送方到接收方的报文段首部中对 CWR（拥塞窗口缩减）比特进行设置。

除了 TCP 以外的其他运输层协议也可以利用网络层发送 ECN 信号。数据报拥塞控制协议（Datagram Congestion Control Protocol，DCCP）［RFC 4340］提供了一种低开销、控制拥塞的类似 UDP 不可靠服务，该协议利用了 ECN。DCTCP（数据中心 TCP）［Alizadeh 2010］是一种专门为数据中心网络设计的 TCP 版本，也利用了 ECN。

3.8 小结

本章我们首先学习了运输层协议能够向网络应用程序提供的服务。在一个极端，运输层协议非常简单，并向应用程序不提供不必要的服务，而仅向通信进程提供多路复用/分解的功能。因特网中的 UDP 协议就是这样一种不提供不必要服务的运输层协议。在另一个极端，运输层协议能够向应用程序提供各种各样的保证，例如数据的可靠交付、时延保证和带宽保证。无论如何，运输层协议能够提供的服务经常受下面网络层协议服务模型的

限制。如果网络层协议不能向运输层报文段提供时延或带宽保证，那么运输层协议就不能向进程间发送的报文提供时延或带宽保证。

在 3.4 节中，我们学习了运输层协议能够提供可靠数据传输，即使下面的网络层是不可靠的。我们看到了提供可靠的数据传送会遇到许多微妙的问题，但都可以通过精心地结合确认、定时器、重传以及序号机制来完成任务。

尽管在本章中我们包含了可靠数据传送，但是我们应该理解在链路层、网络层、运输层或应用层协议中都可以提供可靠数据传送。该协议栈中上面 4 层的任意一层都可以实现确认、定时器、重传以及序号，能够向其上层提供可靠数据传送。事实上，在过去数年中，工程师以及计算机科学家们已经独立地设计并实现了提供可靠数据传送的链路层、网络层、运输层以及应用层协议（虽然这些协议中的许多已经销声匿迹了）。

在 3.5 节中，我们详细地研究了 TCP 协议，它是因特网中面向连接和可靠的运输层协议。我们知道 TCP 是非常复杂的，它涉及了连接管理、流量控制、往返时间估计以及可靠数据传送。事实上，TCP 比我们描述的要更为复杂，即我们有意地避而不谈在各种 TCP 实现版本中广泛实现的各种 TCP 补丁、修复和改进。然而，所有这些复杂性都对网络层应用隐藏了起来。如果某主机上的客户希望向另一台主机上的服务器可靠地发送数据，它只需要打开对该服务器的一个 TCP 套接字，然后将数据注入该套接字。客户 - 服务器应用程序则乐于对 TCP 的复杂性视而不见。

在 3.6 节中，我们从广泛的角度研究了拥塞控制，在 3.7 节中我们阐述了 TCP 是如何实现拥塞控制的。我们知道了拥塞控制对于网络良好运行是必不可少的。没有拥塞控制，网络很容易出现死锁，使得端到端之间很少或没有数据能被传输。在 3.7 节中我们学习了 TCP 实现的一种端到端拥塞控制机制，即当 TCP 连接的路径上判断不拥塞时，其传输速率就加性增；当出现丢包时，传输速率就乘性减。这种机制也致力于做到每一个通过拥塞链路的 TCP 连接能平等地共享该链路带宽。我们也深入探讨了 TCP 连接建立和慢启动对时延的影响。我们观察到在许多重要场合，连接建立和慢启动会对端到端时延产生严重影响。我们再次强调，尽管 TCP 在这几年一直在发展，但它仍然是一个值得深入研究的领域，并且在未来的几年中还可能持续演化。

在本章中我们对特定因特网运输协议的讨论集中在 UDP 和 TCP 上，它们是因特网运输层的两匹“驮马”。然而，对这两个协议的二十多年的经验已经使人们认识到，这两个协议都不是完美无缺的。研究人员因此在忙于研制其他的运输层协议，其中的几种现在已经成为 IETF 建议的标准。

数据报拥塞控制协议（Datagram Congestion Control Protocol，DCCP）［RFC 4340］提供了一种低开销、面向报文、类似于 UDP 的不可靠服务，但是具有应用程序可选择的拥塞控制形式，该机制与 TCP 相兼容。如果某应用程序需要可靠的或半可靠的数据传送，则这将在应用程序自身中执行（也许使用我们已经在 3.4 节中学过的机制）。DCCP 被设想用于诸如流媒体（参见第 9 章）等应用程序中，DCCP 能够利用数据交付的预定时间和可靠性之间的折中，但是要对网络拥塞做出响应。

在谷歌的 Chromium 浏览器中实现了 QUIC（Quick UDP Internet Connections）协议［Iyengar 2016］，该协议通过重传以及差错检测、快速连接建立和基于速率的拥塞控制算法提供可靠性，而基于速率的拥塞控制算法是以 TCP 友好特性为目标，这些机制都是在 UDP 之上作为应用层协议实现的。2015 年年初，谷歌报告从 Chrome 浏览器到谷歌服务器的大约一半请求运行在 QUIC 之上。

DCTCP（数据中心 TCP）［Alizadeh 2010］是一种专门为数据中心网络设计的 TCP 版本，使用 ECN 以更好地支持短流和长流的混合流，这种混合流代表了数据中心负载的特征。

流控制传输协议（Stream Control Transmission Protocol，SCTP)［RFC 4960，RFC 3286］是一种可靠的、面向报文的协议，该协议允许几个不同的应用层次的“流”复用到单个 SCTP 连接上（一种称之为“多流”的方法）。从可靠性的角度看，在该连接中的不同流被分别处理，因此在一条流中的分组丢失不会影响其他流中数据的交付。当一台主机与两个或更多个网络连接时，SCTP 也允许数据经两条出路径传输，还具有失序数据的选项交付和一些其他特色。SCTP 的流控制和拥塞控制算法基本上与 TCP 中的相同。

TCP 友好速率控制（TCP-Friendly Rate Control，TFRC）协议［RFC 5348］是一种拥塞控制协议而不是一种功能齐全的运输层协议。它定义了一种拥塞控制机制，该机制能被用于诸如 DCCP 等其他运输协议（事实上在 DCCP 中可供使用的两种应用程序可选的协议之一就是 TFRC）。TFRC 的目标是平滑在 TCP 拥塞控制中的“锯齿”行为（参见图 3-53），同时维护一种长期的发送速率，该速率“合理地”接近 TCP 的速率。使用比 TCP 更为平滑的发送速率，TFRC 非常适合诸如 IP 电话或流媒体等多媒体应用，这种平滑的速率对于这些应用是重要的。TFRC 是一种“基于方程”的协议，这些协议使用测得的丢包率作为方程的输入［Padhye 2000］，即使用方程估计一个 TCP 会话在该丢包率下 TCP 的吞吐量将是多大。该速率则被取为 TFRC 的目标发送速率。

唯有未来才能告诉我们 DCCP、SCTP、QUIC 或 TFRC 是否能得到广泛实施。虽然这些协议明确地提供了超过 TCP 和 UDP 的强化能力，但是多年来已经证明了 TCP 和 UDP 自身是“足够好”的。是否“更好”将胜出“足够好”，这将取决于技术、社会和商业考虑的复杂组合。

在第 1 章中，我们讲到计算机网络能被划分成“网络边缘”和“网络核心”。网络边缘包含了在端系统中发生的所有事情。既然已经覆盖了应用层和运输层，我们关于网络边缘的讨论也就结束了。接下来是探寻网络核心的时候了！我们的旅程从下一章开始，下一章将学习网络层，并且将在第 6 章继续学习链路层。

课后习题和问题

复习题

3.1～3.3 节

R1. 假定网络层提供了下列服务。在源主机中的网络层接受最大长度 1200 字节和来自运输层的目的主机地址的报文段。网络层则保证将该报文段交付给位于目的主机的运输层。假定在目的主机上能够运行许多网络应用进程。

a. 设计可能最简单的运输层协议，该协议将使应用程序数据到达位于目的主机的所希望的进程。假设在目的主机中的操作系统已经为每个运行的应用进程分配了一个 4 字节的端口号。

b. 修改这个协议，使它向目的进程提供一个的“返回地址”。

c. 在你的协议中，该运输层在计算机网络的核心中“必须做任何事”吗？

R2. 考虑有一个星球，每个人都属于某个六口之家，每个家庭都住在自己的房子里，每个房子都一个唯一的地址，并且某给定家庭中的每个人有一个独特的名字。假定该星球有一个从源家庭到目的家庭交付信件的邮政服务。该邮件服务要求：①在一个信封中有一封信；②在信封上清楚地写上目的家

庭的地址（并且没有别的东西）。假设每个家庭有一名家庭成员代表为家庭中的其他成员收集和分发信件。这些信没有必要提供任何有关信的接收者的指示。

a. 使用对上面复习题 R1 的解决方案作为启发，描述家庭成员代表能够使用的协议，以从发送家庭成员向接收家庭成员交付信件。

b. 在你的协议中，该邮政服务必须打开信封并检查信件内容才能提供它的服务吗？

R3. 考虑在主机 A 和主机 B 之间有一条 TCP 连接。假设从主机 A 传送到主机 B 的 TCP 报文段具有源端口号 x 和目的端口号 y。对于从主机 B 传送到主机 A 的报文段，源端口号和目的端口号分别是多少？

R4. 描述应用程序开发者为什么可能选择在 UDP 上运行应用程序而不是在 TCP 上运行的原因。

R5. 在今天的因特网中，为什么语音和图像流量常常是经过 TCP 而不是经 UDP 发送。（提示：我们寻找的答案与 TCP 的拥塞控制机制没有关系。）

R6. 当某应用程序运行在 UDP 上时，该应用程序可能得到可靠数据传输吗？如果能，如何实现？

R7. 假定在主机 C 上的一个进程有一个具有端口号 6789 的 UDP 套接字。假定主机 A 和主机 B 都用目的端口号 6789 向主机 C 发送一个 UDP 报文段。这两台主机的这些报文段在主机 C 都被描述为相同的套接字吗？如果是这样的话，在主机 C 的该进程将怎样知道源于两台不同主机的这两个报文段？

R8. 假定在主机 C 端口 80 上运行的一个 Web 服务器。假定这个 Web 服务器使用持续连接，并且正在接收来自两台不同主机 A 和 B 的请求。被发送的所有请求都通过位于主机 C 的相同套接字吗？如果它们通过不同的套接字传递，这两个套接字都具有端口 80 吗？讨论和解释之。

3.4 节

R9. 在我们的 rdt 协议中，为什么需要引入序号？

R10. 在我们的 rdt 协议中，为什么需要引入定时器？

R11. 假定发送方和接收方之间的往返时延是固定的并且为发送方所知。假设分组能够丢失的话，在协议 rdt3.0 中，一个定时器仍是必需的吗？试解释之。

R12. 在配套网站上使用 Go-Back-N（回退 N 步）Java 小程序。

a. 让源发送 5 个分组，在这 5 个分组的任何一个到达目的地之前暂停该动画。然后毁掉第一个分组并继续该动画。试描述发生的情况。

b. 重复该实验，只是现在让第一个分组到达目的地并毁掉第一个确认。再次描述发生的情况。

c. 最后，尝试发送 6 个分组。发生了什么情况？

R13. 重复复习题 R12，但是现在使用 Selective Repeat（选择重传）Java 小程序。选择重传和回退 N 步有什么不同？

3.5 节

R14. 是非判断题：

a. 主机 A 经过一条 TCP 连接向主机 B 发送一个大文件。假设主机 B 没有数据发往主机 A。因为主机 B 不能随数据捎带确认，所以主机 B 将不向主机 A 发送确认。

b. 在连接的整个过程中，TCP 的 rwnd 的长度决不会变化。

c. 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 A 发送但未被确认的字节数不会超过接收缓存的大小。

d. 假设主机 A 通过一条 TCP 连接向主机 B 发送一个大文件。如果对于这条连接的一个报文段的序号为 m，则对于后继报文段的序号将必然是 $m+1$。

e. TCP 报文段在它的首部中有一个 rwnd 字段。

f. 假定在一条 TCP 连接中最后的 SampleRTT 等于 1 秒，那么对于该连接的 TimeoutInterval 的当前值必定大于等于 1 秒。

g. 假设主机 A 通过一条 TCP 连接向主机 B 发送一个序号为 38 的 4 个字节的报文段。在这个相同的报文段中，确认号必定是 42。

R15. 假设主机 A 通过一条 TCP 连接向主机 B 发送两个紧接着的 TCP 报文段。第一个报文段的序号为 90，第二个报文段序号为 110。

a. 第一个报文段中有多少数据?

b. 假设第一个报文段丢失而第二个报文段到达主机 B。那么在主机 B 发往主机 A 的确认报文中，确认号应该是多少?

R16. 考虑在 3.5 节中讨论的 Telnet 的例子。在用户键入字符 C 数秒之后，用户又键入字符 R。那么在用户键入字符 R 之后，总共发送了多少个报文段，这些报文段中的序号和确认字段应该填入什么?

3.7 节

R17. 假设两条 TCP 连接存在于一个带宽为 R bps 的瓶颈链路上。它们都要发送一个很大的文件（以相同方向经过瓶颈链路），并且两者是同时开始发送文件。那么 TCP 将为每条连接分配什么样的传输速率?

R18. 是非判断题。考虑 TCP 的拥塞控制。当发送方定时器超时时，其 ssthresh 的值将被设置为原来值的一半。

R19. 在 3.7 节的“TCP 分岔”讨论中，对于 TCP 分岔的响应时间，断言大约是 $4 * RTT_{FE} + RTT_{BE}$ + 处理时间。评价该断言。

习题

P1. 假设客户 A 向服务器 S 发起一个 Telnet 会话。与此同时，客户 B 也向服务器 S 发起一个 Telnet 会话。给出下面报文段的源端口号和目的端口号:

a. 从 A 向 S 发送的报文段。

b. 从 B 向 S 发送的报文段。

c. 从 S 向 A 发送的报文段。

d. 从 S 向 B 发送的报文段。

e. 如果 A 和 B 是不同的主机，那么从 A 向 S 发送的报文段的源端口号是否可能与从 B 向 S 发送的报文段的源端口号相同?

f. 如果它们是同一台主机，情况会怎么样?

P2. 考虑图 3-5。从服务器返回客户进程的报文流中的源端口号和目的端口号是多少? 在承载运输层报文段的网络层数据报中，IP 地址是多少?

P3. UDP 和 TCP 使用反码来计算它们的检验和。假设你有下面 3 个 8 比特字节：01010011，01100110，01110100。这些 8 比特字节和的反码是多少?（注意到尽管 UDP 和 TCP 使用 16 比特的字来计算检验和，但对于这个问题，你应该考虑 8 比特和。）写出所有工作过程。UDP 为什么要用该和的反码，即为什么不直接使用该和呢? 使用该反码方案，接收方如何检测出差错? 1 比特的差错将可能检测不出来吗? 2 比特的差错呢?

P4. a. 假定你有下列 2 个字节：01011100 和 01100101。这 2 个字节之和的反码是什么?

b. 假定你有下列 2 个字节：11011010 和 01100101。这 2 个字节之和的反码是什么?

c. 对于（a）中的字节，给出一个例子，使得这 2 个字节中的每一个都在一个比特反转时，其反码不会改变。

P5. 假定某 UDP 接收方对接收到的 UDP 报文段计算因特网检验和，并发现它与承载在检验和字段中的值相匹配。该接收方能够绝对确信没有出现过比特差错吗? 试解释之。

P6. 考虑我们改正协议 rdt2.1 的动机。试说明如图 3-57 所示的接收方与如图 3-11 所示的发送方运行时，接收方可能会引起发送方和接收方进入死锁状态，即双方都在等待不可能发生的事件。

P7. 在 rdt3.0 协议中，从接收方向发送方流动的 ACK 分组没有序号（尽管它们具有 ACK 字段，该字段包括了它们正在确认的分组的序号）。为什么这些 ACK 分组不需要序号呢?

P8. 画出协议 rdt3.0 中接收方的 FSM。

P9. 当数据分组和确认分组发生篡改时，给出 rdt3.0 协议运行的轨迹。你画的轨迹应当类似于图 3-16 中所用的图。

P10. 考虑一个能够丢失分组但其最大时延已知的信道。修改协议 rdt2.1，以包括发送方超时和重传机制。

非正式地论证：为什么你的协议能够通过该信道正确通信？

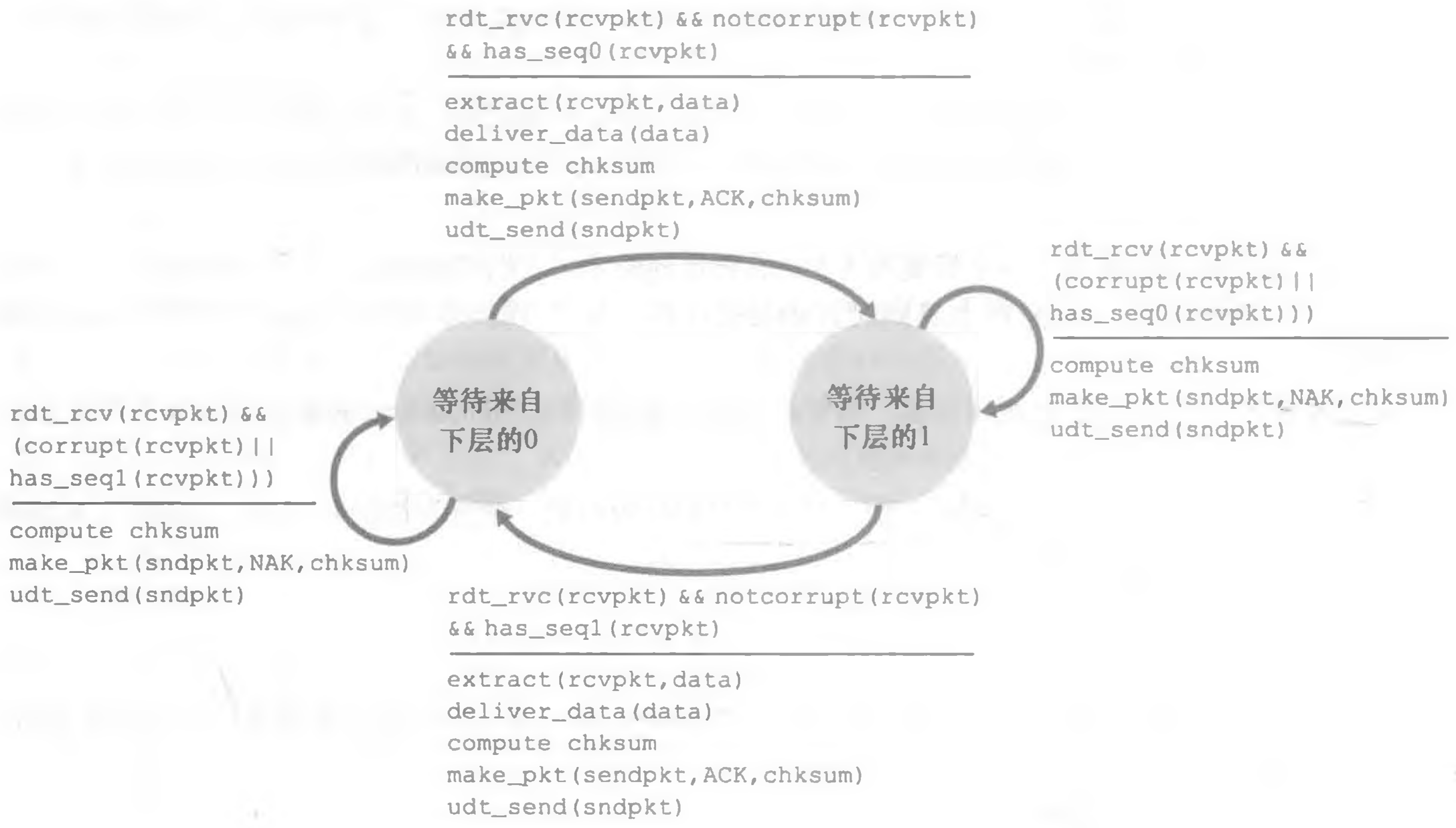

图 3-57 协议 rdt2.1 的一个不正确的接收方

P11. 考虑在图 3-14 中的 rdt2.2 接收方，在状态“等待来自下层的 0”和状态“等待来自下层的 1”中的自转换（即从某状态转换回自身）中生成一个新分组：sndpk = make_ pkt（ACK，1，checksum）和 sndpk = make_ pkt（ACK，0，checksum）。如果这个动作从状态“等待来自下层的 1”中的自转换中删除，该协议将正确工作吗？评估你的答案。在状态“等待来自下层的 0”中的自转换中删除这个事件将会怎样？[提示：在后一种情况下，考虑如果第一个发送方到接收方的分组损坏的话，将会发生什么情况？]

P12. rdt3.0 协议的发送方直接忽略（即不采取任何动作）接收到的所有出现差错和确认分组的确认号（acknum）字段中的值有差错的分组。假设在这种情况下，rdt3.0 只是重传当前的数据分组，该协议是否还能正常运行？（提示：考虑在下列情况下会发生什么情况：仅有一个比特差错时；报文没有丢失但能出现定时器过早超时。考虑到当 n 趋于无穷时，第 n 个分组将被发送多少次。）

P13. 考虑 rdt3.0 协议。如果发送方和接收方的网络连接能够对报文重排序（即在发送方和接收方之间的媒体上传播的两个报文段能重新排序），那么比特交替协议将不能正确工作（确信你清楚地理解这时它不能正确工作的原因），试画图说明之。画图时把发送方放在左边，接收方放在右边，使时间轴朝下，标出交换的数据报文（D）和确认报文（A）。要标明与任何数据和确认报文段相关的序号。

P14. 考虑一种仅使用否定确认的可靠数据传输协议。假定发送方只是偶尔发送数据。只用 NAK 的协议是否会比使用 ACK 的协议更好？为什么？现在我们假设发送方要发送大量的数据，并且该端到端连接很少丢包。在第二种情况下，只用 NAK 的协议是否会比使用 ACK 的协议更好？为什么？

P15. 考虑显示在图 3-17 中的网络跨越国家的例子。窗口长度设置成多少时，才能使该信道的利用率超过 90%？假设分组的长度为 1500 字节（包括首部字段和数据）。

P16. 假设某应用使用 rdt3.0 作为其运输层协议。因为停等协议具有非常低的信道利用率（显示在网络跨越国家的例子中），该应用程序的设计者让接收方持续回送许多（大于 2）交替的 ACK 0 和 ACK 1，即使对应的数据未到达接收方。这个应用程序设计将能增加信道利用率吗？为什么？该方法存在某种潜在的问题吗？试解释之。

P17. 考虑两个网络实体 A 和 B，它们由一条完善的双向信道所连接（即任何发送的报文将正确地收到；

信道将不会损坏、丢失或重排序分组）。A 和 B 将以交互的方式彼此交付报文：首先，A 必须向 B 交付一个报文，B 然后必须向 A 交付一个报文，接下来 A 必须向 B 交付一个报文，等等。如果一个实体处于它不试图向另一侧交付报文的状态，将存在一个来自上层的类似于 rdt_send(data)调用的事件，它试图向下传送数据以向另一侧传输，来自上层的该调用能够直接忽略对于 rdt_unable_to_send(data)调用，这通知较高层当前不能够发送数据。[注意：做出这种简化的假设，使你不必担心缓存数据。]

对该协议画出 FSM 说明（一个 FSM 用于 A，一个 FSM 用于 B）。注意你不必担心这里的可靠性机制，该问题的要点在于创建反映这两个实体的同步行为的 FSM 说明。应当使用与图 3-9 中协议 rdt1.0 有相同含义的下列事件和动作：rdt_send(data)，packet = make_pkt(data)，udt__send(data)，rdt_rcv(packet)，extract(packet,data)，deliver_data(data)。保证你的协议反映了 A 和 B 之间发送的严格交替。还要保证在你的 FSM 描述中指出 A 和 B 的初始状态。

P18. 在 3.4.4 节我们学习的一般性 SR 协议中，只要报文可用（如果报文在窗口中），发送方就会不等待确认而传输报文。假设现在我们要求一个 SR 协议，一次发出一对报文，而且只有在知道第一对报文中的两个报文都正确到达后才发送第二对报文。

假设该信道中可能会丢失报文，但报文不会发生损坏和失序。试为报文的单向可靠传输而设计一个差错控制协议。画出发送方和接收方的 FSM 描述。描述在发送方和接收方之间两个方向发送的报文格式。如果你使用了不同于 3.4 节（例如 udt_send()、start_timer()、rdt_rcv()等）中的任何其他过程调用，详细地阐述这些动作。举例说明（用发送方和接收方的时序踪迹图）你的协议是如何恢复报文丢失的。

P19. 考虑一种情况，主机 A 想同时向主机 B 和主机 C 发送分组。A 与 B 和 C 是经过广播信道连接的，即由 A 发送的分组通过该信道传送到 B 和 C。假设连接 A、B 和 C 的这个广播信道具有独立的报文丢失和损坏特性（例如，从 A 发出的报文可能被 B 正确接收，但没有被 C 正确接收）。设计一个类似于停等协议的差错控制协议，用于从 A 可靠地传输分组到 B 和 C。该协议使得 A 直到得知 B 和 C 已经正确接收到当前报文，才获取上层交付的新数据。给出 A 和 C 的 FSM 描述。（提示：B 的 FSM 大体上应当与 C 的相同。）同时，给出所使用的报文格式的描述。

P20. 考虑一种主机 A 和主机 B 要向主机 C 发送报文的情况。主机 A 和 C 通过一条报文能够丢失和损坏（但不重排序）的信道相连接。主机 B 和 C 由另一条（与连接 A 和 C 的信道独立）具有相同性质的信道连接。在主机 C 上的运输层，在向上层交付来自主机 A 和 B 的报文时应当交替进行（即它应当首先交付来自 A 的分组中的数据，然后是来自 B 的分组中的数据，等等）。设计一个类似于停等协议的差错控制协议，以可靠地向 C 传输来自 A 和 B 的分组，同时以前面描述的方式在 C 处交替地交付。给出 A 和 C 的 FSM 描述。（提示：B 的 FSM 大体上应当与 A 的相同。）同时，给出所使用的报文格式的描述。

P21. 假定我们有两个网络实体 A 和 B。B 有一些数据报文要通过下列规则传给 A。当 A 从其上层得到一个请求，就从 B 获取下一个数据（D）报文。A 必须通过 A—B 信道向 B 发送一个请求（R）报文。仅当 B 收到一个 R 报文后，它才会通过 B—A 信道向 A 发送一个数据（D）报文。A 应当准确地将每份 D 报文的副本交付给上层。R 报文可能会在 A—B 信道中丢失（但不会损坏）；D 报文一旦发出总是能够正确交付。两个信道的时延未知且是变化的。

设计一个协议（给出 FSM 描述），它能够综合适当的机制，以补偿会丢包的 A—B 信道，并且实现在 A 实体中向上层传递报文。只采用绝对必要的机制。

P22. 考虑一个 GBN 协议，其发送方窗口为 4，序号范围为 1024。假设在时刻 t，接收方期待的下一个有序分组的序号是 k。假设媒体不会对报文重新排序。回答以下问题：

a. 在 t 时刻，发送方窗口内的报文序号可能是多少？论证你的回答。

b. 在 t 时刻，在当前传播回发送方的所有可能报文中，ACK 字段的所有可能值是多少？论证你的回答。

P23. 考虑 GBN 协议和 SR 协议。假设序号空间的长度为 k，那么为了避免出现图 3-27 中的问题，对于这

两种协议中的每一种，允许的发送方窗口最大为多少？

P24. 对下面的问题判断是非，并简要地证实你的回答：

a. 对于 SR 协议，发送方可能会收到落在其当前窗口之外的分组的 ACK。

b. 对于 GBN 协议，发送方可能会收到落在其当前窗口之外的分组的 ACK。

c. 当发送方和接收方窗口长度都为 1 时，比特交替协议与 SR 协议相同。

d. 当发送方和接收方窗口长度都为 1 时，比特交替协议与 GBN 协议相同。

P25. 我们曾经说过，应用程序可能选择 UDP 作为运输协议，因为 UDP 提供了（比 TCP）更好的应用层控制，以决定在报文段中发送什么数据和发送时机。

a. 应用程序为什么对在报文段中发送什么数据有更多的控制？

b. 应用程序为什么对何时发送报文段有更多的控制？

P26. 考虑从主机 A 向主机 B 传输 L 字节的大文件，假设 MSS 为 536 字节。

a. 为了使得 TCP 序号不至于用完，L 的最大值是多少？前面讲过 TCP 的序号字段为 4 字节。

b. 对于你在（a）中得到的 L，求出传输此文件要用多长时间？假定运输层、网络层和数据链路层首部总共为 66 字节，并加在每个报文段上，然后经 155Mbps 链路发送得到的分组。忽略流量控制和拥塞控制，使主机 A 能够一个接一个和连续不断地发送这些报文段。

P27. 主机 A 和 B 经一条 TCP 连接通信，并且主机 B 已经收到了来自 A 的最长为 126 字节的所有字节。假定主机 A 随后向主机 B 发送两个紧接着的报文段。第一个和第二个报文段分别包含了 80 字节和 40 字节的数据。在第一个报文段中，序号是 127，源端口号是 302，目的地端口号是 80。无论何时主机 B 接收到来自主机 A 的报文段，它都会发送确认。

a. 在从主机 A 发往 B 的第二个报文段中，序号、源端口号和目的端口号各是什么？

b. 如果第一个报文段在第二个报文段之前到达，在第一个到达报文段的确认中，确认号、源端口号和目的端口号各是什么？

c. 如果第二个报文段在第一个报文段之前到达，在第一个到达报文段的确认中，确认号是什么？

d. 假定由 A 发送的两个报文段按序到达 B。第一个确认丢失了而第二个确认在第一个超时间隔之后到达。画出时序图，显示这些报文段和发送的所有其他报文段和确认。（假设没有其他分组丢失。）对于图上每个报文段，标出序号和数据的字节数量；对于你增加的每个应答，标出确认号。

P28. 主机 A 和 B 直接经一条 100Mbps 链路连接。在这两台主机之间有一条 TCP 连接。主机 A 经这条连接向主机 B 发送一个大文件。主机 A 能够向它的 TCP 套接字以高达 120Mbps 的速率发送应用数据，而主机 B 能够以最大 50Mbps 的速率从它的 TCP 接收缓存中读出数据。描述 TCP 流量控制的影响。

P29. 在 3.5.6 节中讨论了 SYN cookie。

a. 服务器在 SYNACK 中使用一个特殊的初始序号，这为什么是必要的？

b. 假定某攻击者得知了一台目标主机使用了 SYN cookie。该攻击者能够通过直接向目标发送一个 ACK 分组创建半开或全开连接吗？为什么？

c. 假设某攻击者收集了由服务器发送的大量初始序号。该攻击者通过发送具有初始序号的 ACK，能够引起服务器产生许多全开连接吗？为什么？

P30. 考虑在 3.6.1 节中显示在第二种情况下的网络。假设发送主机 A 和 B 具有某些固定的超时值。

a. 证明增加路由器有限缓存的长度可能减小吞吐量（λout）。

b. 现在假设两台主机基于路由器的缓存时延，动态地调整它们的超时值（像 TCP 所做的那样）。增加缓存长度将有助于增加吞吐量吗？为什么？

P31. 假设测量的 5 个 SampleRTT 值（参见 3.5.3 节）是 106ms、120ms、140ms、90ms 和 115ms。在获得了每个 SampleRTT 值后计算 EstimatedRTT，使用 $\alpha = 0.125$ 并且假设在刚获得前 5 个样本之后 EstimatedRTT 的值为 100ms。在获得每个样本之后，也计算 DevRTT，假设 $\beta = 0.25$，并且假设在刚获得前 5 个样本之后 DevRTT 的值为 5ms。最后，在获得这些样本之后计算 TCP TimeoutInterval。

P32. 考虑 TCP 估计 RTT 的过程。假设 $\alpha = 0.1$，令 SampleRT_{T1} 设置为最新样本 RTT，令 SampleRT_{T2} 设置为下一个最新样本 RTT，等等。