制算法可以概括为 AIMD（Additive Increase / Multiplicative Decrease，加法增大 / 乘法减小）。AIMD 是拥有以下特点的拥塞控制算法的总称。

- **在 Open 状态下，拥有 Slow start 和 Congestion avoidance 两种子状态。**
  - 在 Slow start 状态下，*cwnd* 呈指数性增大的趋势。
  - 在 Congestion avoidance 状态下，*cwnd* 呈线性增大的趋势（additive increase）。
- **在迁移到 Recovery 状态时，以常数因子（小于 1）对 *cwnd* 进行缩放（multiplicative decrease）。**

AIMD 与计算公式

如果将以上理论总结为公式，则结果如下所示。

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

首先，在 Open 状态下，收到新的 ACK 之后的计算公式（前面的图 4.3❶）可用上面的伪代码表示。α 相当于在 Congestion avoidance 状态下每个 *RTT* 内 *cwnd* 的增加量，在 NewReno 中 $\alpha=1$。

从 Disorder 状态迁移到 Recovery 状态时的计算公式（前面的图 4.3❷❸）可用下面的伪代码表示。β 相当于在往 Recovery 状态迁移时 *cwnd* 的减小比例，在 NewReno 中 $\beta=0.5$。

$$\begin{aligned}ssthresh &\leftarrow (1-\beta)\cdot cwnd\\cwnd &\leftarrow ssthresh + 3\cdot MSS\end{aligned}$$

不过，在有的图书与论文中，AIMD 中往 Recovery 状态迁移时的 *cwnd* 计算公式也可能是 $cwnd \leftarrow (1-\beta)\cdot cwnd$。为了与 RFC 5681 统一，本

书采用了上面的计算公式。此外，两者在本质上表示的是同一个动作，仅仅是在表现方式上不同，也就是“重复收到 ACK”这一促成状态变迁的契机是否反映到了 *cwnd* 的计算公式上。

——基于 ns-3 的模拟结果 NewReno

为了加深理解，这里介绍一下基于 ns-3 的模拟结果。发送节点在图 4.5 的网络下，往接收节点进行 20 秒的文件发送。如果大家下载了本书的源代码，可以很方便地修改条件，进行不同的模拟。有关使用 ns-3 进行模拟的详细流程，请参考 4.4 节。

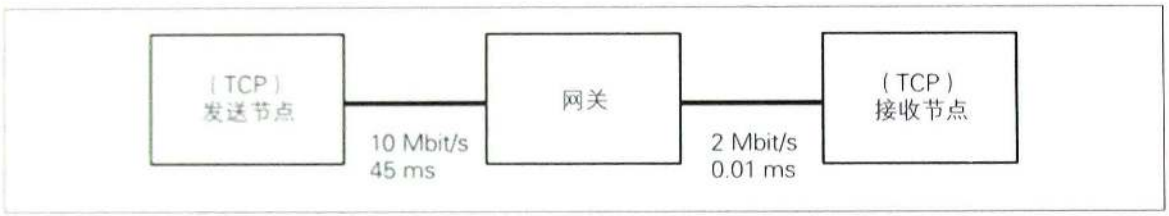

图 4.5 ns-3 中的网络构成

图 4.6 是发送节点的内部变量的变化情况。第 1 幅图表是 *cwnd*，第 2 幅是 *ssthresh*（图中写作 *ssth*，下同），第 3 幅是 *RTT*，而第 4 幅表示的是状态迁移的情况。需要注意的是，由于 *ssthresh* 的初始值非常大，所以第 2 幅图表中初始值会超出图表的范围（图 4.6❶）。此外，在第 4 幅图表中，灰色部分表示的是对应时刻所迁移到的状态。例如，最初的大约 2 秒时间处于 Open 状态，而之后迁移到了 Recovery 状态。

从图 4.6 可以看到如下文所述的状态迁移情况。首先，模拟刚开始的 1.93 秒左右处于 Open（Slow start）状态，*cwnd* 指数性增大（❷）。在 1.93 秒附近多次收到重复的 ACK 后，经过 Disorder 状态最终进入 Recovery 状态，因此 *cwnd* 大致减半（❸），*ssthresh* 也随之减小（❹）。请注意，由于此时停留在 Disorder 状态的时间非常短，所以图 4.6 的第 4 幅图表无法绘制出这部分状态（❺）。2.7 秒左右收到了新的 ACK，此时虽然瞬间返回到了 Open 状态，但是之后又收到重复 ACK，因此又经过 Disorder 状态进入 Recovery 状态（❻）。经过这一系列状态迁移，*cwnd* 和 *ssthresh* 再次减半（❼❽）。在 3.0 秒附近再次收到新的 ACK，并进入 Open 状态（❾）。

由于处于 Congestion avoidance 状态，所以 *cwnd* 呈线性增长趋势（❿）。从图 4.6 的第 4 幅图表中可以看到有一个小的波峰，从这一点可以看出拥塞之后到达的包的 *RTT* 是很大的（⓬）。

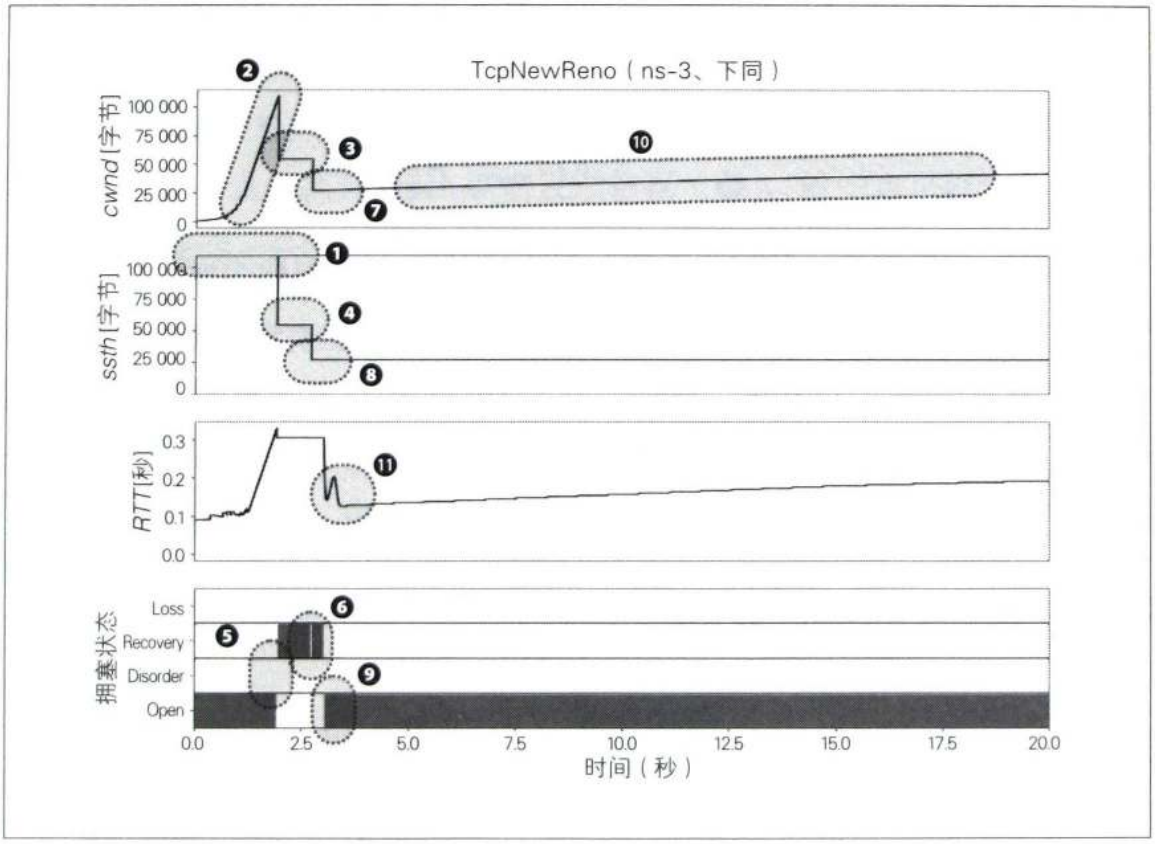

图 4.6 NewReno 的模拟结果

如前文所述，接下来我们将把各个拥塞控制算法中，与 NewReno 计算公式中所不同的部分摘录出来，深入挖掘这些算法各自的特点。

Vegas 具有代表性的基于延迟的拥塞控制算法

以 NewReno 为代表的基于丢包的拥塞控制算法以拥塞事件为契机调整数据发送量，从原理上来说是无法避免拥塞的发生的。为了解决这一问题，基于延迟的拥塞控制算法开始登上舞台，它以 *RTT* 的增减为契机来调整数据量。Vegas 于 1995 年提出，是具有代表性的基于延迟的拥塞控制算法之一。

计算公式 Vegas

Vegas 把根据 *RTT* 推算得到的通信链路上的缓存量 *Diff* 作为唯一指标，来调整数据发送量。在下面的公式中，RTT_{base} 是 *RTT* 的最小值，而 *RTT* 是最新的 *RTT* 值。

$$Diff \leftarrow \frac{cwnd}{RTT_{base}} - \frac{cwnd}{RTT}$$

公式右侧的第 1 项代表期望的发送速率，第 2 项则代表实际的发送速率，它们的差值 *Diff* 就是通信链路中所缓存的数据的发送速率。Vegas 所有的计算都基于此 *Diff* 值。例如，从 Slow start 状态迁移到 Congestion avoidance 状态的条件，便是 *Diff* 值大于一定值。在 Congestion avoidance 状态下，也是将 *Diff* 与两个阈值 α_{vegas} 和 β_{vegas} 相比较，然后调整 *cwnd*。

$$\begin{aligned}
&\textbf{If } Diff < \alpha_{vegas} \\
&\quad \textbf{Then } cwnd \leftarrow cwnd + \frac{1}{cwnd} \\
&\textbf{Else If } Diff > \beta_{vegas} \\
&\quad \textbf{Then } cwnd \leftarrow cwnd - \frac{1}{cwnd}
\end{aligned}$$

当 *Diff* 小于 α_{vegas} 时，就可以认为通信链路中并没有缓存多少数据包，也就是说发生拥塞的可能性较低，此时增大 *cwnd*；当 *Diff* 大于 β_{vegas} 时，也就是说通信链路中缓存了不少数据包，可以认为发生拥塞的可能性较高，于是减小 *cwnd*；当 *Diff* 在 α_{vegas} 和 β_{vegas} 之间时，就保持 *cwnd* 不变。$\beta_{vegas} - \alpha_{vegas}$ 的大小是调整 *cwnd* 稳定性的参数，如果此值较大，则 *cwnd* 相对比较稳定，但是会导致对 *RTT* 变化不敏感，最终增加发生拥塞的风险。而如果 $\beta_{vegas} - \alpha_{vegas}$ 的值较小，则虽然能及时响应 *RTT* 的变化，但会对很小的变化出现过度的反应，造成 *cwnd* 不稳定。

在 Slow start 状态下，*cwnd* 计算公式基本上与 NewReno 相同，不过 Vegas 有一点不同，那就是在每次 *RTT* 变化时会交替计算 *cwnd* 与 *Diff* 的值。

——基于 ns-3 的模拟结果 Vegas

图 4.7 所示的就是在与 NewReno 相同的条件下进行模拟的结果。从图中可以看出，在 Slow start 状态下，在每次 *RTT* 变化后，*cwnd* 的值都会跟前文所述的一样被更新（图 4.7❶）。进入 Congestion avoidance 状态后，*cwnd* 和 *RTT* 的值比较稳定，这一点可以说是 Vegas 最显著的特征了（❷❸）。从第 4 幅图表可以看出，发送节点在 20 秒内一次也没有收到重复的 ACK，一直保持着 Open 状态发送数据（❹）。这一点可以说是 Vegas 与 NewReno 等一系列基于丢包的拥塞控制算法的最大不同点。

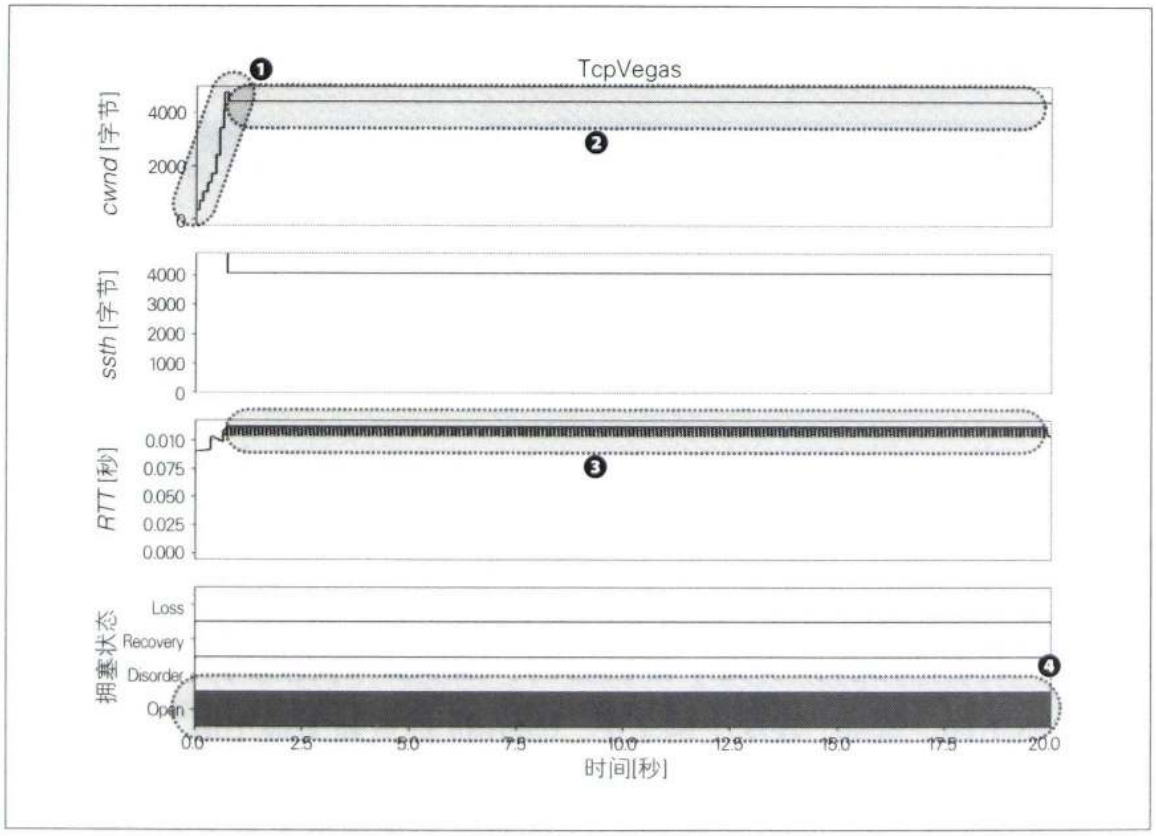

图 4.7 Vegas 的模拟结果

Westwood 面向无线通信的混合型拥塞控制算法

Westwood 于 2001 年提出，是一个主要针对无线通信的混合型拥塞控制算法。当状态迁移至 Recovery 时，NewReno 算法会毫无理由地直接将

ssthresh 值减半。然而在无线通信等即使没有发生拥塞也会出现丢包的网络链路中，这样做会导致带宽利用率恶化[①]。

计算公式 Westwood

因此，Westwood 根据 ACK 的接收间隔推测端到端的带宽，并以此为基础，提出了状态迁移到 Recovery 时 *ssthresh* 的计算方法（快速恢复）。在下面的公式中，*BWE* 代表根据 ACK 推测的端到端的带宽，而 RTT_{base} 代表 *RTT* 的最小值。

$$ssthresh \leftarrow BWE \cdot RTT_{base}$$
$$cwnd \leftarrow ssthresh + 3 \cdot MSS$$

当状态迁移到 Loss 时，也同样更新 *ssthresh*。需要注意，*cwnd* 会被初始化为 *MSS*。

$$ssthresh \leftarrow BWE \cdot RTT_{base}$$
$$cwnd \leftarrow MSS$$

基于 ns-3 的模拟结果 Westwood

图 4.8 所示的是在与 NewReno 相同条件下模拟的结果。从图中可以看出，当状态迁移到 Recovery 和 Loss 时，*ssthresh* 的表现与其他拥塞控制算法有所不同。例如，在这个例子中，*ssthresh* 被更新为最小值[②]（图 4.8 ✪），究其原因应该是预测的 *BWE* 值非常小。由于篇幅所限，这里无法详细分析，不过如果使用 4.4 节介绍的 ns-3，并变换各种条件进行模拟，就可以深入研究和分析为何 *BWE* 如此之小。

---

① 与通信链路比较稳定的有线链路相比，无线通信由于电波强度的衰减与变化比较剧烈，哪怕使用了各种技术进行优化，丢包也会在 10^{-2} 左右。

② 在 ns-3.27 的实现（src/internet/model/tcp-westwood.cc）中，*ssthresh* 的计算结果的下限值是 2 MSS。

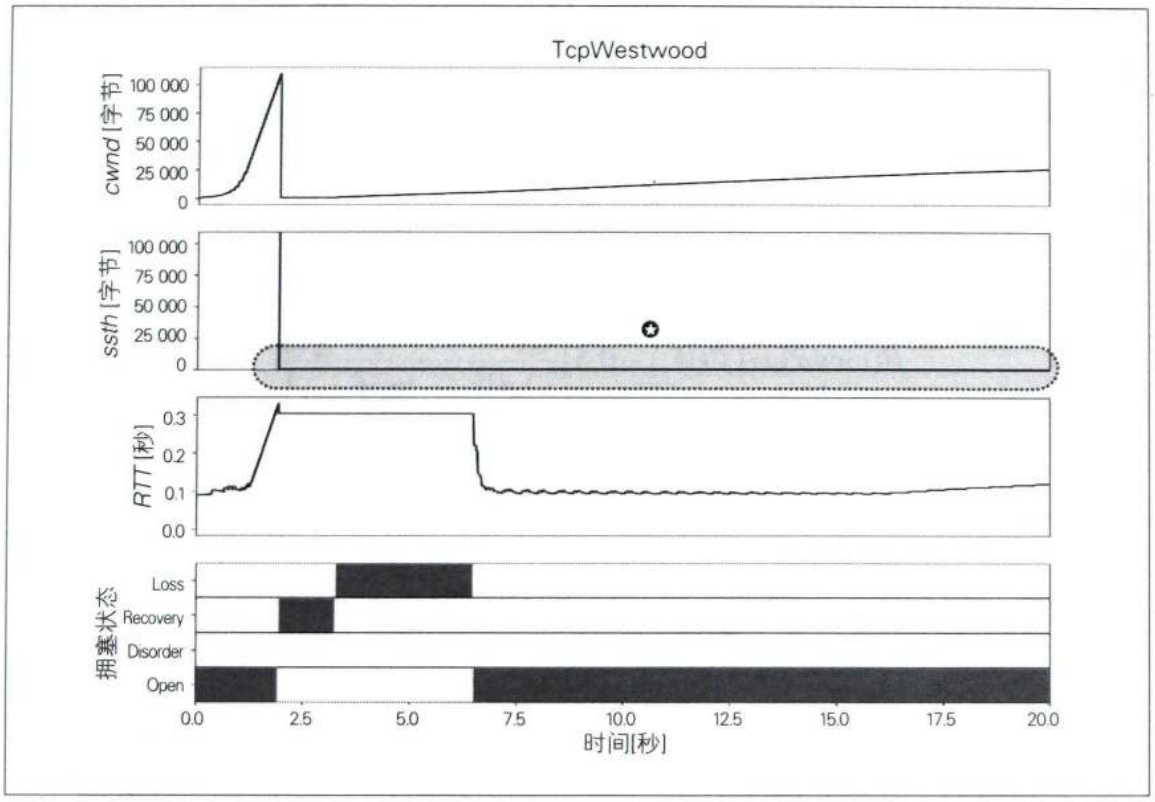

图 4.8 Westwood 的模拟结果

HighSpeed 面向长肥管道的基于丢包的拥塞控制算法 1

HighSpeed 于 2002 年提出，是一个面向**长肥管道**（long fat pipe，高速、远距离通信链路）的基于丢包的拥塞控制算法。此算法的主要特点是在 Congestion avoidance 状态下 *cwnd* 增幅比较大，在 Recovery 状态下 *cwnd* 值恢复较快。上述操作只在 *cwnd* 大于 W_{thresh} 且丢包率小于 P_{thresh} 时才进行，这样做主要是考虑到在 HighSpeed 和 NewReno 共存的网络中，一旦拥塞发生，HighSpeed 不会单方面占有所有的网络带宽。

此外，在面向长肥管道的基于丢包的拥塞控制算法中，具有代表性的除了 HighSpeed 之外，还有 Scalable、BIC 和 CUBIC。本章还会介绍 Scalable 和 BIC，第 5 章则介绍 CUBIC（5.3 节也会介绍 BIC），请大家在阅读的同时，关注一下这些算法之间的异同。

计算公式 HighSpeed

HighSpeed是扩展了AIMD思想的拥塞控制算法。α和β的函数如下所示。此外，P_1（$P_1 < P$）表示丢包率的目标值，W_1（$W_1 > W$）表示*cwnd*的目标值。

$$\beta(cwnd) = [\beta(W_1) - 0.5]\frac{\lg(cwnd) - \lg(W_{thresh})}{\lg(W_1) - \lg(W_{thresh})} + 0.5$$

$$\alpha(cwnd) = \frac{2 \cdot cwnd^2 \cdot \beta(cwnd) \cdot p(cwnd)}{2 - \beta(cwnd)}$$

此外，通过满足下面的公式，可以计算得到$p(cwnd)$。

$$\lg[p(cwnd)] = [\lg(P_1) - \lg(P_{thresh})]\frac{\lg(cwnd) - \lg(W_{thresh})}{\lg(W_1) - \lg(W_{thresh})} + \lg(P_{thresh})$$

基于ns-3的模拟结果 HighSpeed

在与NewReno相同的条件下，HighSpeed的模拟结果如图4.9所示。从图中可以看出，在Congestion avoidance状态下*cwnd*的增幅较大（图4.9❶），而迁移到Recovery状态时*cwnd*的降幅比较小（❷），这与HighSpeed的设计目的一致。

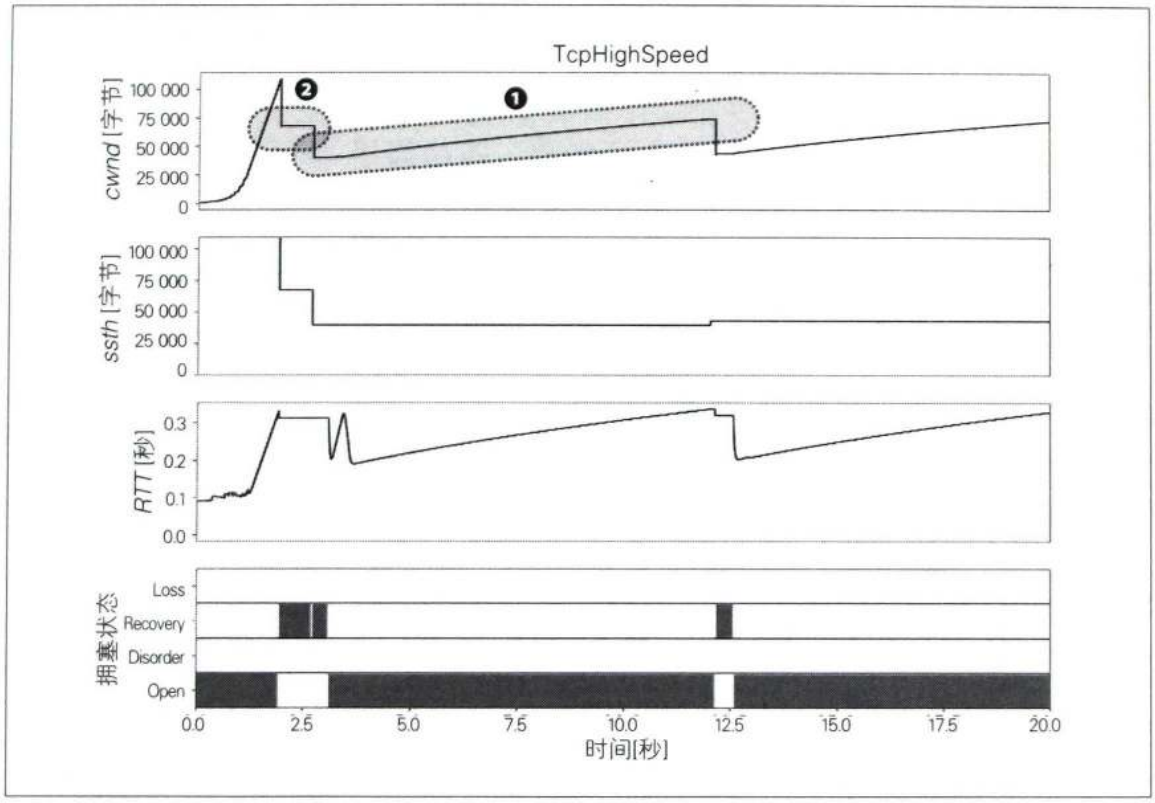

图 4.9 HighSpeed 的模拟结果

Scalable 面向长肥管道的基于丢包的拥塞控制算法 2

Scalable 于 2003 年提出，也是一个面向长肥管道的基于丢包的拥塞控制算法。它的特点是在 Congestion avoidance 状态下，*cwnd* 呈指数级增长。

计算公式 Scalable

在 Open 状态下，Scalable 的计算公式可用以下伪代码表示。此处，α[①] 应满足 $0 < \alpha < 1$。原论文推荐的取值是 $\alpha = 0.01$。

If $cwnd \leqslant ssthresh$
  Then $cwnd \leftarrow cwnd + MSS$
  Else $cwnd \leftarrow cwnd + \alpha \cdot MSS$

① 这个 α 与前面介绍 AIMD 时的 α 不同。

状态往 Recovery 迁移时的计算公式可用与 AIMD 相同的以下伪代码表示。原论文推荐的取值是 $\beta=0.125$。

$$ssthresh \leftarrow (1-\beta)\cdot cwnd$$
$$cwnd \leftarrow ssthresh + 3\cdot MSS$$

基于 ns-3 的模拟结果 Scalable

在与 NewReno 相同的条件下，Scalable 的模拟结果如图 4.10 所示。我们从图中可以看出，在 Congestion avoidance 状态下，*cwnd* 是呈指数级增长的（图 4.10❶）。另外，还可以看出，在往 Recovery 状态迁移时，*cwnd* 和 *ssthresh* 的值并没有大幅减小（❷）。而往 Recovery 状态迁移的频率，在本章介绍的拥塞控制算法中，Scalable 算法是最高的（❸）。换句话说，Scalable 是最主动的算法。这和之前介绍的 Vegas 形成了鲜明的对比。

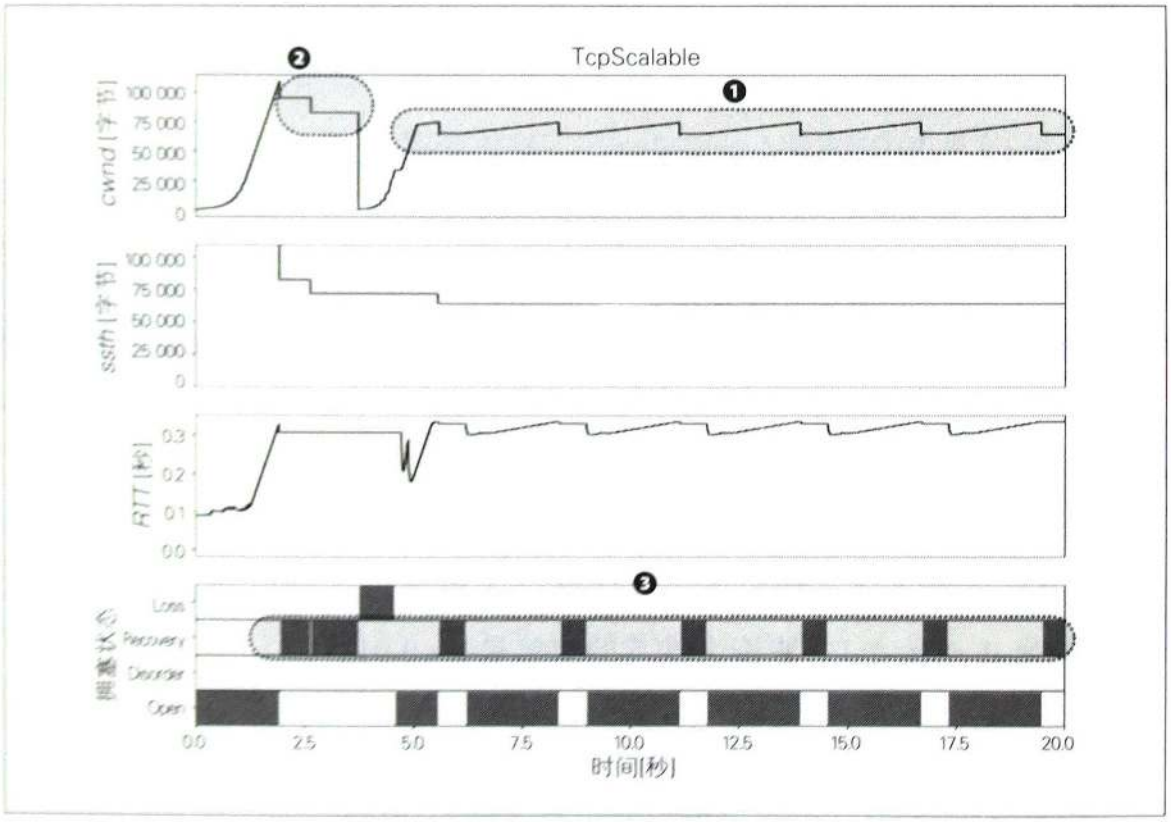

图 4.10 Scalable 的模拟结果

Veno 面向无线通信的混合型拥塞控制算法

Veno 也是 2003 年提出的，它是一个主要面向无线通信的拥塞控制算法。

NewReno 等过去的 AIMD 算法由于无法将随机丢包引起的（与拥塞无关的）重复 ACK 与拥塞引起的重复 ACK 区分开来，所以一直存在无线通信发送速率过低的问题。因此，Veno 使用 Vegas 算法所引入的 *Diff* 来预测拥塞程度，并以此来规避前述问题。如前文所述，Veno 名称的由来是 Reno 和 Vegas。因为 Veno 使用重复 ACK 和 *RTT* 来控制数据发送量，所以它属于混合型拥塞控制算法。Veno 也是 AIMD 算法的一种。

计算公式 Veno

Veno 算法会持续计算下列公式中的 N 值，并将 N 作为预测通信链路拥塞状况的指标。

$$N = Diff \cdot RTT_{base} = \left(\frac{cwnd}{RTT_{base}} - \frac{cwnd}{RTT} \right) \cdot RTT_{base}$$

在 Open 状态下，计算公式可用以下伪代码表示。请注意，在处于 Congestion avoidance 状态且 $N \geq \beta_{veno}$ 时，由于通信链路中缓存了较多数据，所以每收到 2 次 ACK 只会更新 1 次 *cwnd* 值。

If $cwnd \leq sstresh$

Then $cwnd \leftarrow cwnd + MSS$，每收到 1 次 ACK

Else If $N < \beta_{veno}$

Then $cwnd \leftarrow cwnd + \frac{MSS}{cwnd}$　　每收到 1 次 ACK

Else $cwnd \leftarrow cwnd + \frac{MSS}{cwnd}$　　每收到 2 次 ACK 更新 1 次

状态往 Recovery 迁移时的计算公式可用以下伪代码表示。当 $N < \beta_{veno}$ 时，由于通信链路中缓存的数据量比较少，所以可以认为重复 ACK 是因为无线通信中的随机丢包引起的，所以要控制 *ssthresh* 的减小量。

$$
\begin{aligned}
&\textbf{If } N < \beta_{veno} \\
&\quad \textbf{Then } ssthresh \leftarrow 0.8 \cdot cwnd \\
&\quad \textbf{Else } ssthresh \leftarrow 0.5 \cdot cwnd \\
&cwnd \leftarrow ssthresh + 3 \cdot MSS
\end{aligned}
$$

基于 ns-3 的模拟结果 Veno

在与 NewReno 相同的条件下，Veno 的模拟结果如图 4.11 所示。

由于此次试验发生在几乎不会发生随机丢包的模拟环境下，因此实际得到的结果与 NewReno 基本相同。大家如果使用 4.4 节介绍的 ns-3 调整数据包错误率，并观察 Veno 的运行情况，想必一定会有更深的理解。

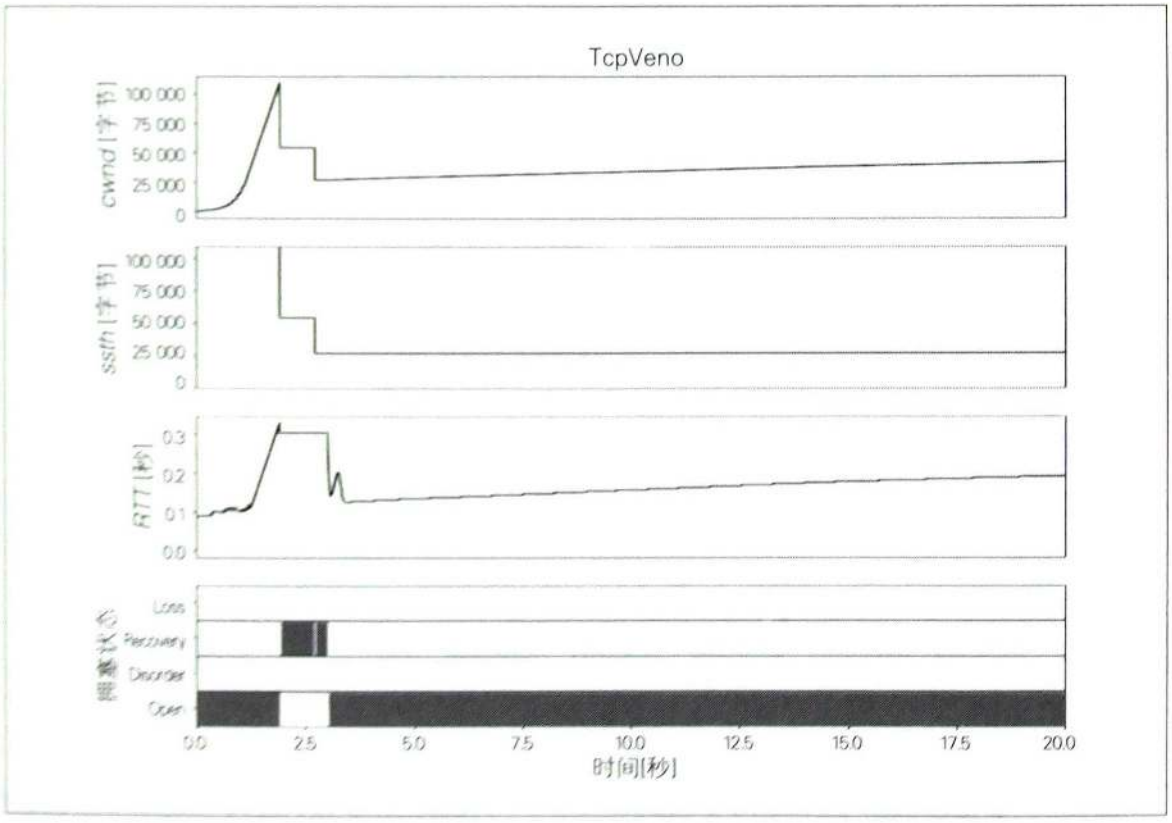

图 4.11 Veno 的模拟结果

BIC 面向长肥管道的基于丢包的拥塞控制算法 3

BIC（Binary Increase Congestion control）于 2004 年提出，是一个面向长肥管道的基于丢包的拥塞控制算法。

Scalable 和 HighSpeed 都是面向长肥管道的算法，但它们都有着 *RTT* 公平性（RTT fairness）较差的问题。所谓的 *RTT* 公平性，指的是当 *RTT* 不同的多个网络流并存时，它们之间可以公平地分配带宽的特性。特别是 Scalable 的 *RTT* 公平性极差，*RTT* 较小的网络流甚至会独占全部的带宽。这主要是 *cwnd* 呈指数级增长，两个网络流的 *cwnd* 之间的差距不断拉大所造成的结果。

计算公式 BIC

为了解决这一问题，BIC 提出了让 *cwnd* 按照对数函数增长的方法。BIC 是通过二分搜索（binary search）寻找最佳 *cwnd* 的算法。也就是说，将状态迁移到 Recovery 之前的 *cwnd* 值作为 $W_{\max}$，使用当前的 *cwnd* 与 $W_{\max}$ 中间的值作为新的 *cwnd*。

状态往 Recovery 迁移时的计算公式可用下面的伪代码表示。请注意，当 *cwnd* 小于阈值 W_{thresh} 时，BIC 的行为与 NewReno 一致。此外，在状态迁移到 Recovery 之前，如果 *cwnd* 小于 $W_{\max}$，则很有可能出现可用带宽减少的趋势，因此通常会将 $W_{\max}$ 设为比平常值更小一点的值（$W_{\max}$ 和新 *cwnd* 中间的值），以实现更快收敛的目的。这称为 Fast convergence（快速收敛）。

If $cwnd < W_{thresh}$
  Then $cwnd \leftarrow 0.5 \cdot cwnd$
  Else
    If $cwnd < W_{\max}$
      Then $W_{\max} \leftarrow \dfrac{cwnd + (1-\beta) \cdot cwnd}{2}$
      Else $W_{\max} \leftarrow cwnd$
  $cwnd \leftarrow (1-\beta) \cdot cwnd$

在Open状态下，BIC的计算公式可用以下伪代码表示。这里与前面往Recovery状态迁移时一样，当*cwnd*小于阈值W_{thresh}时，运行逻辑与NewReno相同。此外，α_{max}是α的上限值。

$$
\begin{aligned}
&\textbf{If } cwnd < W_{thresh}\\
&\quad \textbf{Then } \alpha \leftarrow 1\\
&\quad \textbf{Else}\\
&\qquad \textbf{If } cwnd < W_{\max}\\
&\qquad\quad \textbf{Then } \alpha \leftarrow \frac{W_{\max} - cwnd}{2 \cdot MSS}\\
&\qquad\quad \textbf{Else } \alpha \leftarrow \frac{cwnd - W_{\max}}{MSS}\\
&\qquad \alpha \leftarrow \min(\alpha, \alpha_{\max})\\
&\qquad \alpha \leftarrow \max(\alpha, 1)\\
&cwnd \leftarrow cwnd + \frac{\alpha \cdot MSS}{cwnd}
\end{aligned}
$$

基于ns-3的模拟结果 BIC

在与NewReno相同的条件下，BIC的模拟结果如图4.12所示。从最上面的图中我们可以看出，从Loss状态迁移到Open状态之后，*cwnd*以对数函数的形式逐渐增长并逼近之前的最大值（图4.12❶）。与*cwnd*呈指数级增长的Scalable算法相比，BIC的主要特点是往Recovery状态迁移的频率较低（❷）。

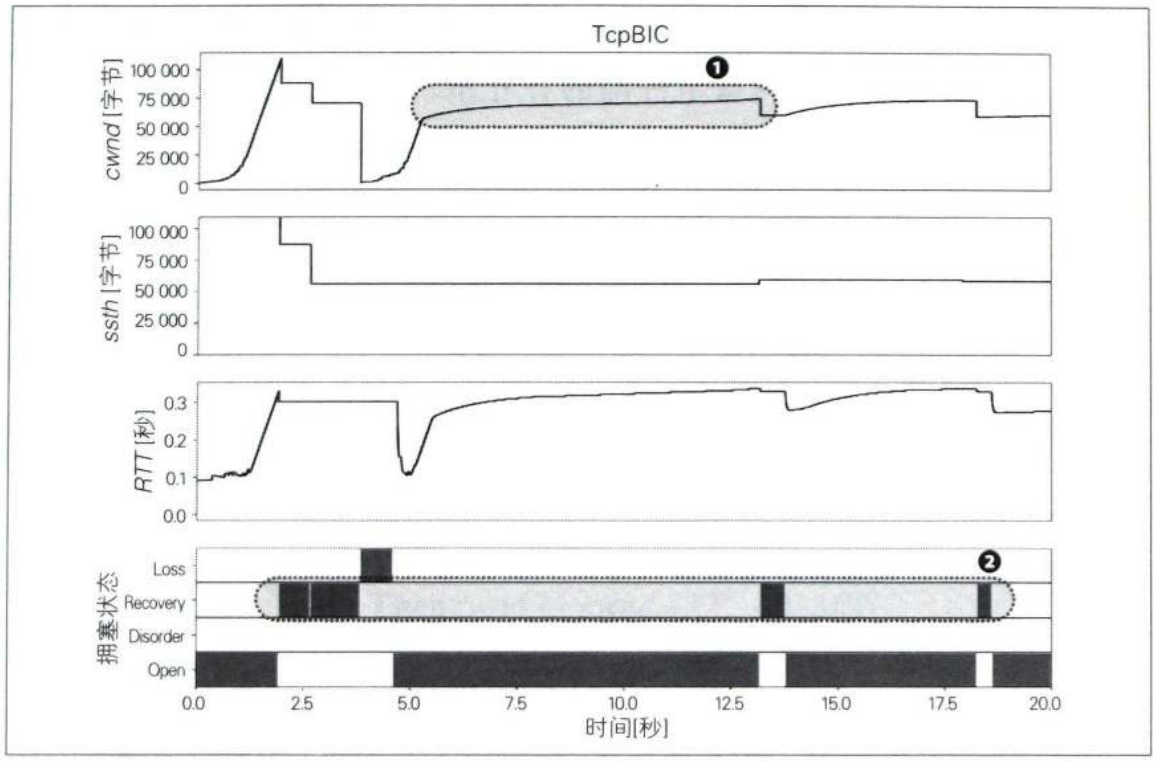

图 4.12 BIC 的模拟结果

H-TCP 面向长肥管道的混合型拥塞控制算法

H-TCP 于 2004 年提出，是一个面向长肥管道的混合型拥塞控制算法。H-TCP 算法的特点是，使用 *RTT* 等数值计算 AIMD 的 α 和 β 参数。

计算公式 H-TCP

具体来说，α 是通过以下伪代码计算出来的。在公式中，Δ 表示从刚发生的拥塞事件开始经过的时间。Δ_{thresh} 是需要提前设置的参数。此外，最后的 $\alpha \leftarrow 2(1-\beta)\alpha$ 是实现 TCP 友好性所必需的调整公式。

$$
\begin{aligned}
&\textbf{If } \Delta \leqslant \Delta_{thresh} \\
&\quad \textbf{Then } \alpha \leftarrow 1 \\
&\quad \textbf{Else } \alpha \leftarrow 1 + 10(\Delta - \Delta_{thresh}) + \left(\frac{\Delta - \Delta_{thresh}}{2}\right)^2 \\
&\alpha \leftarrow 2(1-\beta)\alpha
\end{aligned}
$$