了相应的改进。接下来，本书也和前面一样，以图示的形式展示NewReno算法的流程，请看图3.18。

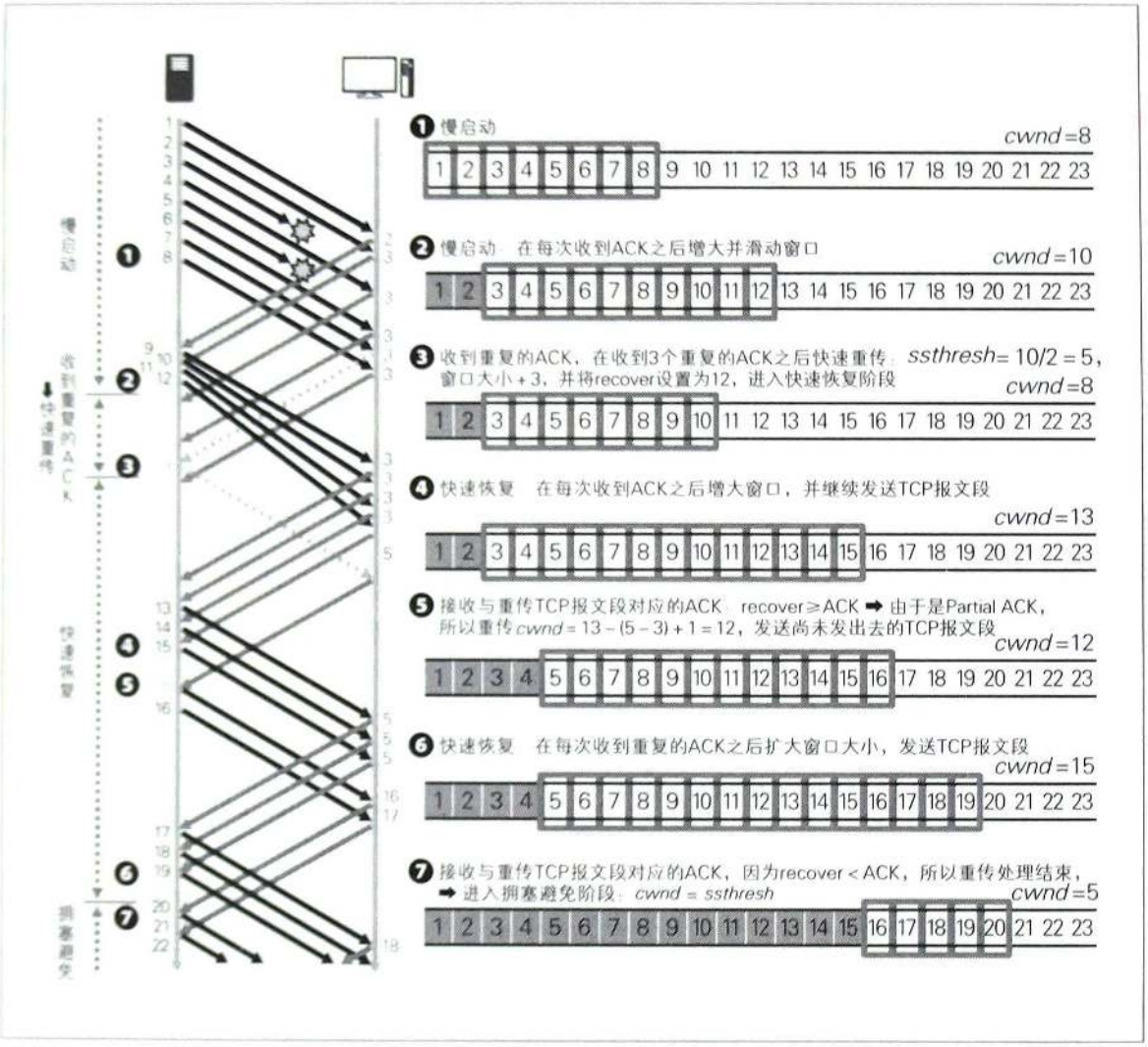

图3.18 NewReno的流程示例

假设当*cwnd*的值达到8时，丢失了2个TCP报文段（3号和4号）（❶）。由于接收方已经正确地收到了1号和2号的TCP报文段，因此发送方根据返回的ACK数据，将窗口大小扩大2格，同时滑动窗口，发送全新的4个TCP报文段（❷）。接下来，和图3.17中的情况一样，发送方在收到3号TCP报文段的重复ACK之后，进行快速重传（❸），将*ssthresh*设置为5，将*cwnd*设置为8。此时，引入新的参数*recover*。*recover*此时记录的是当前已发送的最大序列号，也就是说，*recover*=12。

随后，进入快速恢复阶段，发送方即使收到重复ACK，也仍然增大

窗口大小，并同时发送尚未发送的数据（❹）。接下来，它收到与重传 TCP 报文段（3 号）对应的 ACK（❺）。ACK 请求的是之后丢失的 5 号 TCP 报文段。发送方比较刚才记录下来的 *recover* 的值（=12）与 ACK 所请求的序列号（=5）。显然，从时间点❺的 recover ≥ ACK 可以看出来，接收方并非在请求尚未发送的 TCP 报文段[①]。于是，发送方就不再等待 5 号报文段的重复 ACK，而直接重传 5 号 TCP 报文段（❺），并继续遵循快速恢复算法运行。发送方在收到 5 号报文段的对应重复 ACK 之后增大 *cwnd*，同时继续发送尚未发送的 TCP 报文段（❻）。接下来，当 5 号 TCP 报文段成功发送之后，接收方就会返回请求新 TCP 报文段（也就是 16 号报文段）的 ACK。

当收到这个请求时，发送方会再次将 *recover* 的值（=12）与 ACK 请求的序列号（=16 号）进行比较。此时，recover < ACK，也就是说接收方在请求尚未发送的 TCP 报文段，因此发送方就会认为重传已经完成，并进入拥塞避免阶段（❼）。

如上例所述，NewReno 引入了新的参数 *recover*，通过将 *recover* 与重传报文段对应的 ACK 请求序列号相比较，便可以判断出下个丢失的 TCP 报文段。利用此方法，快速重传阶段的重传效率会进一步改善。

Vegas 基于延迟的控制方法的出现

与以往的基于丢包的控制方法不同，Vegas 是**基于延迟**的控制方法，使用了 *RTT* 进行窗口控制。Vegas 基于 *RTT* 预测网络的拥堵情况（以吞吐量为指标），并根据其变化情况调整传输量。因此，丢包量大幅降低，实现了在保持稳定运行前提下的高吞吐量。

Vegas 的拥塞窗口大小的更新公式如下所示。

① 此时，如果是 Reno 算法，则会在完成重传之后将窗口大小减小到 *ssthresh*。由于控制了数据的发送量，所以很有可能导致请求丢失的 5 号报文段的重复 ACK 的发送量大幅度减少。在图 3.18 的例子中，发送方在时间点❻虽然会因为收到 3 次重复 ACK 进行快速重传，但是会花费不少时间。总的来说，以往的 Reno 算法，其第一个丢失的 TCP 报文段对应的重复 ACK 占据主导地位，这样会导致之后丢失的其他报文段所对应的 ACK 很难发送出来。此类 ACK 被称为 Partial ACK（局部 ACK）。

$$cwnd = \begin{cases} cwnd + 1 & (\text{当} Diff < \alpha_v \text{时}) \\ cwnd - 1 & (\text{当} Diff < \beta_v \text{时}) \\ cwnd & (\text{其他}) \end{cases}$$

公式中的 α_v 与 β_v 表示发送缓冲区内保存的TCP报文段的最小值和最大值，控制思路主要是通过将 *Diff* 保持在 α_v 与 β_v 之间来调整 *cwnd*。Vegas的算法依赖于这两个阈值的设置，一般来说，$\alpha_v=1$，$\beta_v=3$。此外，*Diff* 是基于 *RTT* 计算出来的，代表传输数据量，其计算方式如下。

$$Diff = \left(\frac{cwnd}{RTT_{\min}} - \frac{cwnd}{RTT} \right) RTT_{\min}$$

RTT_{min} 代表获取到的 *RTT* 的最小值，而 *RTT* 是最新的实时 *RTT* 值。*Diff* 表示的是最大期望吞吐量（expected throughput）$\frac{cwnd}{RTT_{\min}}$ 与实际的吞吐量（actual throughput）$\frac{cwnd}{RTT}$ 之间的差值。当实际吞吐量比较小时，*Diff* 值就比较大，此时就需要将 *cwnd* 减小。而实际吞吐量增大后，需要将 *cwnd* 增大。

Vegas的拥塞窗口控制概念如图3.19所示。其中，横轴是拥塞窗口大小，纵轴是当时的对应吞吐量。因此，期望吞吐量随着窗口大小增加，按比例增长，而根据此期望吞吐量，便可以确定基于某一刻的 *cwnd* 得到的实际吞吐量。通过将实际吞吐量的值稳定地控制在两个阈值 $\frac{\alpha_v}{RTT}$ 与 $\frac{\beta v}{RTT}$ 之间，便可以完成下一个 *cwnd* 值的调整。

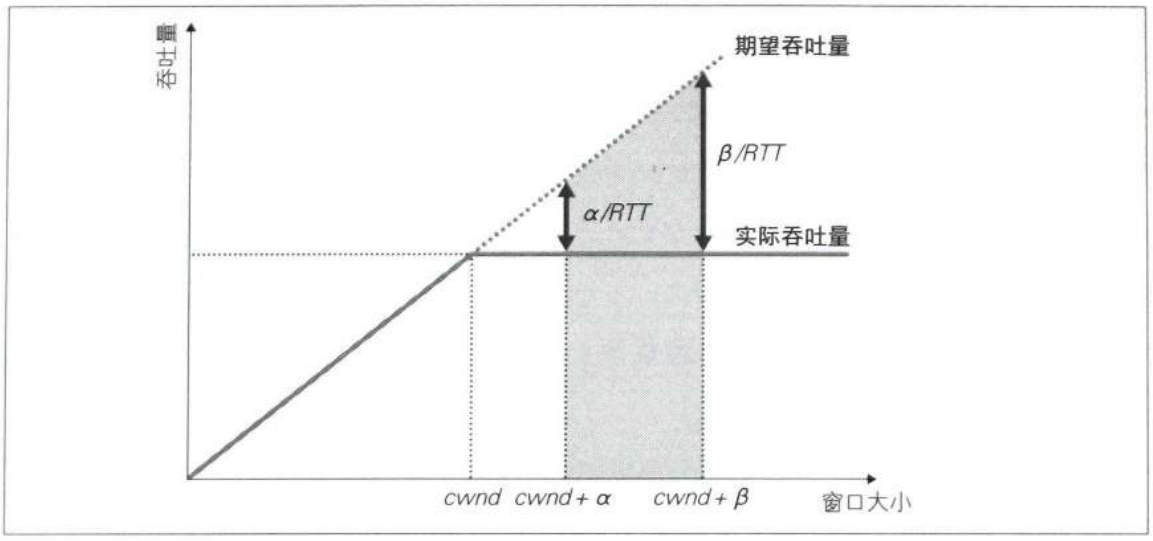

图 3.19 Vegas 中的拥塞窗口变化

3.7 小结

本章介绍了 TCP 的基本功能。我们从这些功能的出现背景中可以很明显地看出，为了能最大限度地有效利用网络通信资源，掌控一片混沌的网络状况，相关研究者必定是花了相当多的功夫才完成了这些功能的设计。

在前文所述的算法中，既有像从 Tahoe 到 Reno、NewReno 这样，发现已具备的功能中的缺陷并增加了新的算法，从而得以进化的例子，也有像 Vegas 这样，因为有了其他的思路而更新功能的例子。此外，也有针对安全性方面的弱点进行改进的例子。网络每时每刻都在变化，或者说都在进化。为了跟上网络进化的脚步，或者说为了进一步地提高效率，人们基于本章所介绍的算法，相继提出了 Scalable、Veno、BIC 和 YeAH 等算法。要理解这些算法的详细理论，重要的是理解 TCP 算法的本质。同时，要理解算法的详细运行流程，从状态变迁的角度思考也非常有效果。

接下来的第 4 章将基于状态变迁情况，通过与模拟得到的数据进行联系与对比，详细介绍 TCP 算法的一系列特点。

参考资料

- 竹下隆史，村山公保，荒井透，等 . 图解 TCP/IP. [M]. 乌尼日其其格，译 . 北京：人民邮电出版社，2013.
- 西田佳史 . TCP 詳説 [EB/OL]. パシフィコ横浜：Internet Week 99，1999.
- 《传输控制协议》（RFC 793）.
- 《TCP 慢启动、拥塞避免和快速重传》（RFC 2001）.
- 《TCP 拥塞控制》（RFC 2581）.
- 《TCP 快速恢复算法的优化版 NewReno》（RFC 6582）.
- Van Jacobson，Michael J.Karels. Congestion Avoidance and Control [J]. Proceedings of SIGCOMM, 1988.
- 甲藤二郎，村瀬勉 . TCP（Transimission Control Protocol）の改善 [J/OL]. 電子情報通信学会，3 群 4 編 - 二章，2014.

第4章

程序员必学的拥塞控制算法

逐渐增长的通信数据量与网络的变化

这里再回顾一下“拥塞”这个词，它实际指的就是网络中发生的拥堵现象。关于如何调整 TCP 数据传输量以避免拥塞的研究从未停止过。

本章首先简单介绍 TCP 的拥塞控制机制，然后通过 Wireshark 和 ns-3 进行模拟，引领大家详细观察拥塞控制算法的执行过程。

4.1 节介绍拥塞控制的目的、基本设计、状态迁移和拥塞控制算法的基本情况。4.2 节综合介绍各种代表性拥塞控制算法，并从定性和定量两个方面对各个拥塞控制算法进行分析研究。4.3 节使用 Wireshark，分析在虚拟机间进行文件传输时，TCP 的详细运行流程。4.4 节则使用 ns-3，探明无法通过抓包掌握的 TCP 内部变量的详细情况。

4.1 拥塞控制的基本理论

目的与设计，计算公式的基础知识

网络（通信网络）通常是由多台设备共享的。因此，假如单台设备随意发送大量数据，就很可能发生数据包拥堵，最终给整个网络带来很大的危害。这种情况便称为拥塞，到现在为止，人们已经针对拥塞避免问题进行了大量的研究。

本节将概述 TCP **拥塞控制**的基本理论。

拥塞控制的目的

假设把数据包比作车辆，把通信链路（有线、无线）比作道路，把路由器比作交通标识，那么通信网络几乎就是一个“交通网”。显而易见，两者都有出发地和目的地，而且都是用户一起共享整个网络。当进入网络内的“内容”超过网络本身的上限容量时，内部就会发生拥堵，这一点两个网络也一样。

通信网络与交通网最根本的不同，便是在发生拥堵时数据包可能直接被废弃[1]。不仅如此，交通网中的司机们都是主观能动地选择道路，而数据包不同，它们没有自己的意志。因此，在通信网络中，发送节点必须在发送数据包前预判网络的拥堵情况，并根据实际情况调整数据包发送量。

避免网络中发生**拥塞**的方法，便是拥塞控制技术。经过人们长年的研究，拥塞控制技术不断地进化发展。当发生拥塞时，无法处理的数据包会被废弃，导致发送者与接收者遭受损失。不仅如此，局部网络也会变得无法使用，给其他的使用者带来很大的麻烦。特别是 TCP，由于发送节点需要重新传输未收到确认应答的数据包（**重传控制**），因此会在拥塞时重复发送同一个数据包，导致数据传输量显著下降。正因如此，TCP 的拥塞避免技术一直都是研究的重点。本章将总览这些研究技术成果，并通过

1. 在交通网中，即使发生了拥堵，车辆显然也不会被废弃。

Wireshark 和 ns-3 确认其详细的运行流程。

此外，一些文献会使用“拥塞避免”（congestion avoidance）一词来代替“拥塞控制”（congestion control），但是由于本书后面的介绍将用“Congestion avoidance 状态”来表示拥塞状态中的一种，所以本书会统一使用“拥塞控制”的表述，以便进行区分。

拥塞控制的基本设计

首先，请大家将网络想象成一个由 TCP 发送节点（TCP sender）、接收节点（TCP receiver）和连接两者的网关所构成的简单网络。接下来，我们基于这个简单网络来介绍一下拥塞控制的整体概念（图 4.1）。

图 4.1 网络构成

TCP 拥塞控制的核心行为如下：发送节点根据收发节点之间的往返时延（*RTT*），以及从接收节点返回的确认应答（ACK）来预测网络的拥堵情况，并基于这些数据调整可发送数据量 *swnd*。

图 4.2 是某个发送节点的 TCP 报文段发送情况。其中 1 号到 6 号报文段已经发送完毕，而 7 号报文段尚未发送。在已发送的 TCP 报文段中，1 号到 3 号已经完成了确认应答（acked），而 4 号到 6 号尚未完成（*inflight*）。

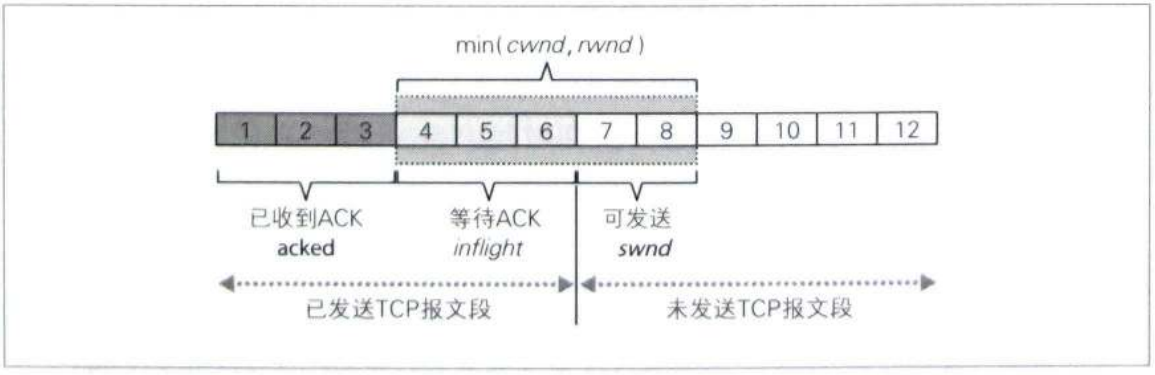

图 4.2 可发送的数据量（swnd）

如第3章所述，*cwnd* 指**拥塞窗口大小**，表示发送节点可以一次性发送且不需要等待ACK的最大可发送TCP报文段数量。而 *rwnd* 是**接收窗口大小**，表示接收节点可接收的最大TCP报文段数量，此上限值会由接收节点通知给发送节点。因此，min(*cwnd*, *rwnd*) 表示在同时考虑到发送节点和接收节点的情况下，无须等待ACK便可以一次性发送的 *MSS* 个数。在图4.2的情况下，min(*cwnd*, *rwnd*) 是5，可发送的TCP报文段是7号和8号。将以上过程用公式表示，结果如下。

$$swnd = \min(cwnd, rwnd) - inflight$$

此处值得注意的一点是，发送节点可以直接控制的、能决定 *swnd* 值的变量，只有 *cwnd* 一个。那么，如何控制 *cwnd* 的大小，才能避免拥塞发生呢？

举例来说，如果 *RTT* 值比较大，则可以认为当前网络中有数据包发生了拥堵。此外，如果发送节点收到接收节点发来的多个重复的ACK，显然可以认为网络中有数据包被丢弃了。无论是哪种情况，此时都需要减小 *cwnd*，竭尽全力控制数据发送量。反之，假如 *RTT* 比较小，就可以认为网络目前比较空闲，没有什么拥堵情况，此时无疑应该增大 *cwnd*，努力增加数据发送量才对。

从直观感受上来看，上述策略无疑完美无缺，然而又有个新的问题。到底 *RTT* 大到什么程度，才可以认为网络发生了拥堵呢？或者，重复收到多少次同样的ACK之后，才可以确信有数据包被丢弃了呢？这些问题很难回答。究其原因，通信网络是多台设备共享使用的，发送节点自身无法单独、实时地掌握网络的整体情况[1]。

为了解决此问题，TCP研究者们提出了一系列的拥塞控制算法。这些拥塞控制算法基于ACK去预测丢包和时延，并更新 *cwnd* 的值。

---

[1] 然而，如果网络环境本身是由TCP发送节点所构建，同时发送节点还可以控制所有的网络内设备的具体运行，那么就是另外一回事了。但是，在这种网络环境下，显然根本不可能发生网络拥塞，也就没必要进行拥塞控制。

拥塞控制中的有限状态机

TCP 拥塞控制算法采用有限状态机（finite state machine），根据情况使用不同的 *cwnd* 计算公式。简单来说，有限状态机就是指拥有有限状态，并且包含这些状态之间的迁移情况的一种数学概念。由于其与工业问题兼容性较好，所以常被用在自动售货机、语法分析、半导体设计和通信协议等多个领域中[①]。

描述拥塞控制算法的有限状态机有许多种。例如，Kurose 等是遵循 RFC 5681 的有限状态机[②]，但由于它是基于丢包的拥塞控制算法，也就是说，它以丢包为契机调整 *cwnd* 值，所以无法支持本书介绍的部分算法（4.2 节将介绍基于丢包的拥塞控制算法）。

本书采用的状态机基于 Linux 的具体实现（`net/ipv4/tcp.h`），具体情况如图 4.3 所示。

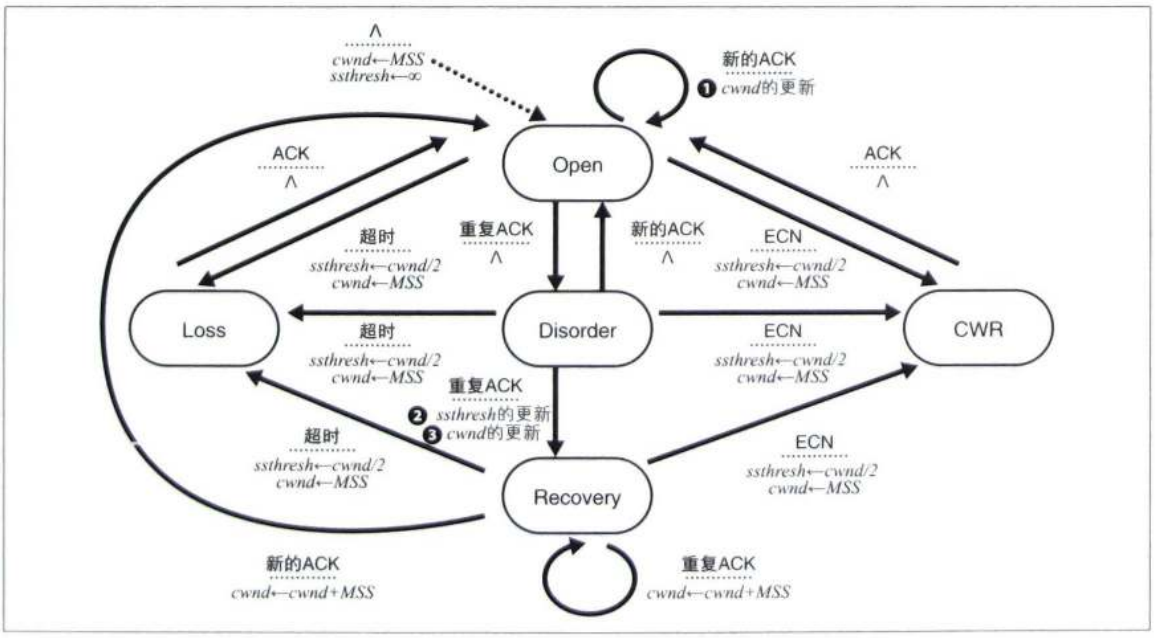

图 4.3 拥塞控制算法的状态迁移图

① 由于篇幅有限，本书无法对有限状态机进行详细定义，有兴趣的读者请参考计算机工程教科书来详细学习。

② James Kurose，Keith Ross. 计算机网络（第 6 版）：自顶向下方法 [M]. 陈鸣，译. 北京：机械工业出版社，2014.

5种状态

图4.3中的有限状态机有以下5种状态。

- Open：也就是正常状态。包括后文所述的Slow start和Congestion avoidance状态。收到全新ACK时的*cwnd*计算公式（图4.3❶），在不同的拥塞控制算法中有所不同。
- Disorder：在Open状态下，如果连续收到2次重复的ACK，便进入此状态。此时可能发生了轻微的网络拥塞。如果再次收到重复的ACK，便迁移到Recovery状态。从Disorder进入Recovery状态时，*ssthresh*和*cwnd*的计算公式（图4.3❷❸）在不同的拥塞控制算法中也有所不同。
- Recovery：在Open状态下，如果连续收到3次重复的ACK，便进入此状态。此时很可能发生了严重的网络拥塞。在Recovery状态下，如果收到全新的ACK，则进入Open状态。
- CWR：收到ECN（显式拥塞通知）后进入此状态。具体运行与Loss状态没有区别。
- Loss：*RTT*的值比超时重传时间（***RTO***）大，也就是说是检测到ACK超时但尚未收到新ACK的状态。此时可能发生了严重的网络拥塞。

状态迁移图 状态迁移条件与迁移后的行为

图4.3这样的图称为状态迁移图。它通常用来描述有限状态机等状态机（state machine）的状态及状态变化，其各个顶点代表状态，而各条边代表状态之间的迁移。各条边旁边的说明性文字中，虚线上面的是迁移条件，虚线下面的是迁移后的行为。

例如，从Disorder到Loss的箭头旁边有一行黑体字，从中可以看出，在Disorder状态下如果超时，则进入Loss状态，同时状态迁移之后*ssthresh*会变成*cwnd*的一半，即*cwnd*变成*MSS*。在这里，*ssthresh*代表从Slow start状态迁移到Congestion avoidance状态时*cwnd*的阈值，有关这两种状态，请参考后文的介绍。

lambda
Λ 表示没有对应的动作，虚线上方的 Λ 表示没有迁移条件。也就是说，箭头前方的状态是初始状态，而虚线下方的 Λ 则表示状态迁移后没有任何动作。

图 4.3 的❶～❸是拥塞控制算法的各个特点表现得最为明显的部分。4.2 节将以此部分为着眼点，对比代表性的拥塞控制算法。

拥塞控制算法示例 NewReno

4.2 节将详细比较拥塞控制算法，在那之前，我们先来看一下代表性算法 NewReno 的计算公式，了解一下其整体情况。

在 Open 状态下，NewReno 有以下两种子状态。

- Slow Start：当 *cwnd* 小于等于 *ssthresh* 时，进入此状态。在 Slow start 状态下，通常认为通信网络上发生拥塞的可能性较小，因此需要随着时间指数性地增大 *cwnd*。
- Congestion avoidance：当 *cwnd* 大于 *ssthresh* 时，进入此状态。在 Congestion avoidance 状态下，会名副其实地规避拥塞，因此需要随着时间线性地增大 *cwnd*。

综上所述，最终可以得到如下的伪代码。这与图 4.3❶一致。

$$\begin{aligned}&\textbf{If } cwnd \leqslant sshtresh\\&\quad\textbf{Then } cwnd \leftarrow cwnd + MSS\\&\quad\textbf{Else } cwnd \leftarrow cwnd + \frac{MSS}{cwnd}\end{aligned}$$

在 Slow start 状态下，每收到 1 个 ACK 就让 *cwnd* 增大 *MSS* 的大小，1 个 RTT 之后 *cwnd* 会变为 2 倍。重复此过程之后，*cwnd* 的增长曲线就是一个相对于时间的指数函数。而在 Congestion avoidance 状态下，每收到 1 个 ACK 就让 *cwnd* 增大 $\frac{MSS}{cwnd}$ 的大小，1 个 *RTT* 之后 *cwnd* 只增大 *MSS* 大

小。重复此过程之后可以发现，显然 *cwnd* 与时间的关系是线性函数关系。

当状态从 Disorder 迁移到 Recovery 时，*ssthresh* 和 *cwnd* 的计算会使用以下伪代码。这与图 4.3❷❸ 一致。

$$ssthresh \leftarrow \frac{cwnd}{2}$$

$$cwnd \leftarrow ssthresh + 3 \cdot MSS$$

专 栏

Linux 中拥塞控制算法的实现

有兴趣的读者请试着研究一下 Linux 中 TCP 的算法实现。我们可以通过“The Linux Kernel Archives”获取 Linux 内核的源代码[①]。

由于篇幅所限，这里只简单介绍。拥塞控制算法主要实现在 `net/ipv4/tcp_{算法名}.c` 的各个文件中。例如，后文描述的 BIC 算法中的关键算法二分搜索（binary search），主要实现在 `net/ipv4/tcp_bic.c` 的 `bictcp_update()` 函数中。读者也可以尝试看看其他各种拥塞控制算法。

NewReno 算法是后续各种拥塞控制算法的基础，这些控制算法大部分重写了 NewReno 的部分计算公式。因此，在接下来的 4.2 节，本书将聚焦于这些算法中所使用的、与 NewReno 不同的计算公式，并以此展示各个拥塞控制算法的特点。

① 可通过关键字“The Linux Kernel Archives”搜索其网址。——编者注

4.2 拥塞控制算法

通过理论 × 模拟加深理解

本节介绍具有代表性的拥塞控制算法，具体内容包括这些算法提出的背景和目的、计算公式以及模拟结果。此外，本节还会涉及用于拥塞控制算法分类的 3 种反馈形式。

本书介绍的拥塞控制算法 基于丢包、基于延迟、混合型

人们已经研究出了各种各样的拥塞控制算法。由于篇幅所限，本书无法全部涉及，因此仅将其中具有代表性的一些算法摘录出来展示在图 4.4 中。

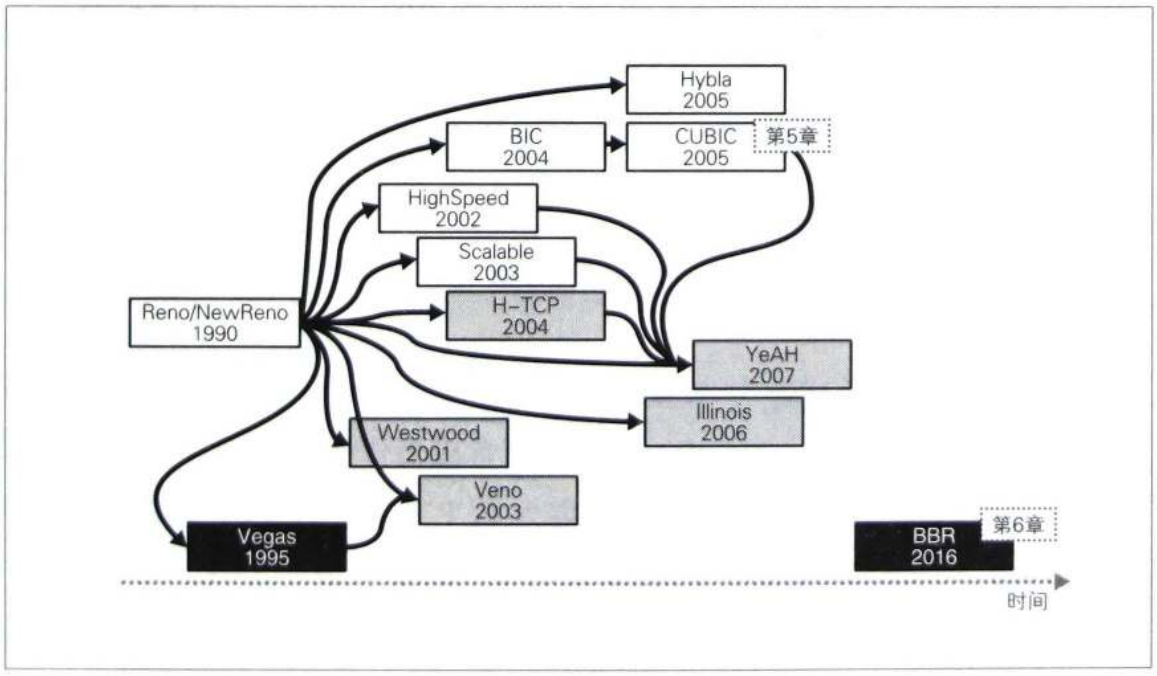

图 4.4 本书介绍的拥塞控制算法

在图 4.4 中，方框内是各个拥塞控制算法的名称与问世年份。横轴是时间轴，越靠近右边的拥塞控制算法，其年代越新。

不同的涂色代表不同的反馈形式，其中白底的是**基于丢包**的算法，黑

底的是**基于延迟**的算法，而灰底的则是**混合型**算法。在更新 *cwnd* 的值时，基于丢包的算法以“丢包”为基准，基于延迟的算法以“时延”为基准，混合型算法则以“前两者的结合”为基准。

箭头主要表示在 Open 状态下的计算公式的沿用关系。从图中可以看出，除了 BBR 以外的所有拥塞控制算法，都部分性地沿用了 NewReno 的思路。由于 CUBIC 是 BIC 的改良版本，所以从 BIC 到 CUBIC 有相应的箭头。YeAH 则根据情况不同选择 NewReno 或其他积极性拥塞控制算法（CUBIC、HighSpeed、Scalable 和 H-TCP）来使用。Veno 名副其实，是一个融合了 NewReno 和 Vegas 的拥塞控制算法。

在近些年提出和实现的拥塞控制算法中，**CUBIC** 和 **BBR** 算法尤其重要。CUBIC 是基于丢包的主流拥塞控制算法之一，默认搭载在 Linux 2.6.19 以后的版本中。而 BBR 是基于延迟的主流拥塞控制算法之一。BBR 在 2016 年 9 月由谷歌发布以后，在 Linux 内核 4.9 以后的版本中也可以选择使用，而且 Google Cloud Platform 和 YouTube 等平台也使用此算法。本书第 5 章和第 6 章将分别详细介绍 CUBIC 和 BBR 算法。本节将按顺序介绍其他的 11 种拥塞控制算法，这些算法有助于大家理解 CUBIC 和 BBR 算法。此外，为了行文方便，本书有时可能会使用与原论文不同的符号。

NewReno 拥塞控制算法的参考模型

NewReno 是在 1996 年被提出来的，它针对 1990 年提出的拥塞控制算法 Reno 进行了改良。NewReno 算法通常被当作拥塞控制算法的参考模型。与 NewReno 的亲和性[1]，一般被称为 TCP 友好性或 TCP 兼容性，它可以说是人们对 NewReno 以后的拥塞控制算法的要求条件之一。

AIMD

NewReno 的计算公式已经在 4.1 节介绍过了，因此这里就从另外的视角介绍一下 NewReno 的定位。NewReno 及其后的部分基于丢包的拥塞控

[1] 指的是与 NewReno 流量共存时，不会单方面地独占带宽的性质。