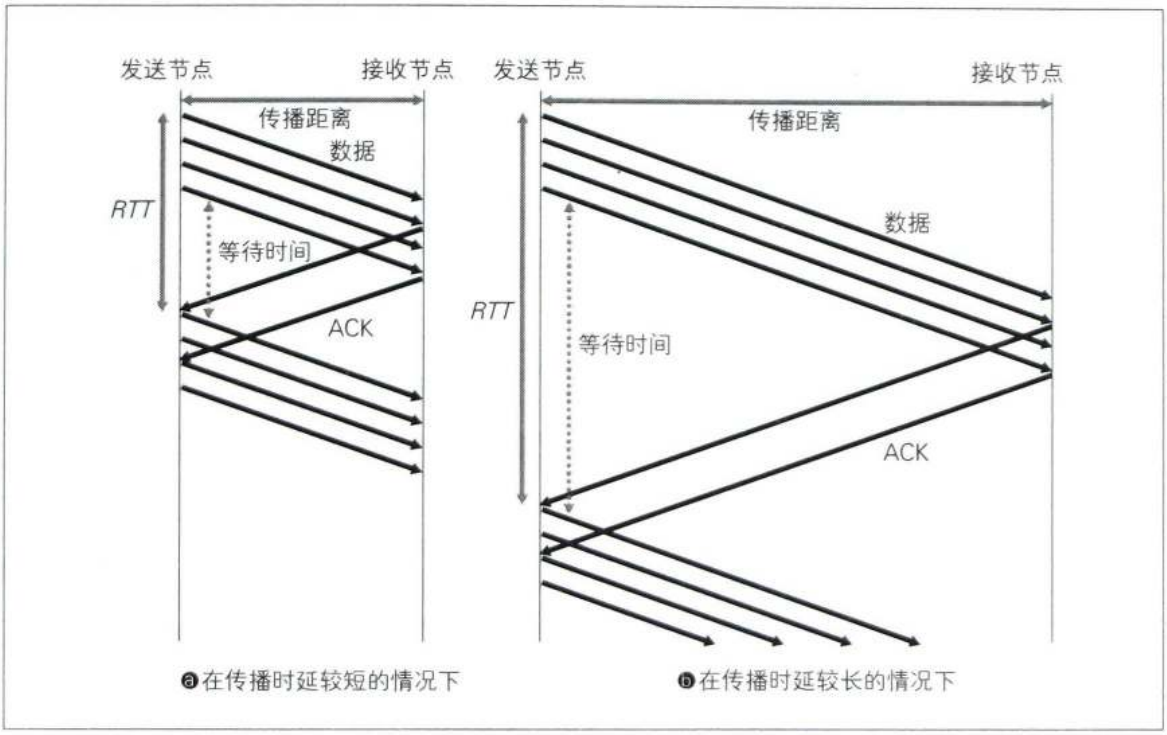


图 5.5 时延提高导致等待时间增加

最终的结果是，在拥塞窗口大小不变的情况下，通信环境的时延越大，吞吐量也就越小。

上文大致总结了长肥管道下 NewReno 算法所产生的问题。不过，本节只是定性地或者说是从理论上进行了介绍。下一节将进行模拟，并结合实际的模拟结果进一步详细介绍本节提到的问题。

5.2 基于丢包的拥塞控制

以丢包情况为指标的一种历史悠久的方法

在 TCP 拥塞控制算法中，有一种**基于丢包**的拥塞控制算法。它以丢包情况作为判断拥塞的指标，一直以来被广泛地应用在互联网中。这里将以 NewReno 为代表，介绍一下基于丢包的拥塞控制算法的基本流程。

基于丢包的拥塞控制算法的基本情况 丢包数量、拥塞窗口大小和AIMD

在TCP拥塞控制算法中，NewReno算法被分类为基于丢包的拥塞控制算法。基于丢包的拥塞控制算法指的是基于丢包情况判断网络拥塞状态的算法。

在通常情况下，如果网络较为空闲，则数据包不会被丢弃，全部都可以顺利地传输到目的地；反之，网络越拥堵，则由于缓冲区溢出等原因，链路上被废弃的数据包就越多。

基于丢包的拥塞控制算法以此为基本思路，如果发现**被丢弃的数据包**变多，则认为拥塞情况变得严重了，此时就调整**拥塞窗口大小**。简单来说，在没有发生丢包时，缓慢地增大拥塞窗口大小，而在检测出丢包时，大幅度减小拥塞窗口大小。换句话说，只要数据包不被丢弃，就尽可能地提高数据吞吐量。

然而，由于无法直接知道网络的具体拥塞情况，所以只能缓慢地增加数据包数量，并在发现数据包被丢弃时减少数据包的数量。如同第4章所介绍的一样，此方法被称为AIMD，下面将进行详细的介绍。

AIMD控制 加法增大，乘法减小

在发生丢包之前，缓慢地增大拥塞窗口大小（additive increase，加法增大），拥塞发生之后，大幅减小拥塞窗口大小（multiplicative decrease，乘法减小），这便是AIMD控制算法。图5.6展示的就是AIMD控制的大致流程。

AIMD控制算法在调整拥塞窗口大小时，一般使用下页的公式[①]，利用时刻 t 的拥塞窗口大小 $w(t)$ 计算时刻 $t+1$ 的拥塞窗口大小。此外，拥塞窗口大小一般会在每次收到ACK之后更新。

① 在第4章，我们使用 *cwnd* 表示拥塞窗口大小，但在第5章和第6章，使用 $w(t)$ 表示拥塞窗口大小。

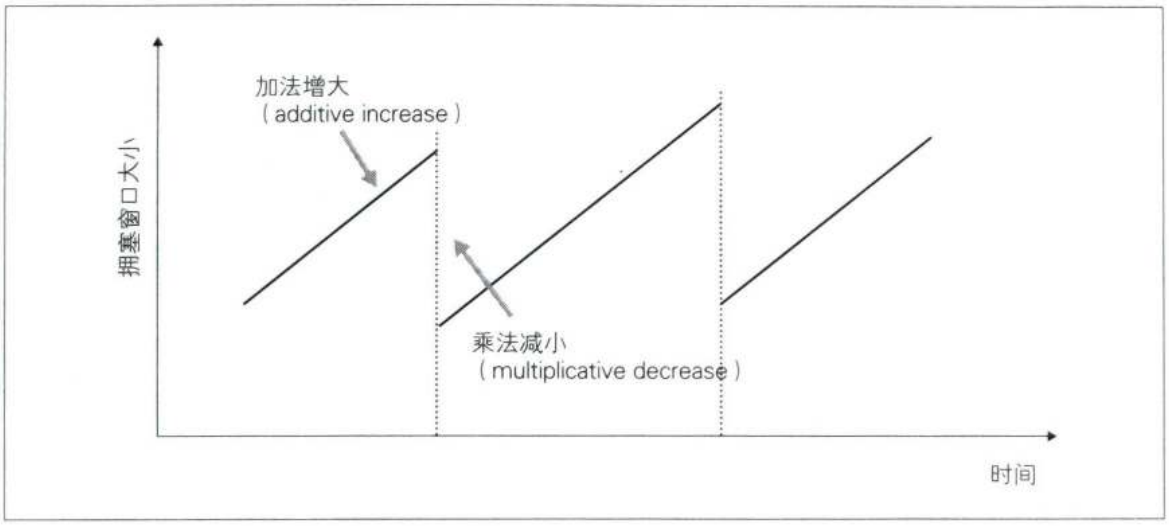


图 5.6 AIMD 控制的流程

$$w(t+1)=\begin{cases} w(t)+\alpha & \text{非拥塞时} \\ w(t)\cdot\beta & \text{拥塞时} \end{cases} \qquad \text{（公式 5.1）}$$

这里的变量 α 和 β[①] 对应的分别是每个 *RTT* 内非拥塞时拥塞窗口大小的增加量，以及发现丢包时拥塞窗口大小的减小量。公式的含义如下：如果网络上没有发生拥塞，则拥塞窗口大小每次就增大 α；如果发生了拥塞，则拥塞窗口大小每次按照 $w(t)\cdot\beta$ 的值进行乘法减小。

NewReno 的（严格来说是在拥塞避免阶段的）拥塞控制，毫无疑问就是 AIMD 控制算法，其中 $\alpha=1/w(t)$，$\beta=0.5$，其对应的行为就是对于每个 *RTT*，拥塞窗口大小增大 1。

显然，只要修改上面公式中的参数 α 和 β，就可以随意地更改拥塞窗口大小的增大速度和减小速度。例如，只要增大 α，就可以加快拥塞窗口大小的增大速度，而如果增大 β，就可以降低拥塞窗口大小的减小速度。事实上，研究者们提出了若干个调整了这两个参数的优秀方案，本书将在后文介绍这些方案。

这里，我们先以 NewReno 为例，结合实际的模拟测试来看一下运用

① 变量 β 的定义在第 4 章和第 5 章不太一样（主要原因是在不同的参考文献中，变量的定义和公式有所不同）。

AIMD控制算法时拥塞窗口大小的变化情况。

[实测]NewReno的拥塞窗口大小的变化情况

从确认模拟条件和测试项目开始

图5.7展示的是模拟条件。这里的模拟条件和第4章一样，但其中的拥塞控制算法使用NewReno。我们以一侧链路的速度和传播时延为变量，多次测试并比较结果数据。

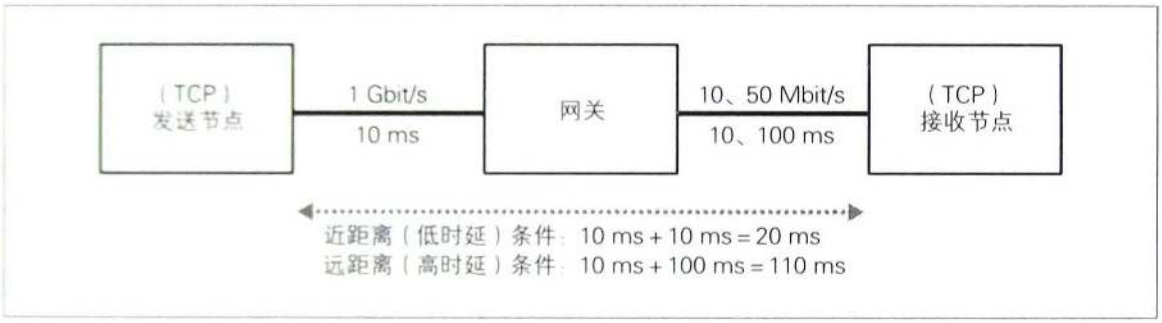


图5.7 模拟条件（实验1）

也就是说，将链路速度分成两档——低速（=10 Mbit/s）和高速（=50 Mbit/s），将传播时延也分为两档，分别是近距离[=20 ms（=10 ms+10 ms）]和远距离[=110 ms（=10 ms+100 ms）]，两者组合总共形成4种测试。这里将本次的模拟条件称为“实验1”。

要收集的数据具体有以下4项。

- 拥塞窗口大小（*cwnd*）
- 吞吐量（基于到达接收节点的数据量，每5秒计算一次）
- 拥塞状态
- *RTT*

此外，将接收窗口大小（*rwnd*）设为65 535字节，打开窗口扩大选项，把最大数据包大小*MTU*设置为1500字节，测试时间设置为100秒。由于本次模拟的主要目的是观察拥塞窗口大小的变化情况，所以将网关队列的最大容量设置为10个包，这样就更容易发生丢包。

——实验 1 的运行和测试结果的保存位置

要运行实验 1，需要打开 ns-3[①] 的根目录（`~/ns3/ns-allinone-3.27/ns-3.27`），并输入以下命令。

```
$ ./scenario_5_1.sh      ←运行实验1
```

测试的运行结果保存在 `data/chapter5` 目录下的 `05_xx-scl-*.data` 文件中，同时相对应的图形化数据保存在同一目录下的 `05_xx-scl-*.png` 文件中。

此外，由于客户操作系统的 `~/ns3/ns-allinone-3.27/ns-3.27/data` 目录与宿主操作系统的 `tcp-book/ns3/vagrant/shared/` 目录是同步的，所以也可以在客户操作系统中查看相应的文件。下文的 `tcp-book/` 文件夹指的是从 https://github.com/ituring/tcp-book 下载回来的文件夹。

——模拟运行结果 确定长肥管道下 NewReno 的问题

图 5.8 集中展示了模拟运行结果。我们可以看到，无论在哪种情况下，都是流量从 1 Gbit/s 的高速网络出来并经过网关之后速度降低，导致丢包出现，随后便进入拥塞避免阶段。因此，接下来将具体介绍在这之后，不同的网络环境下的表现。

在**窄带、低时延环境**（图 5.8 的 10 Mbit/s、10 ms）下，通过重复进行 AIMD 控制，最终得到了一个比较稳定的吞吐量。换句话说，缓慢地增大拥塞窗口大小，在其达到一定值后拥塞发生，此时一口气减小拥塞窗口大小。虽然进行了这一系列操作，但由于带宽较窄，网络很快就能恢复到原来的吞吐量，因此很难观察到吞吐量下降。

在**窄带、高时延环境**（图 5.8 的 10 Mbit/s、100 ms）下，由于 *RTT* 变大，吞吐量恢复到 10 Mbit/s 需要花费大约 50 秒。之后，直到 100 秒也没有观察到拥塞现象，这段时间 *cwnd* 大小缓慢地增大。虽然说是窄带宽，但其实高时延带来的影响已经开始表现出来了。

---

① 有关 ns-3 的环境搭建，请参考 4.4 节的内容。

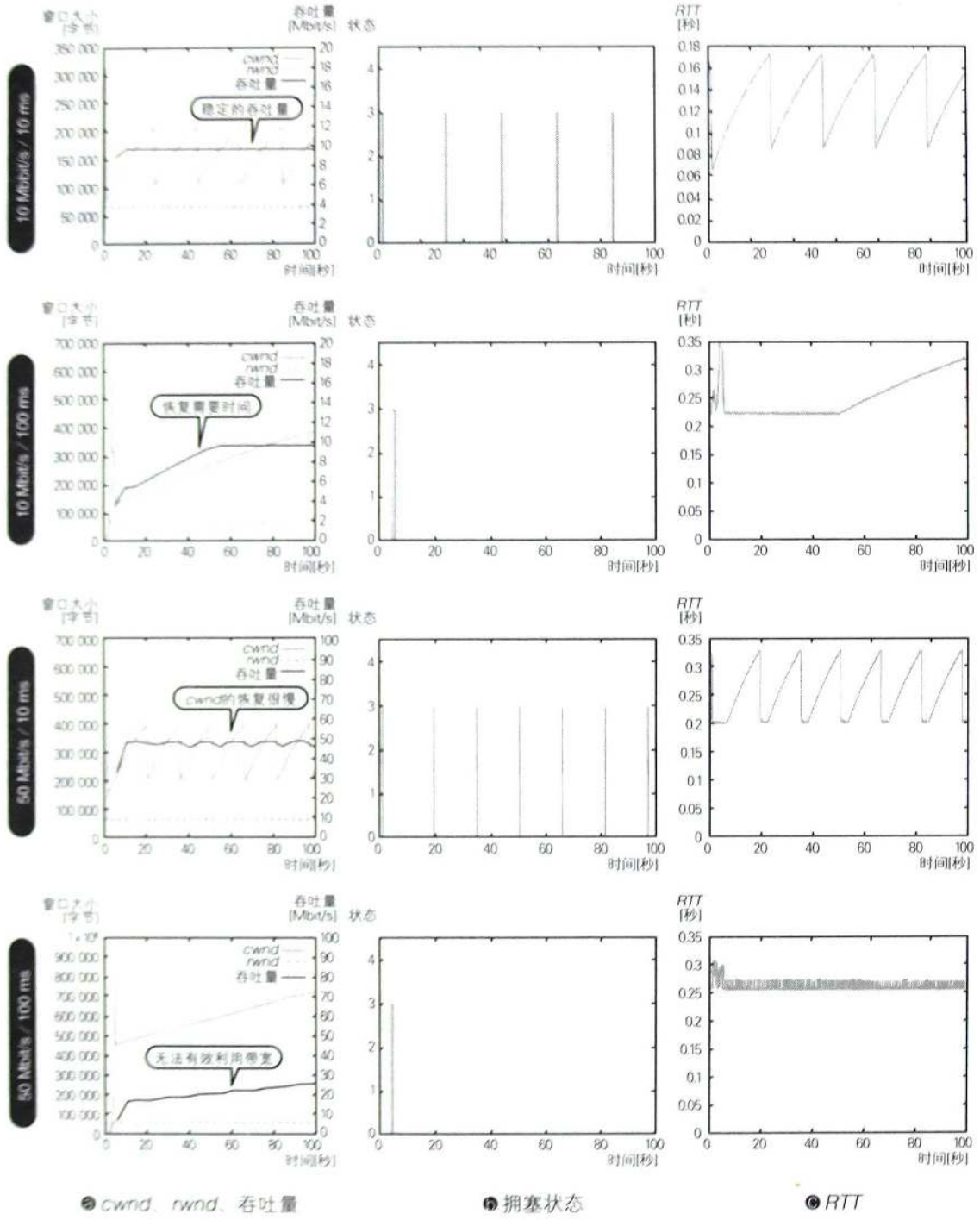


图5.8 模拟结果（实验1）

接下来，在**宽带、低时延环境**（图5.8的50 Mbit/s、10 ms）下，重复运行AIMD算法，我们稍微可以看到其对吞吐量的影响。究其原因，主要是随着带宽的增大，拥塞窗口大小恢复的速度也变得更慢，而拥塞窗口大小减小也意味着吞吐量的下降。

最后，在**宽带、高时延环境**（图 5.8 的 50 Mbit/s、100 ms）下，可以看到在第 100 秒前后吞吐量只达到了 25 Mbit/s，完全没能有效地利用带宽。这主要是因为，虽然逐渐增大了拥塞窗口大小，但由于 *RTT* 较大，所以增大的过程需要花费相当长的时间。这样一来，好好的宽带环境就白白浪费了。

从以上模拟结果中，我们看到了长肥管道下 NewReno 所表现出来的问题。

HighSpeed 与 Scalable 针对长肥管道的拥塞控制

如前文所述，一旦将过去常用的 NewReno 放在宽带、高时延的环境下，就会出现无法有效利用宽带的问题。

针对此问题，多个面向长肥管道的拥塞控制算法被提了出来。接下来要介绍的 HighSpeed（HighSpeed TCP，HSTCP）和 Scalable 是其中的代表。第 4 章已经介绍了这两种算法，因此这里就不再详述。下文将一边回顾复习，一边概述一下这两个算法，并采用与前面相同的条件进行模拟，然后将结果与 NewReno 进行对比。

首先，HighSpeed[①] 算法根据拥塞窗口大小调整 AIMD 控制（公式 5.1）中的变量 α 和 β。当拥塞窗口大小小于一定的阈值时，α 和 β 的值与 NewReno 中的一样，而当拥塞窗口大小超过了这个阈值时，α 和 β 的值就用拥塞窗口大小的函数来表示。此时，拥塞窗口大小越大，α 的值就越大，而对应的 β 值就越小。通过此修改，就可以实现拥塞窗口大小增大和减小之后的快速恢复。

接下来，在 Scalable[②] 中，将公式 5.1 中的变量 α 的值设为 0.01，拥塞窗口大小的增加值设为常量。此修改主要是为了解决过去的控制算法中拥塞窗口大小越大，拥塞窗口大小的增大速度就越慢的问题。

此外，Scalable 还会将丢包时拥塞窗口大小的减小量调整为现在的

① 《面向大型拥塞窗口的高速 TCP》（RFC 3649）。

② Tom Kelly. Scalable TCP: improving performance in highspeed wide area networks [J]. ACM SIGCOMM Computer Communication Review，Vol 33，No 2，pp 83-91，2003.

1/8。这相当于将公式5.1中的变量β的值设为0.875。显然，与NewReno中β=0.5相比，这里的目的是将拥塞窗口大小保持在一个较大的值上。

HighSpeed与Scalable的模拟

接下来，我们也通过模拟来看一下HighSpeed与Scalable的拥塞窗口大小控制算法。模拟使用的基本条件与之前的实验1相同，但这里将拥塞控制算法设置为ns-3中内置的`TcpHighSpeed`和`TcpScalable`。这里将本次的模拟条件称为“实验2”。

打开ns-3的根目录，输入以下命令，运行实验2。

```
$ ./scenario_5_2.sh
```

※保存位置：data/chapter5目录下（测试数据：05_xx-sc2-*.data；图表：05_xx-sc2-*.png）

模拟运行结果 确认HighSpeed和Scalable的问题

首先，图5.9展示是使用HighSpeed时的模拟运行结果。

从图中我们一眼就可以看出来，拥塞窗口大小的增大速度，与之前的NewReno算法相比明显更快。结果就是，在**低时延环境**下AIMD的控制周期变得非常短，不仅如此，在**窄带、高时延的环境**（10 Mbit/s、100 ms）下，吞吐量到达线路传输率花费的时间也被控制在20秒以内。

此外，即使在**宽带、高时延环境**（50 Mbit/s、100 ms）下，拥塞窗口大小也增大得很快，吞吐量在几十秒内便可以恢复。然而，在此模拟条件下，由于网关的最大队列长度设置得比较小，所以拥塞窗口大小一旦变大，很快就会发生丢包，导致吞吐量无法一直保持在较高的状态。

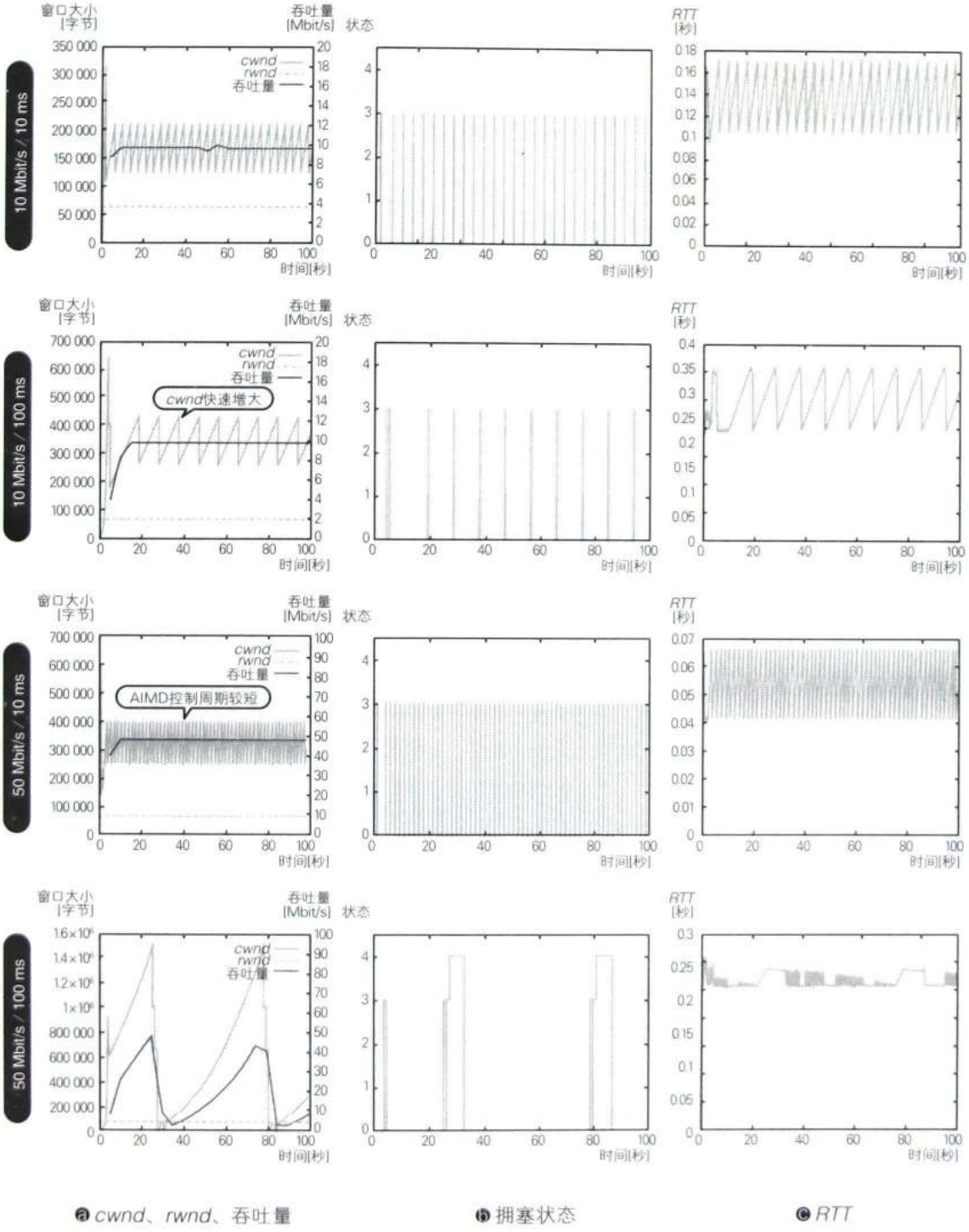


图 5.9 模拟结果（实验 2：HighSpeed）

接下来，图 5.10 展示的是使用 Scalable 时的模拟运行结果。这里的拥塞窗口大小的增大速度，与 NewReno 的数据相比明显更快。Scalable 的整体情况与 HighSpeed 类似，但其 AIMD 控制周期更短，在**宽带、高时延环**

境（50 Mbit/s、100 ms）下的吞吐量同样恢复得很快。

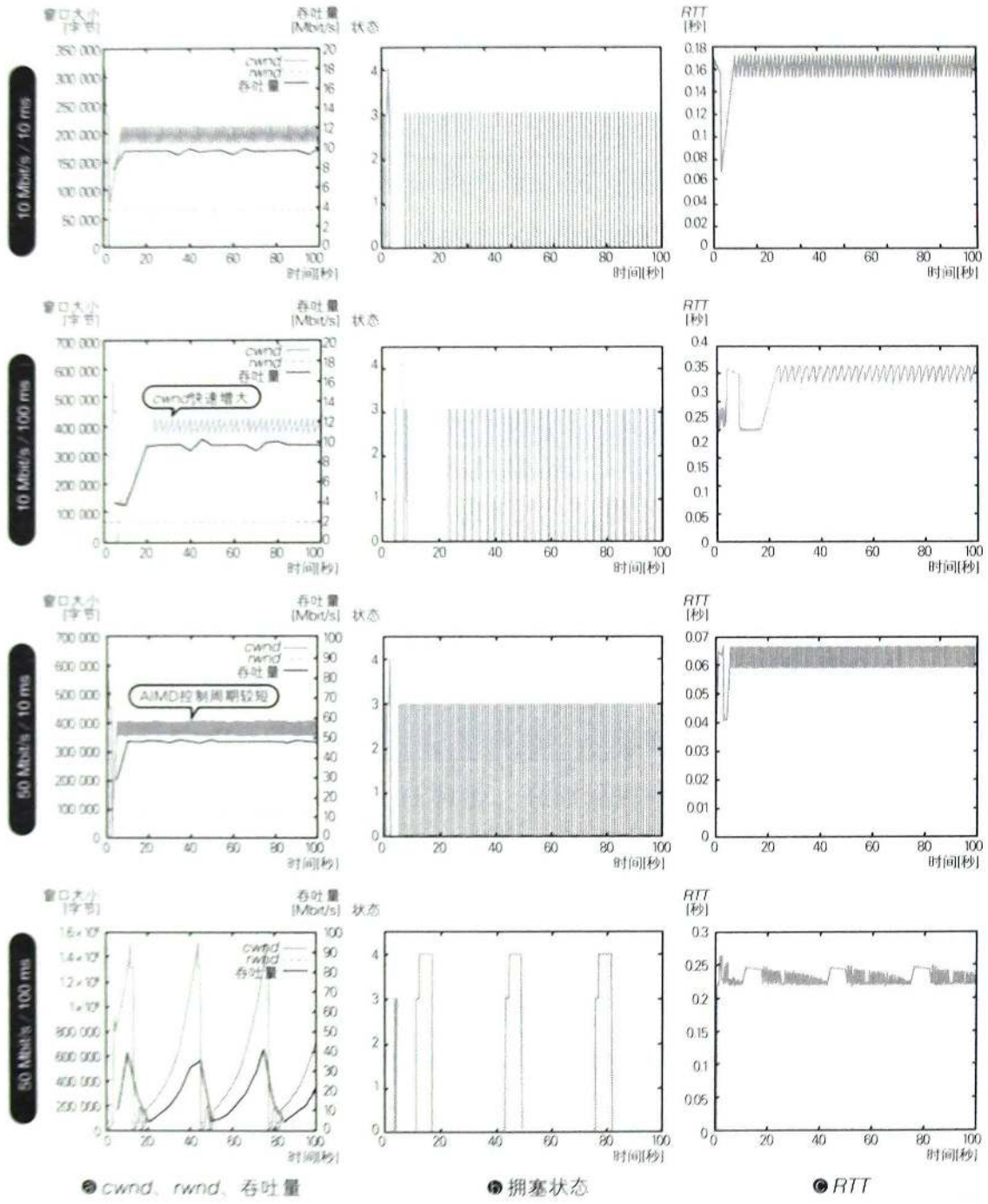


图5.10 模拟结果（实验2：Scalable）

亲和性 HighSpeed 与 Scalable 的问题❶

从前面的数据可以看出，使用针对长肥管道开发的 HighSpeed 和 Scalable 算法，便可以在宽带、高时延环境下实现高效率的 TCP 通信。

当 TCP 网络流只有一个时，显然可以认为前面问题已经得到了解决。然而，实际上，当这两种算法与过去的 NewReno 一起使用时，它们会挤占 NewReno 的带宽，带来问题。究其原因，主要是这两种算法与 NewReno 相比，拥塞窗口大小更容易维持在一个较大的值上[①]。

——通过模拟验证亲和性问题

接下来，我们就通过模拟来观察一下上述问题。

模拟条件如图 5.11 所示。其中有两个发送节点连接网关，在网关与接收节点之间发生拥塞。将发送节点❶设置为 NewReno 算法，发送节点❷设置为 HighSpeed 或者 Scalable 算法。为了避免 NewReno 处于不利的环境下，这里使用与实验 1 中宽带、低时延环境（50 Mbit/s、10 ms）相同的链路速度和传播时延作为参数。

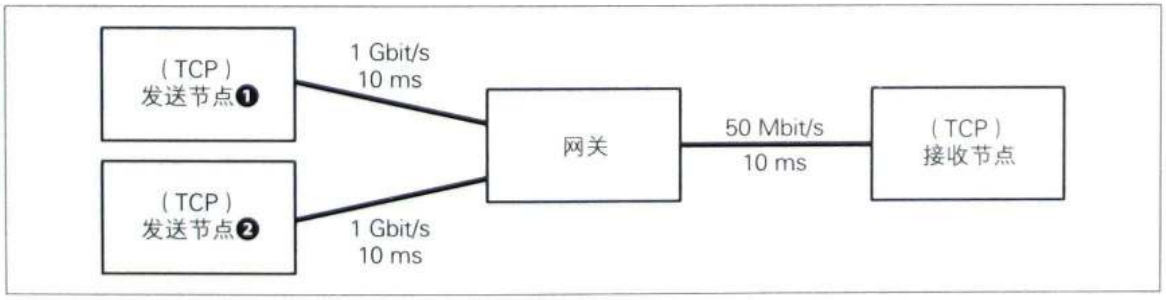


图 5.11 模拟条件（实验 3）

这里将本次的模拟条件称为“实验 3”，通过以下命令来运行它。

```
$ ./scenario_5_3.sh
※保存位置：data/chapter5目录下（测试数据：05_xx-sc3-*.data，图表：05_xx-sc3-*.png）
```

① 这种性质称为积极性。

模拟运行结果 过于积极，导致NewReno算法无立身之地（存在亲和性问题）

在模拟中，各个TCP网络流的吞吐量测试结果如图5.12所示。从图中可以看出，无论是使用HighSpeed还是Scalable，它们的网络流都会占有全部的50 Mbit/s带宽，使得NewReno再无容身之地。

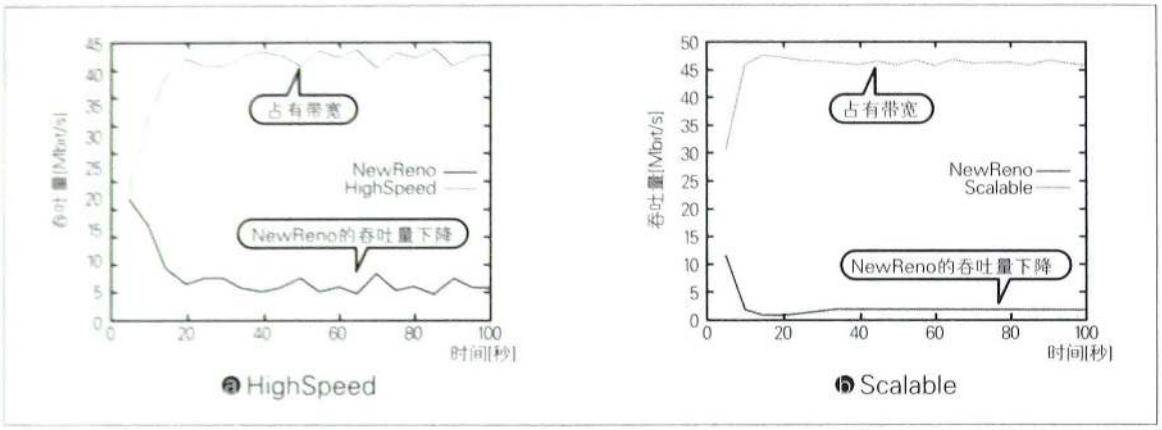


图5.12 模拟结果（实验3）

从结果可以看出，HighSpeed和Scalable算法过于积极，很难与NewReno共存。

网络是无数人或者说是无数台设备所共有的，一个算法能否与过去的算法共存，是非常重要的。换句话说，如果使用了新拥塞控制算法的设备接入网络之后，网络上原本正在使用旧的拥塞控制算法通信的大量现存设备就会因为新设备的接入而无法通信，那么对于这样的问题，我们必须予以规避。此外，想要将新算法安装到种类繁多、数量巨大的所有现存旧设备上，显然是不现实的。

前面我们提过，这样的思考视角便叫作与现有算法的**亲和性**，或者也可以称为公平性。因此，我们需要寻求一种与目前使用最为广泛的NewReno算法更具亲和性的新算法。

RTT公平性 HighSpeed和Scalable的问题2

此外，还有一个重要的问题，那便是***RTT*公平性**。*RTT*公平性这个

概念指的是 *RTT* 不同的网络流之间的吞吐量的公平性。HighSpeed 与 Scalable 算法存在一个问题，那便是当与 *RTT* 不同的网络流共享瓶颈链路时，*RTT* 较小的网络流会占据 *RTT* 较大的网络流的生存空间。

——通过模拟验证 RTT 公平性问题

下面，我们就通过模拟来确认一下这个问题。图 5.13 展示的是模拟条件。两个发送节点连接到网关上，这与之前的模拟相同。不同的地方是，两个发送节点设置了不同的传播时延，以便拉开 *RTT* 的差距，其中发送节点 ❶ 的传播时延设置为 10 ms，而发送节点 ❷ 的传播时延设置为 100 ms。

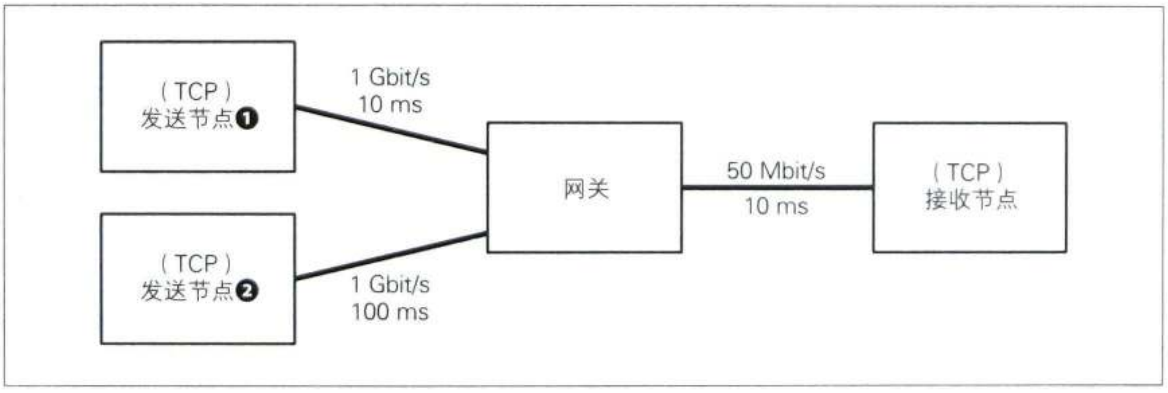


图 5.13 模拟条件（实验 4）

这里将本次的模拟条件称为“实验 4”，通过以下命令来运行它。

```
$ ./scenario_5_4.sh
```

※保存位置：data/chapter5目录下（测试数据：05_xx-sc4-*.data，图表：05_xx-sc4-*.png）

——模拟运行结果 RTT 较小的网络流独占网络，存在 RTT 公平性问题

实验 4 运行后，各个 TCP 网络流的吞吐量测试数据如图 5.14 所示。无论是使用 HighSpeed 还是 Scalable，*RTT* 较小的网络流都会独占整个 50 Mbit/s 的带宽，而 *RTT* 较大的网络流几乎无法通信。

如模拟结果所示，HighSpeed 与 Scalable 的 *RTT* 公平性很差。这两种算法为了提高**扩展性**（scalability）而优待拥塞窗口较大的网络流。在这样的控制逻辑下，*RTT* 公平性显然会下降。

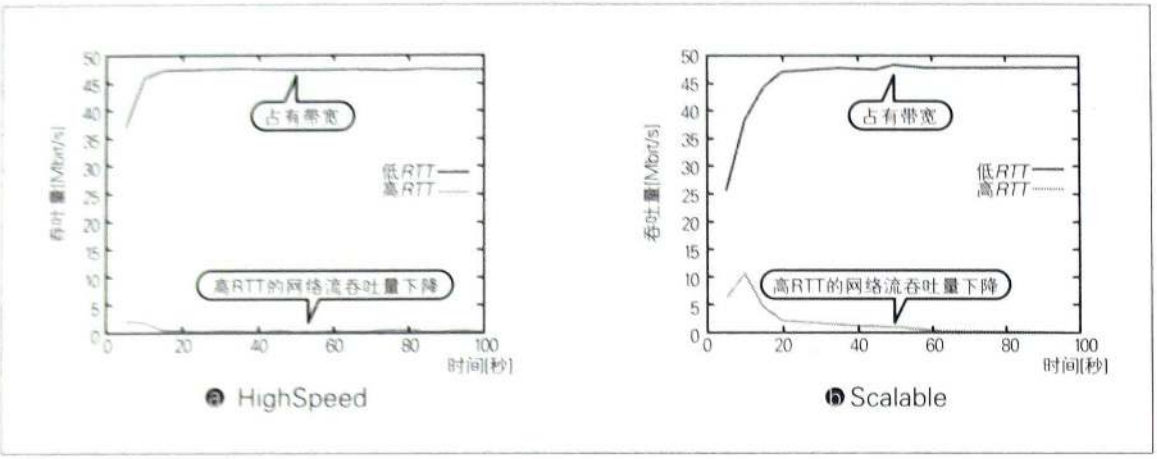


图5.14 模拟结果（实验4）

下一节介绍的BIC算法则是一个以提高*RTT*公平性为主要目的被开发出来的拥塞控制算法。

5.3 BIC

以宽带、高时延环境为前提的算法

BIC（BIC-TCP）算法是一个兼具稳定性、扩展性、*RTT*公平性，以及与现有算法的亲和性等多个优点的拥塞控制算法。不仅如此，BIC也是如今主流的拥塞控制算法之一CUBIC的基础，十分重要。

BIC是什么

BIC公布于2004年[1]。此后，BIC在Linux 2.6.8到2.6.18系统中一直常驻，直到CUBIC出现之后才被其取代。

BIC也是为了满足在宽带、高时延环境下的使用需求而开发的拥塞控

[1] Lisong Xu, Khaled Harfoush, Injong Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks [C]. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, pp.2514-2524, 2004.

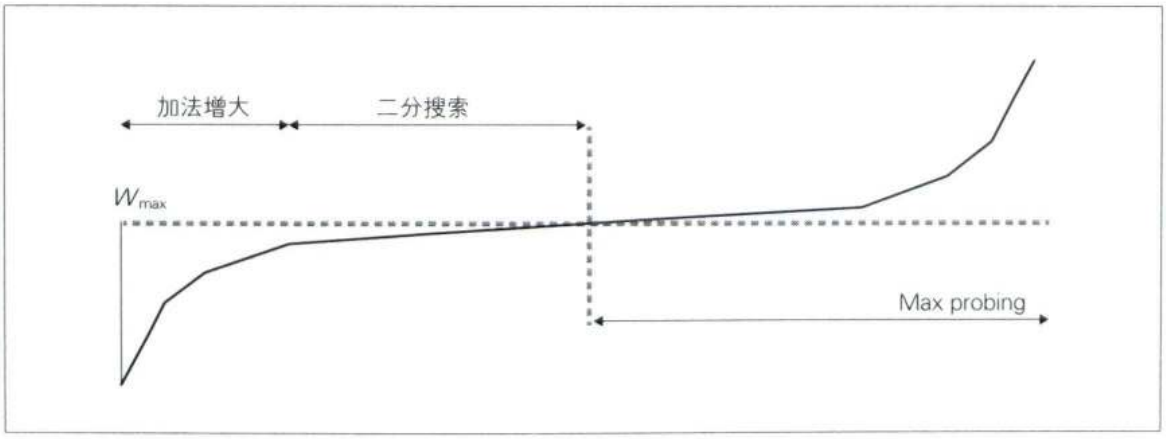
制算法，其稳定性和扩展性都很优秀。此外，BIC 算法也考虑了之前介绍的与现有算法的亲和性。

开发 BIC 的最大目的是改善 *RTT* 公平性。在高速网络中，拥有尾丢包（详见 6.1 节）队列的路由器会同时废弃多个连接中的数据包，包含此问题在内的 *RTT* 公平性问题逐渐暴露出来，而 BIC 正是以解决此类问题为首要目的而设计出来的。

第 4 章已经介绍过 BIC 算法，因此这里不再详细介绍。接下来，我们一边回顾 BIC 算法的概要与流程，一边结合实际的模拟测试详细介绍上面提出的问题。

增大拥塞窗口大小的两个阶段 加法增大和二分搜索

BIC 的拥塞窗口大小的增大情况如图 5.15 所示。BIC 算法通过**加法增大**和**二分搜索**（binary search）两个阶段来增大拥塞窗口大小。



※ 出处：Injong Rhee，Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant [C]. PFLDnet，Figure 1(a) BIC-TCP window growth function，2005

图 5.15 BIC 的拥塞窗口大小的增大

BIC 以发生丢包时的拥塞窗口大小（W_{max}）为目标，根据当前的拥塞窗口大小切换阶段。换句话说，当拥塞窗口大小较小时，使用加法增大的方法快速增大拥塞窗口大小，以提高扩展性和 *RTT* 公平性。然后，当拥塞窗口大小变大后，通过二分搜索法缓慢增大拥塞窗口大小，以避免出现过多的丢包。