此外，当拥塞窗口大小超过 W_{max} 之后，会进入 Max probing（最大值搜索）阶段。在这个阶段，拥塞窗口大小的增长函数的曲线，与拥塞窗口大小增长到 W_{max} 之前的曲线完全对称，同时拥塞窗口大小遵循新的曲线继续增长，直到发现下一次丢包。

BIC的拥塞窗口大小的变化情况

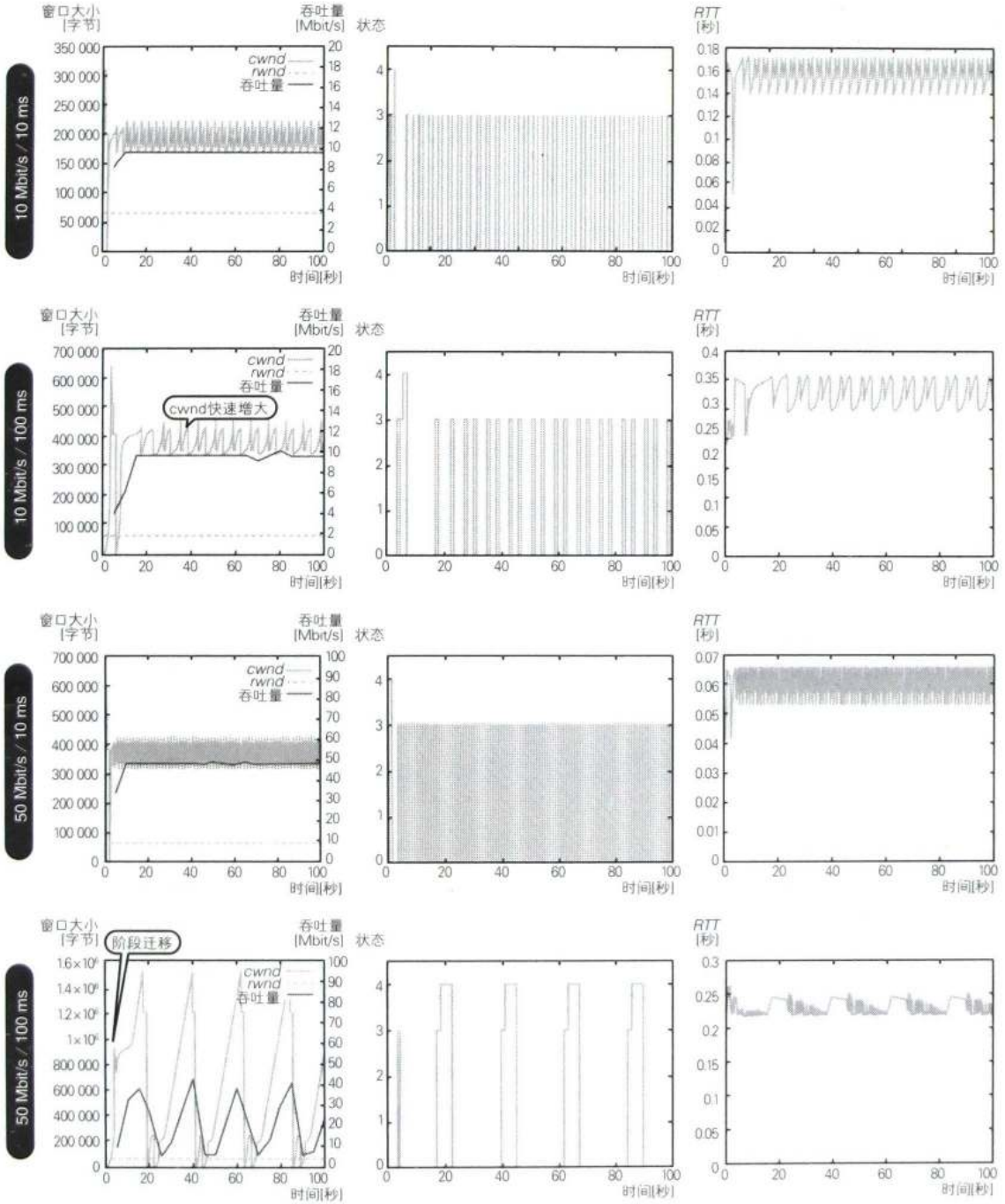
接下来，我们通过模拟来看一下BIC的拥塞窗口大小的变化情况。基本的模拟条件和实验1一致，不同的是这里将拥塞控制算法设置为ns-3中自带的TcpBic。

这里将本次的模拟条件称为“实验5”。打开ns-3的根目录，通过以下命令来运行实验5。

```
$ ./scenario_5_5.sh
```

※保存位置 data/chapter5目录下（测试数据：05_xx-sc5-*.data；图表：05_xx-sc5-*.png）

使用BIC时的模拟运行结果如图5.16所示。从图中可以看出，无论在哪种环境下，拥塞窗口大小都呈高速增大的态势。毫无疑问，BIC表现出了与HighSpeed和Scalable相似的性能。图中也能看到BIC的关键特征，即加法增大、二分搜索和Max probing三个阶段的迁移，尤其是在宽带、高时延的环境（50 Mbit/s、100 ms）下，在第10秒前后的时候最为明显。从结果可以看出，BIC拥有很强的可扩展性。



ⓐ *cwnd*、*rwnd*、吞吐量　　ⓑ 拥塞状态　　ⓒ *RTT*

图 5.16 模拟结果（实验 5）

BIC的问题

目前，BIC已经被下一节将要介绍的CUBIC所替代。CUBIC拥塞控制算法继承了BIC的优点，并进行了进一步的改进。因此为了避免重复，下一节将使用CUBIC进行*RTT*公平性和亲和性等的数据测试。

BIC不仅提高了扩展性，也就是宽带、高时延环境下的效率，还提升了*RTT*公平性和与现有算法的亲和性，但它也被指出存在若干问题。这便是CUBIC诞生的背景。

在这些问题中，首先便是BIC在窄带、低时延的网络环境下会不当地占有带宽。此外，由于BIC的拥塞窗口大小的增大算法分为加法增大、二分搜索和Max probing等多个阶段，所以协议的解析十分复杂，而且性能预测和网络设计也十分困难。

为了解决这些问题，下一节将要介绍的CUBIC被开发了出来。

5.4 CUBIC的机制

使用三次函数大幅简化拥塞窗口大小控制算法

CUBIC默认搭载在Linux中，是主流拥塞控制算法之一。它通过简单的算法实现了BIC的核心优点——扩展性、*RTT*公平性和亲和性。

CUBIC的基本情况

CUBIC默认搭载在Linux 2.6.19以后的版本中，毫无疑问，目前它已经成为主流的拥塞控制算法之一。作为BIC的改良版本，它大幅简化了BIC中拥塞窗口大小的复杂控制机制。

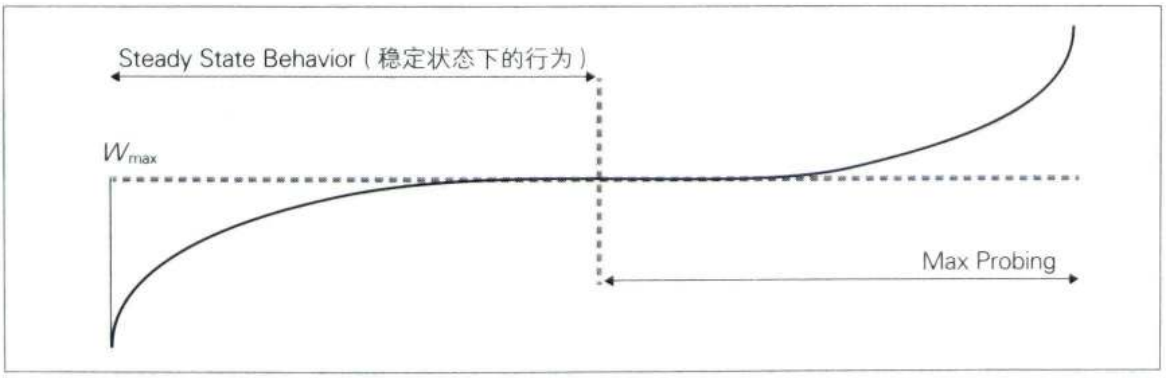
CUBIC通过将前面图5.15中所展示的BIC拥塞窗口大小的增长函数替换为三次函数（cubic function），省去了阶段切换，实现了算法的简化。

最终的结果就是，拥塞窗口大小的增大量只由两个连续的拥塞事件之间的时间间隔[①]决定，这一点成为 CUBIC 的显著特点。换句话说，这意味着“拥塞窗口大小的增大速度与 *RTT* 无关”，好处就是能够提升 *RTT* 公平性。此外，按照设计，CUBIC 在 *RTT* 较小的情况下会控制拥塞窗口大小的增加量。因此很显然，与现有 TCP 的亲和性较高也是 CUBIC 的优点。

下文将结合实际的模拟数据详细介绍 CUBIC 窗口控制算法，以及 CUBIC 带来的具体效果。另外，下一节将详细介绍 CUBIC 拥塞控制算法的具体内容。

窗口控制算法的关键点

图 5.17 展示的是 CUBIC 拥塞窗口大小的增长函数。从图中可以看出，此图形与前面的图 5.15 中展示的 BIC 拥塞窗口大小的增长函数极为相似。换句话说，CUBIC 使用以“快速恢复开始后的经过时间”为自变量的三次函数，近似模拟了 BIC 中复杂的拥塞窗口大小控制。



※ 出处：Sangtae Ha，Injong Rhee，Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant [C]. ACM SIGOPS operating systems review，Figure 1(b) BIC-TCP window growth function，vol.42，no.5，pp.64-74，2008.

图 5.17 CUBIC 的拥塞窗口大小的增大

实现此结果的拥塞窗口大小增长函数可以用公式 5.2 表示。

$$w(t) = C(t-K)^3 + W_{max} \qquad （公式 5.2）$$

① 这个时间间隔的起点，其实就是丢包后出现的快速恢复阶段开始的时间点。

在这个公式中，W_{max}代表发现丢包时的拥塞窗口大小。C是CUBIC参数，而t是从快速恢复阶段开始的经过时间。还有，K是决定拥塞窗口大小增大速度的参数，可以用下面的公式5.3来计算。此外，公式5.3中的β代表的是丢包时拥塞窗口大小的减小量。

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \qquad \text{（公式 5.3）}$$

CUBIC的拥塞窗口大小的变化情况

接下来，我们通过模拟来看一下CUBIC的拥塞窗口大小的控制情况。基本的模拟条件与实验1一致，不过这里将拥塞控制算法设置为TcpCubic。TcpCubic并没有搭载在当前的ns-3的官方发布版本中，不过TcpCubic模块已经在Web上公开发布，本书的模拟环境中已经安装了它。

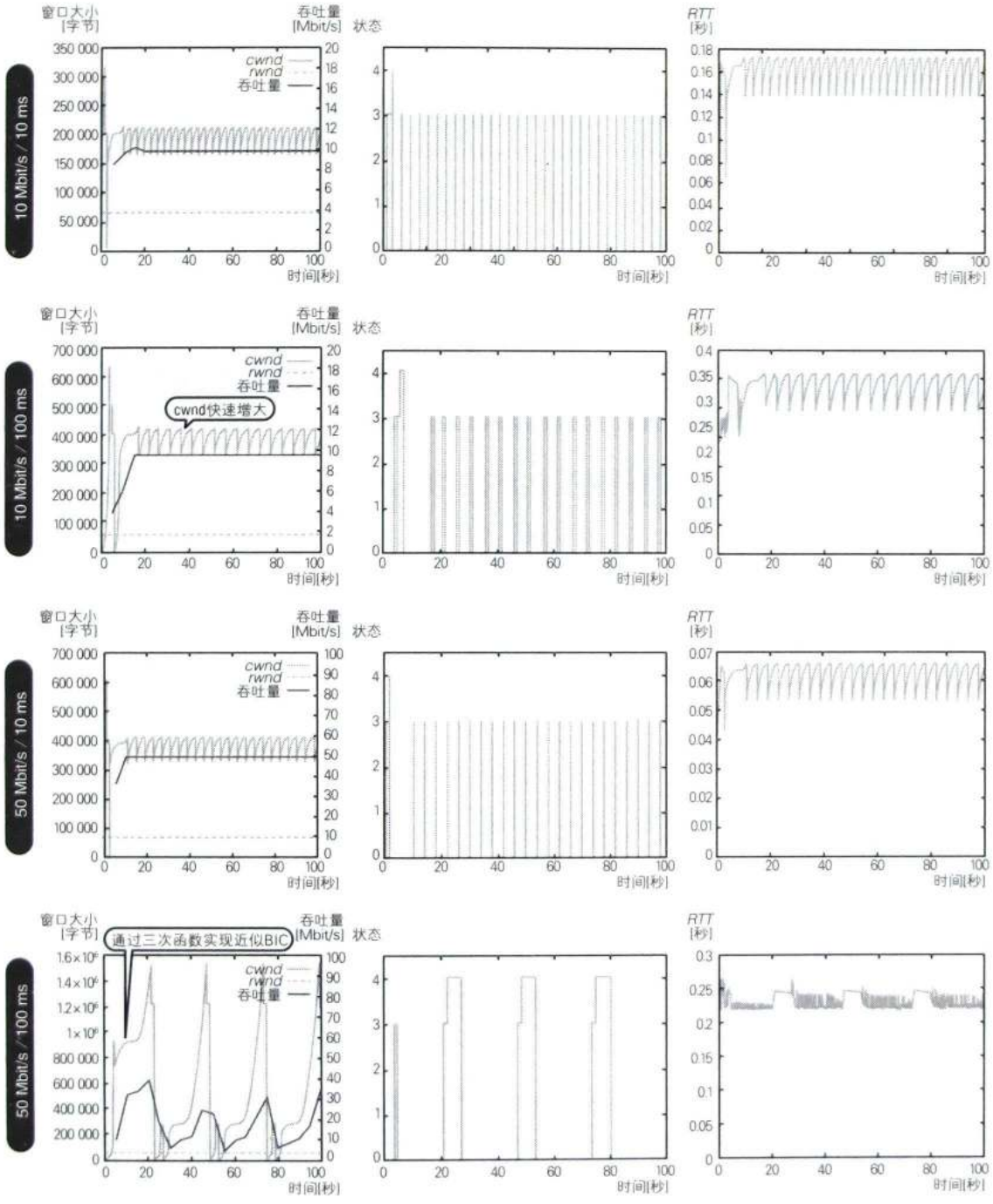
这里将本次的模拟条件称为“实验6”，通过以下命令来运行它。

```
$ ./scenario_5_6.sh
```

※保存位置：data/chapter5目录下（测试数据：05_xx-sc6-*.data；图表：05_xx-sc6-*.png）

使用CUBIC时的模拟运行结果如图5.18所示。

我们从图中大致可以看到，CUBIC的行为与BIC基本相同，两者都会快速地增大拥塞窗口大小，合理、有效地利用带宽。另外，从图中还可以看到CUBIC的核心特征——三次函数的近似表现，尤其是在宽带、高时延的环境（50 Mbit/s、100 ms）下，在第10秒前后较为明显。从这个结果可以看到CUBIC在扩展性上的优异表现。



ⓐ *cwnd*、*rwnd*、吞吐量　　ⓑ 拥塞状态　　ⓒ *RTT*

图 5.18 模拟结果（实验 6）

模拟结果中展现出来的高亲和性

接下来，我们通过模拟来看一下CUBIC的特点，也就是与现有TCP的高亲和性。模拟条件与实验3基本一致（前面的图5.11），这次的测试要比较NewReno和CUBIC的吞吐量。

这里将本次的模拟条件称为“实验7”，通过以下命令来运行它。

```
$ ./scenario_5_7.sh
```

※保存位置：data/chapter5目录下（测试数据：05_xx-sc7-*.data；图表：05_xx-sc7-*.png）

运行实验7，并收集各个TCP网络流的吞吐量数据，结果如图5.19所示。

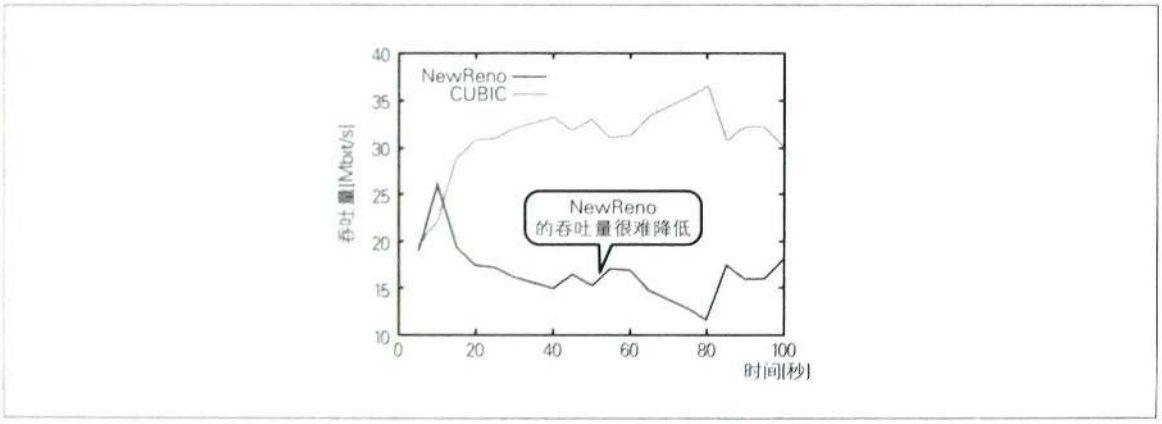


图5.19 模拟结果（实验7）

从结果可以看出，与HighSpeed和Scalable相比，NewReno的吞吐量有很大幅度的改善。也就是说，CUBIC通过抑制积极性提高了与现有TCP网络流的亲和性。

模拟结果中展现出来的RTT公平性

接下来，我们通过模拟来看一下CUBIC的另一个特点——*RTT*公平性的提升。模拟条件与实验4（前面的图5.13）基本一致，这里将收集各个CUBIC网络流的吞吐量数据并进行比较。

这里将本次的模拟条件称为“实验 8”，通过以下命令来运行它。

```
$ ./scenario_5_8.sh
```

※保存位置：data/chapter5目录下（测试数据：05 xx-sc8-*.data，图表：05 xx-sc8-*.png）

运行实验 8，并收集各个网络流的吞吐量数据，最终汇总如图 5.20 所示。

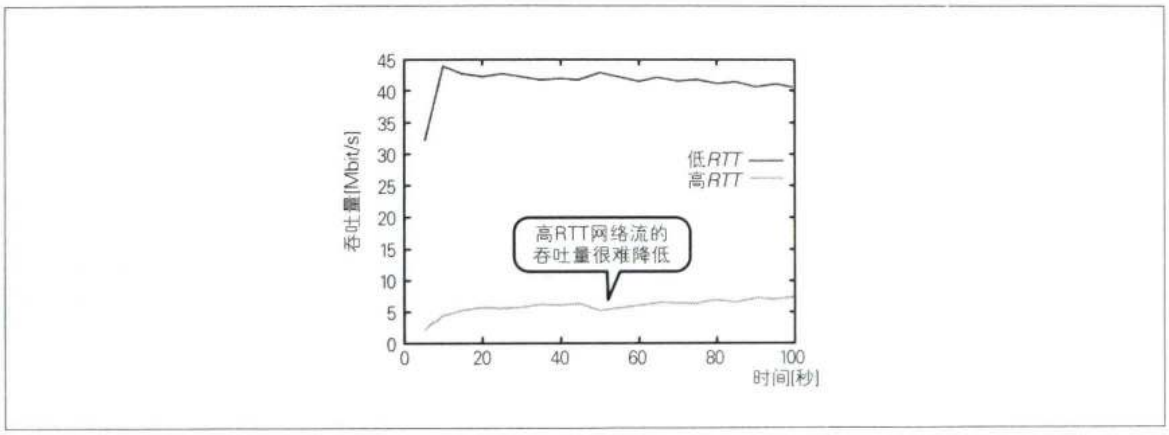


图 5.20 模拟结果（实验 8）

在本次的模拟条件下，*RTT* 之间的差别非常大，因此无法完全实现 *RTT* 的公平化。但是，与刚才使用 HighSpeed 和 Scalable 时相比（前面的图 5.14），高时延网络流的吞吐量显然得到了大幅提升。这也是 CUBIC 的“拥塞窗口大小的增大速度不依赖于 *RTT*”这一特点的最好表现。

窄带、低时延环境下的适应性

接下来，我们再来确认一下 BIC 在窄带、低时延环境下拥塞窗口大小急速增大的问题，以及使用 CUBIC 时对它的抑制效果。模拟环境如图 5.21 所示。这里连接网关的虽然也是两个发送节点，但从网关到接收节点之间是 10 Mbit/s 的窄带环境。发送节点①设置为 NewReno，发送节点②设置为 BIC 或者 CUBIC。

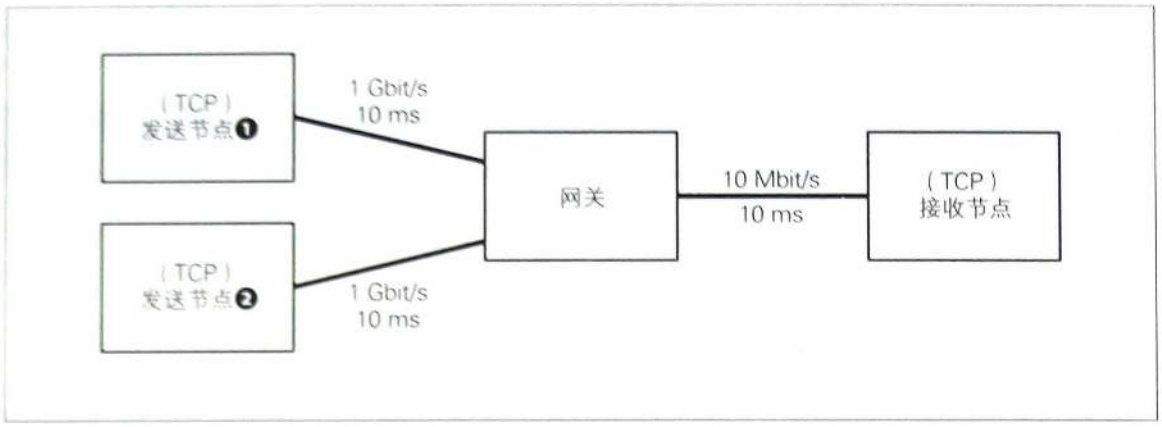


图5.21 模拟条件（实验9）

这里将本次的模拟条件称为“实验9”，通过以下命令来运行它。

```
$ ./scenario_5_9.sh
```
※保存位置：data/chapter5目录下（测试数据：05_xx-sc9-*.data；图表：05_xx-sc9-*.png）

运行实验9，并收集各个TCP网络流的吞吐量数据，结果如图5.22所示。

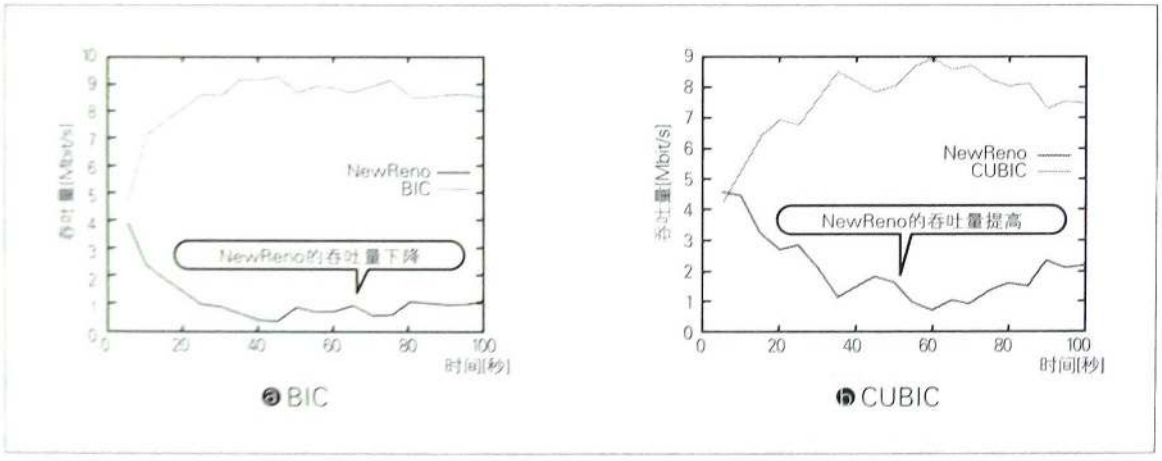


图5.22 模拟结果（实验9）

从图中可以看出，BIC占据了大部分的带宽，导致NewReno的吞吐量极小。而在使用CUBIC时，NewReno的吞吐量得到了改善，提升了接近一倍。从结果来看，显然CUBIC解决了BIC在窄带、低时延环境下占有带宽的老问题。

CUBIC 的问题

从前面的结果来看，CUBIC 的出现解决了一系列问题。其中主要有，现有算法与宽带、高时延环境的适应性（**扩展性**）这个老问题，还有不同 *RTT* 的网络流之间吞吐量公平性的问题，以及与现有拥塞控制算法的亲和性问题等。

但是，CUBIC 并不能解决所有问题。从拥塞窗口大小增长函数可以看出，只要网络上不发生拥塞，拥塞窗口大小就会一直增大，因此只要不发生拥塞，链路中路由器等设备的缓冲区就会被不停地填充，直到发生丢包为止。因此，当网络中的缓冲区较多时，就会导致**队列时延**增大。

针对此问题，这里同样使用模拟来确认一下。基本的模拟条件同实验 7 的宽带、低时延环境（50 Mbit/s、10 ms）一致，然后将网关的最大队列长度调整为 100 个数据包、10 000 个数据包，并收集对应的 *RTT* 数据。

这里将本次的模拟条件称为“实验 10”，通过以下命令来运行它。

```
$ ./scenario_5_10.sh
※保存位置：data/chapter5目录下（测试数据：05_xx-sc10-*.data；图表：05_xx-sc10-*.png）
```

收集到的 *RTT* 数据如图 5.23 所示。

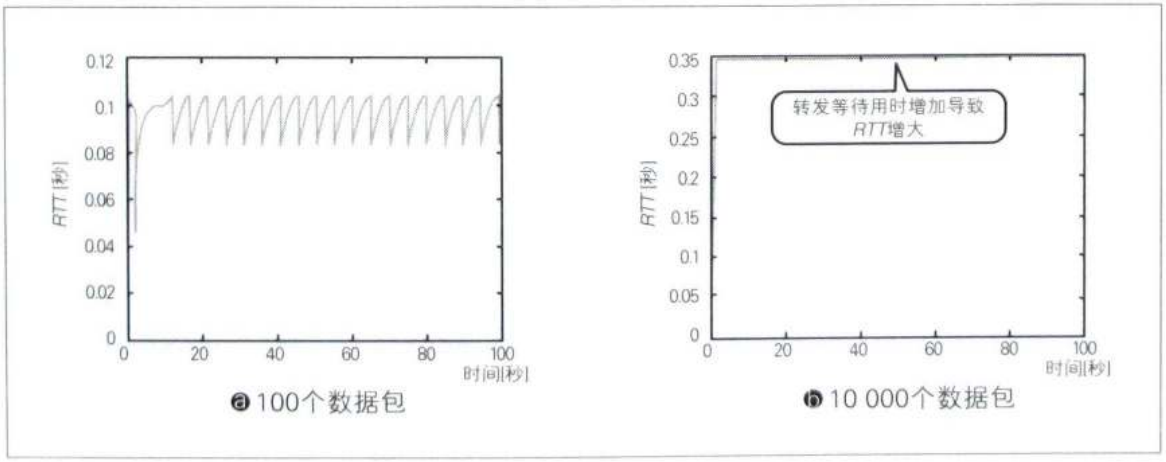


图 5.23 模拟结果（实验 10）

当最大队列长度设置为 100 个数据包时，*RTT* 最大也只有 0.1 秒左右，而当最大队列长度设置为 10 000 个数据包时，*RTT* 则一直保持在 0.35 秒。

这个差距代表的是发送队列中转发等待用时的增加。在本次的模拟环境下，只有1个网关的转发等待用时增加了，因此影响有限，然而如果传输链路上多台路由器、交换机等设备上的转发等待用时同时增加，那么最终给 *RTT* 和吞吐量带来的影响恐怕会十分巨大。

为了应对这种时延增大的问题，基于延迟的拥塞控制算法被开发了出来。下一章将详细介绍这类算法。

在介绍此类算法之前，我们将在下一节使用伪代码详细介绍CUBIC算法的具体内容。

5.5 使用伪代码学习 CUBIC 算法

主要的行为与处理过程

本节将使用伪代码分步骤介绍 Linux CUBIC algorithm v2.2[1] 算法在初始化、丢包和超时时的各种情况。

初始化

首先是初始化处理，相应的伪代码如下所示。

```
tcp_friendliness <- 1 // 提高TCP亲和性
β <- 0.2 // 丢包时拥塞窗口大小的减小量
fast_convergence <- 1 // 加速丢包时拥塞窗口大小的恢复过程
C <- 0.4 // CUBIC参数
cubic_reset() // 重置各种变量值
```

其中，β 如上文所述，是丢包时拥塞窗口大小的减小量，而 C 是CUBIC参数。此外，`tcp_friendliness` 是一个二进制变量，用于打开或关闭提高 TCP 亲和性的功能；`fast_convergence` 也是一个二进制变量，用

[1] Sangtae Ha, Injong Rhee, Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. ACM SIGOPS operating systems review, vol.42, no.5, pp.64-74, 2008.

于打开或关闭在丢包时加速拥塞窗口大小恢复的功能。cubic_reset() 是用于重置各个变量的函数，后文将具体介绍它。

收到 ACK 时的行为

在发送节点收到 ACK 时，算法按照以下伪代码进行处理。

```
If dMin then dMin <- min(dMin, RTT) // 计算代表RTT最小值的变量dMin
else dMin <- RTT
if cwnd ≤ ssthresh then cwnd <- cwnd + 1
// 当cwnd的值小于等于ssthresh时，对cwnd进行加法运算
else
  cnt <- cubic_update() // 调整cwnd的增大速度
  if cwnd_cnt > cnt then cwnd <- cwnd + 1, cwnd_cnt <- 0
  |_  else cwnd_cnt <- cwnd + 1
```

首先，计算代表 *RTT* 最小值的变量 dMin。这一值将在后面用于计算从拥塞事件开始后经过的时间。接下来，当拥塞窗口大小 cwnd 的值小于等于 ssthresh 时，对 cwnd 进行自增运算。而当 cwnd 大于 ssthresh 时，使用 cubic_update() 函数调整 cwnd 的增大速度。

丢包时的行为

在发生丢包时，算法按照以下伪代码进行处理。

```
epoch_start <- 0 // epoch开始时间的初始化处理
if cwnd < W_last_max and fast_convergence then W_last_max <- cwnd * (2 - β) / 2
                                          // 根据选项计算W_last_max的值
else W_last_max <- cwnd
ssthresh <- cwnd <- cwnd * (1 - β) // 根据β减小cwnd、ssthresh的值
```

首先，将此拥塞事件后的控制阶段（称为 epoch）的开始时间初始化。其次，计算要使用的变量 W_{last_max} 的值，它用于保持拥塞发生时的拥塞窗口大小。此时，如果 fast_convergence 选项处于激活状态，就调整这个值。也就是说，当 W_{last_max} 的值比上次小时，就认为网络拥塞变得更严重了，此时就将 W_{last_max} 的值减小，以提高效率。最后，根据变量 β 的

值减小 cwnd 和 ssthresh 的值。β 的值为 0.2，和初始化设置的一致。

超时时的行为

当发生超时时，算法将按照下面的命令运行 cubic_reset() 函数。

```
cubic_reset()
```

主要的函数与处理

下面，我们来看一下各个阶段出现的函数及其相应的处理过程。

cubic_update() 函数

cubic_update() 函数是用于调整拥塞窗口大小增大量的重要函数，会在收到 ACK 时被运行。实际上，其运行逻辑如以下伪代码所示。

```
ack_cnt <- ack_cnt + 1
if epoch_start ≤ 0 then
|    epoch_start <- tcp_time_stamp // epoch开始时的初始化处理
|    if cwnd < W_last_max then K <- sqrt[3]{(W_last_max-cwnd) / C}, origin_poin t <- W_last_max
|    else K <- 0, origin_point <- cwnd
|    ack_cnt <- 1
|    W_tcp <- cwnd
t <- tcp_time_stamp + dMin - epoch_start // epoch开始后经过的时间
target <- origin_point + C(t - K)^3 // 设置目标值
if target > cwnd then cnt <- cwnd/(target-cwnd)
// ✪ 在convage、convex模式下对cwnd进行加法运算
else cnt <- 100 * cwnd
if tcp_friendliness then cubic_tcp_friendliness()
// 开始进行提高TCP亲和性的处理
```

首先将公式 5.2 中的 $W(t)$ 的值设置为拥塞窗口大小增加量的目标值。此时，根据 cwnd 值的不同，分别使用 3 种模式。

第 1 种，当 cwnd 小于假定使用 Reno 和 NewReno 时的期望值时，就使用 TCP 模式。后文将介绍此模式下的行为。第 2 种，当 cwnd 小于

W_{last_max} 时，进入 concave 模式。除此以外的情况，则使用 convex 模式[①]。

在 concave 模式下，cwnd 的增加量是 $[W(t+RTT) - cwnd] / cwnd$，上述伪代码中使用✪进行了标注。在 convex 模式下，由于 cwnd 的值超过了上一次拥塞事件时的值，所以认为此时网络的拥塞情况得到了缓解，于是缓慢地增大拥塞窗口大小的增大量。此模式也被称为 Max probing 阶段（详见前面的图 5.17）。

cubic_tcp_friendliness 函数

cubic_tcp_friendliness() 函数在 cubic_update() 函数的最后运行，主要进行以下伪代码的处理过程。

```
Wtcp <- Wtcp + (3 β ack_cnt) / ((2 - β) cwnd) // Reno或NewReno的cwnd期望值
ack_cnt <- 0
if Wtcp > cwnd then // TCP模式
|   max_cnt <- cwnd/Wtcp-cwnd
|   if cnt > max_cnt then cnt <- max_cnt
```

其中，W_{tcp} 是使用 Reno 或 NewReno 时的拥塞窗口大小的期望值。当 cwnd 小于此值时，CUBIC 工作在 TCP 模式，并将 W_{tcp} 的值代入 cwnd。此方法可以提高拥塞算法与 Reno 或 NewReno 算法在一起时的公平性。

cubic_reset 函数

cubic_reset() 函数会在初始化和超时时被调用，用于重置各个变量的值。

```
Wlast_max <- 0, epoch_start <- 0, origin_point <- 0
dMin <- 0, Wtcp <- 0, ack_cnt <- 0
```

① concave 和 convex 的意思分别是凹和凸，它们分别表示 CUBIC 拥塞窗口大小的三次函数处于下凹或凸出的区间。

5.6 小结

本章结合模拟实验详细介绍了过去的拥塞控制算法在近些年来随着网络环境变化而不断浮现出来的问题，同时也介绍了新出现的CUBIC拥塞控制算法。下面简单回顾一下本章的内容。

随着互联网的普及，TCP也被广泛地推广开来。与此同时，Reno和NewReno拥塞控制算法作为标准也被广泛地应用开来。随后，近些年来，网络环境发生变化，传输速率提升、云服务普及等变化使得名为长肥管道的宽带、高时延环境逐步推广。在这种新环境下，Reno和NewReno算法出现了问题，主要包括快速恢复时吞吐量恢复所需时间变长、无法有效利用宽带环境等。

为了处理此类问题，HighSpeed和Scalable等一系列针对长肥管道的算法被提了出来。这些算法可以提升快速恢复阶段的拥塞窗口大小增大速度，最终使拥塞窗口大小保持在一个较大的值。然而，这些算法也存在一些问题，例如由于这种行为过于激进，所以这些算法与Reno和NewReno之间的亲和性较低，而且不同*RTT*的网络流之间的吞吐量公平性也较低等。

于是，为了解决这些问题，人们逐渐开始使用BIC。但是BIC也存在一些问题：在窄带环境或者低时延的网络环境下过于激进，而且控制过程过于复杂等。

经过一系列的发展，CUBIC被开发了出来，它拥有BIC的可扩展性、*RTT*公平性和与现有算法的亲和性等优势，并使用较为简单的算法将这些特性实现了出来。

CUBIC算法使用以快速恢复阶段开始后的经过时间为自变量的三次函数来确定拥塞窗口大小的增加量，这就使得拥塞窗口大小的增大速度不再依赖于*RTT*的值，提高了*RTT*的公平性。此外，它还会在*RTT*较小时控制拥塞窗口大小的增加量，这就解决了BIC存在的问题。CUBIC可以说是迄今为止所有基于丢包的拥塞控制算法的集大成之作，目前默认搭载