边调整数据的发送速度。BBR通过这种方法，实现了在网络可处理范围内的最大吞吐量。

接下来，我们详细看一下BBR对拥塞窗口大小的控制。

BBR的拥塞窗口大小控制机制 RTprop、BtlBw

BBR使用*RTprop*（Round-Trip propagation time，往返传播时延）和*BtlBw*（Bottleneck Bandwidth，瓶颈带宽）两个指标来调节拥塞窗口大小。

*RTprop*其实就是*RTT*，它是使用ACK计算出来的数值。*BtlBw*则是瓶颈链路的带宽，使用该指标，是因为就算TCP网络流在传输中会经过若干个链路，但决定其最终吞吐量的仍然是瓶颈链路的转发速度。

通过图来理解BBR inflight、BtlBw、RTprop、BDP

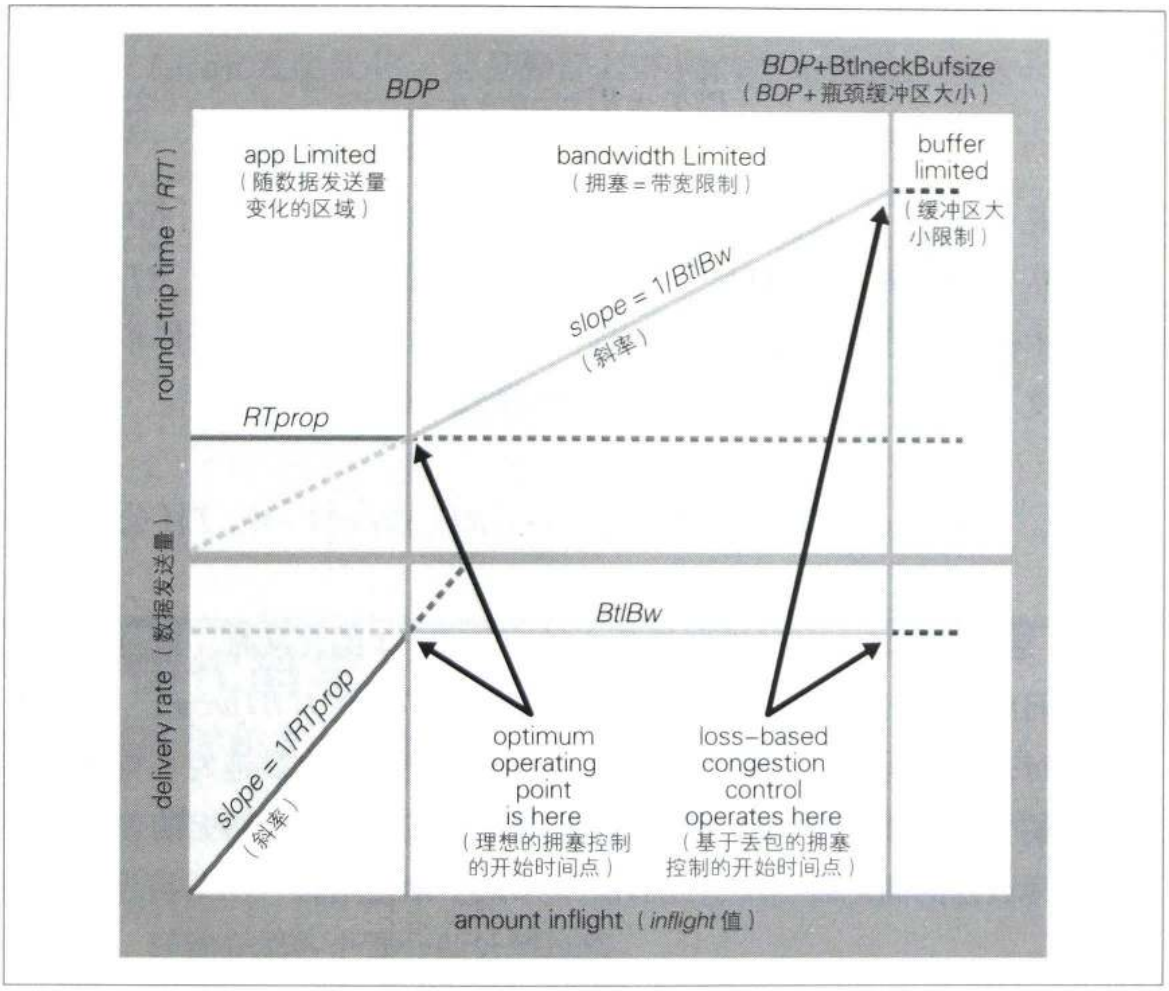
我们通过图6.9来进一步加深对BBR的理解。在此图中，横轴是"*inflight*"，它表示网络上正在发送的数据量；纵轴的上半部分是"*RTT*"，下半部分则是"数据发送量"。

如果从完全不发送数据的状态开始就缓慢增大*inflight*的值，那么刚开始的时候数据发送量会随之增加，但*RTT*不会变化。这只能说明数据包在空空如也的网络上，无须等待便会被转发出去。

接下来，当*inflight*的值超过一定值之后，数据发送量就不再增长。这意味着网络上某一段链路进入了拥塞状态，这段链路便是瓶颈链路，TCP的吞吐量会受这段瓶颈链路的制约。此时，即使增大*inflight*的值，数据发送量也不会超过此时的*BtlBw*的值，但*RTT*却会持续增大。这说明数据包堆积在瓶颈链路的缓冲区之中，队列时延在不断增大。接着，当*inflight*的值过大，超过缓冲区大小时，就会发生丢包。

在上述说明中，CUBIC等基于丢包的拥塞控制算法开始进行拥塞控制的时间点，是在*inflight*的值变大并超过缓冲区大小且发生了丢包时。但如果缓冲区较大，从数据发送量达到*BtlBw*开始直到发生丢包所花费的时间较长，那么此方法显然效率很低。在数据发送量达到*BtlBw*时，毫无疑问吞吐量完全不会再增长了。通俗一点来说，就算继续增大*inflight*的

值，也完全是白费功夫。



※ 出处：Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, et al. BBR Congestion Control [R]. Google Networking Research Summit, 2017.

图 6.9 数据发送量与吞吐量、RTT 的关系

因此，BBR 的目标便是“*inflight*=*BtlBw* × *RTprop*”这一状态，其值称为 ***BDP***（Bandwidth-Delay Product，带宽时延积）。根据计算，此时的数据发送量正好达到 *BtlBw* 这一阈值。

RTprop 的估算

我们已经知道了 *inflight* 的值最好是 *BtlBw* 和 *RTprop* 的乘积，但是如何知道 *BtlBw* 和 *RTprop* 的值呢？请看接下来的具体介绍。

首先介绍 *RTprop*。TCP 在发送某个数据包之后，计算从此时开始到

收到这一数据包对应的ACK为止所经过的时间，这便是*RTT*的值。此时，时刻*t*的*RTT*的值使用公式6.2来表示。

$$RTT_t = RTprop_t + \eta_t \quad （公式6.2）$$

在此公式中，η的值大于0，其代表的是由队列时延等引起的噪声，也就是传播时延等固定时延以外的一些可变参数。简而言之，*RTprop*表示的是由传播时延等组成的固定时延，只要网络拓扑等物理条件不变，这一数值就不会变。BBR中的*RTprop*的估算公式如公式6.3所示。

$$\widehat{RTprop} = RTprop + \min(\eta_t) = \min(RTT_t)\forall t \in [T - W_R, T] \quad （公式6.3）$$

W_R是时间窗口，一般设置为几十秒。公式6.3的含义是，取过去几十秒的时间中统计出来的*RTT*的值，以其中的最小值作为*RTprop*。

此时，将时间窗口分割为过去几十秒的单位值，主要是为了与网络拓扑结构的变化等相对应。换句话说，就是将由当前传输链路上的、除了缓冲区时延以外的固定时延所组成的值作为*RTT*来使用。

BtlBw的估算

接下来介绍*BtlBw*的估算方法。与*RTT*不同，TCP中没有计算瓶颈带宽的机制，但BBR可以使用*deliveryRate*（数据发送速率）估算瓶颈带宽。也就是说，BBR预先保存数据包的发送时间和数据发送量，然后在收到ACK时，与*RTT*值结合起来计算到达数据量。接下来，计算一定时间窗口内的到达数据量，这便是*deliveryRate*。最后，通过*deliveryRate*来估算*BtlBw*的值。*BtlBw*的估算公式如公式6.4所示。

$$\widehat{BtlBw} = \max(deliveryRate_t)\forall t \in [T - W_B, T] \quad （公式6.4）$$

W_B是时间窗口，通常被设置为*RTT*的6到10倍。设置时间窗口

主要是为了能与估算 *RTT* 时一样，适配网络拓扑的变化情况等。但是需要注意，*RTprop* 与 *BtlBw* 是相互独立的。简而言之，即使传输链路变化，*RTprop* 发生变化，但只要经过相同的瓶颈链路，*BtlBw* 是有可能不变的。

从公式 6.4 可以看出，最近的 *deliveryRate* 的最大值是 *BtlBw*。接下来，我们就使用估算出来的 *BtlBw* 和 *RTprop* 来调节数据发送量。本节介绍了估算公式和大致的流程，接下来会使用伪代码详细介绍实际的 BBR 算法。

6.4 使用伪代码学习 BBR 算法

收到 ACK 时和发送数据时

BBR 算法大致由“收到 ACK 时”和“发送数据时”两部分组成。这里，笔者将使用“BBR：Congestion-Based Congestion Control”[①] 中记载的伪代码，详细介绍各个部分的处理过程。

在第 5 章中，因为 CUBIC 算法是通过三次函数近似 BIC 后，才实现了窗口大小控制，所以笔者先介绍了基础算法 BIC。这样一来，大家理解起来比较容易，所以在了解了 CUBIC 的概要之后便可以确认它的具体流程。至于具体的算法，笔者则放在了后面介绍。但是，相对来说，大家要想使用本章之前介绍过的知识来理解 BBR 的流程尚有些困难，因此下面笔者将首先介绍具体的算法，然后再通过模拟实验介绍其具体流程。

① Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, et al. BBR: Congestion-Based Congestion Control [C]. ACM Queue, vol.14, no.5, p.50, 2016.

收到ACK时

BBR算法会在收到ACK时计算*RTT*和数据发送速率（deliveryRate），随后更新*RTprop*和*BtlBw*。这部分逻辑的伪代码如下所示。

```
function onAck(packet)
  rtt = now - packet.sendtime
  update_min_filter(RTpropFilter, rtt)
  delivered += packet.size
  delivered_time = now
  deliveryRate = (delivered - packet.delivered) / (now - packet.delivered_time)
  if(deliveryRate > BtlBwFilter.currentMax || ! packet.app_limited)
    update_max_filter(BtlBwFilter, deliveryRate)
  if(app_limited_until > 0)
    app_limited_until -= packet.size
```

首先，计算*RTT*的值，使用公式6.3计算*RTprop*的值。接下来，使用delivered变量获取到达数据量，计算出deliveryRate的值。

必须注意，在if语句的执行块中，发送方的数据发送量是由应用程序决定的。换句话说，应用程序的实际发送速率，可能并不足以使数据填满瓶颈带宽区域的带宽。此时，BBR会将此项约束作为“应用程序约束”（application limited）来进行处理，与链路带宽约束分开看待。

发送数据时

接下来，笔者再来介绍发送数据时的算法。BBR会调整发送数据间隔，以便与瓶颈链路带宽适配。这部分逻辑的伪代码如下所示。

```
function send(packet)
  bdp = BtlBwFilter.currentMax * RTpropFilter.currentMin
  if(inflight >= cwnd_gain * bdp)
    // 等待ACK或者超时
    return
  if(now >= nextSendTime)
    packet = nextPacketToSend()
    if(! packet)
      app_limited_until = inflight
      return
    packet.app_limited = (app_limited_until > 0)
```

```
    packet.sendtime = now
    packet.delivered = delivered
    packet.delivered_time = delivered_time
    ship(packet)
    nextSendTime = now + packet.size / (pacing_gain * BtlBwFilter.currentMax)
  timerCallbackAt(send, nextSendTime)
```

首先如前所述，计算 *BtlBw* 和 *RPprop* 的估算值之积，即 *BDP*。

`cwnd_gain` 是用于调整数据发送量的参数。根据网络环境的不同可能出现 ACK 被一并返回的情况，因此如果 `inflight` 被限制到 1 *BDP*，数据发送会被暂时中止。我们使用 `cwnd_gain` 正是为了规避这一情况。根据环境的具体情况，可以将 `cwnd_gain` 设置为 2 或其他较大的值，这样的话即使 ACK 迟到，也能发送适量的数据。

在其他情况下，就只是简单地根据当前数据包的大小，安排下一个数据包的发送时间。然后，比较由 `cwnd_gain` 补正的 `bdp` 值与 `inflight` 值，如果 `inflight` 较大，就停止发送数据包。

看完以上的介绍，大家有没有觉得理解 BBR 的行为变得更容易了呢？ BBR 没有复杂的控制逻辑，只是估算 *RTprop* 和 *BtlBw* 的值，然后根据它们的值调整数据包的发送时间间隔。

6.5 BBR 的流程

模拟实验中的各种流程

前面已经介绍了 BBR 的工作原理和算法，在本节中，笔者将结合模拟实验详细介绍 BBR 的实际行为与性能。

只有 BBR 网络流时的表现

首先，我们来观察在最简单的条件下，也就是只使用一个 BBR 网络流的情况下 BBR 的表现，确认一下其具体行为。基本的模拟条件和实验

11一致，拥塞控制算法设置为TcpBbr。初始阶段只从发送节点❶发送数据。

此外，TcpBbr和TcpCubic一样，没有包含在当前的ns-3官方发布版本中。但是TcpBbr模块已经在Web公开。本书发布和使用的模拟环境中已经安装了TcpBbr。

这里将本次的模拟条件称为“实验14”，通过以下命令来运行该实验。

```
$ ./scenario_6_14.sh
※保存位置 data/chapter6目录下（测试数据：06 xx-sc14-*.data，图表：06 xx-sc14-*.png）
```

模拟结果如图6.10所示。这里，我们主要关注*inflight*的值、拥塞窗口大小和*RTT*的值。

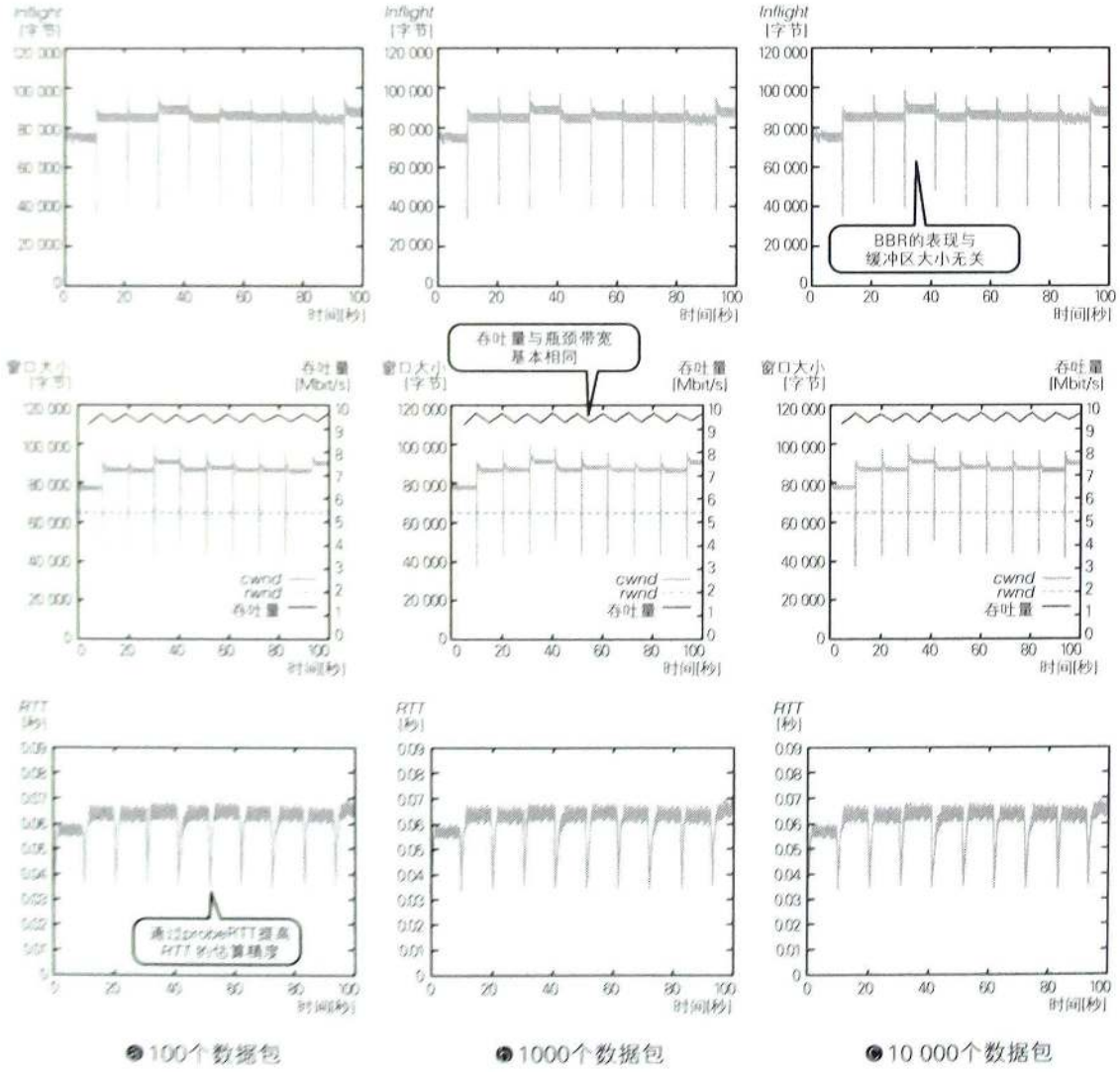


图6.10 模拟结果（实验14）

模拟运行结果 不受缓冲区大小影响，probeRTT

从结果来看，BBR 的行为完全不受缓冲区大小的影响。这主要是由于算法会调整数据发送量，以防止产生缓冲区时延。在所有条件下，吞吐量都达到了瓶颈链路带宽 10 Mbit/s 左右。此外，还可以看到，*inflight* 的值基本上就是瓶颈链路带宽 10 Mbit/s 和约 60 ms 的 *RTT* 值的乘积。最后，*inflight* 和拥塞窗口大小的值显然是相互联动的。

不仅如此，还能看到 BBR 的一些特色，比如 *inflight*、*RTT* 和拥塞窗口大小以大约 10 秒的间隔暂时下降。这种现象称为 probeRTT，它是为了确认是否发生了缓冲区时延而特意定时进行的一个动作。换句话说，即使 *RTprop* 长时间持续不变，也无法保证缓冲区时延就一定不会发生。也有可能是数据一直堆积在缓冲区中，且这一状态持续不变。

那么，如果 *RTTprop* 估算值在一定时间内不变，就将一定时间内（200 ms 左右）的 *cwnd* 的值减小，即减少数据发送量，以此来提高 *RTprop* 的估算精度。BBR 的基本思路便是将 *probeRTT* 的时间设计为总时间的 2% 左右，以此来保证吞吐量增加和 *RTT* 估算精度提升之间的平衡。

至于为何将 *probeRTT* 的时间定为 200 ms，主要是基于以下考虑：即使在不同 *RTT* 网络流混杂的环境下，*probeRTT* 的区间也会有相互重合的时间段。

当多个 BBR 网络流同时存在时

接下来，我们通过模拟来确认一下多个 BBR 网络流共享瓶颈链路时的情况。当多个网络流同时进入时，与单独的网络流不同，网络流还会受其他网络流的影响。这里的模拟条件与实验 14 基本相同，在本次模拟实验中，所有的发送节点会同时发送 BBR 网络流。

这里将本次的模拟条件称为“实验 15”，输入以下命令来运行该实验。

```
$ ./scenario_6_15.sh
※保存位置：data/chapter6目录下（测试数据：06_xx-sc15-*.data，图表：06_xx-sc15-*.png）
```

模拟结果如图6.11所示。这里显示的是发送节点❶发送的网络流的行为表现，其他的网络流与节点❶一样，没有任何区别。

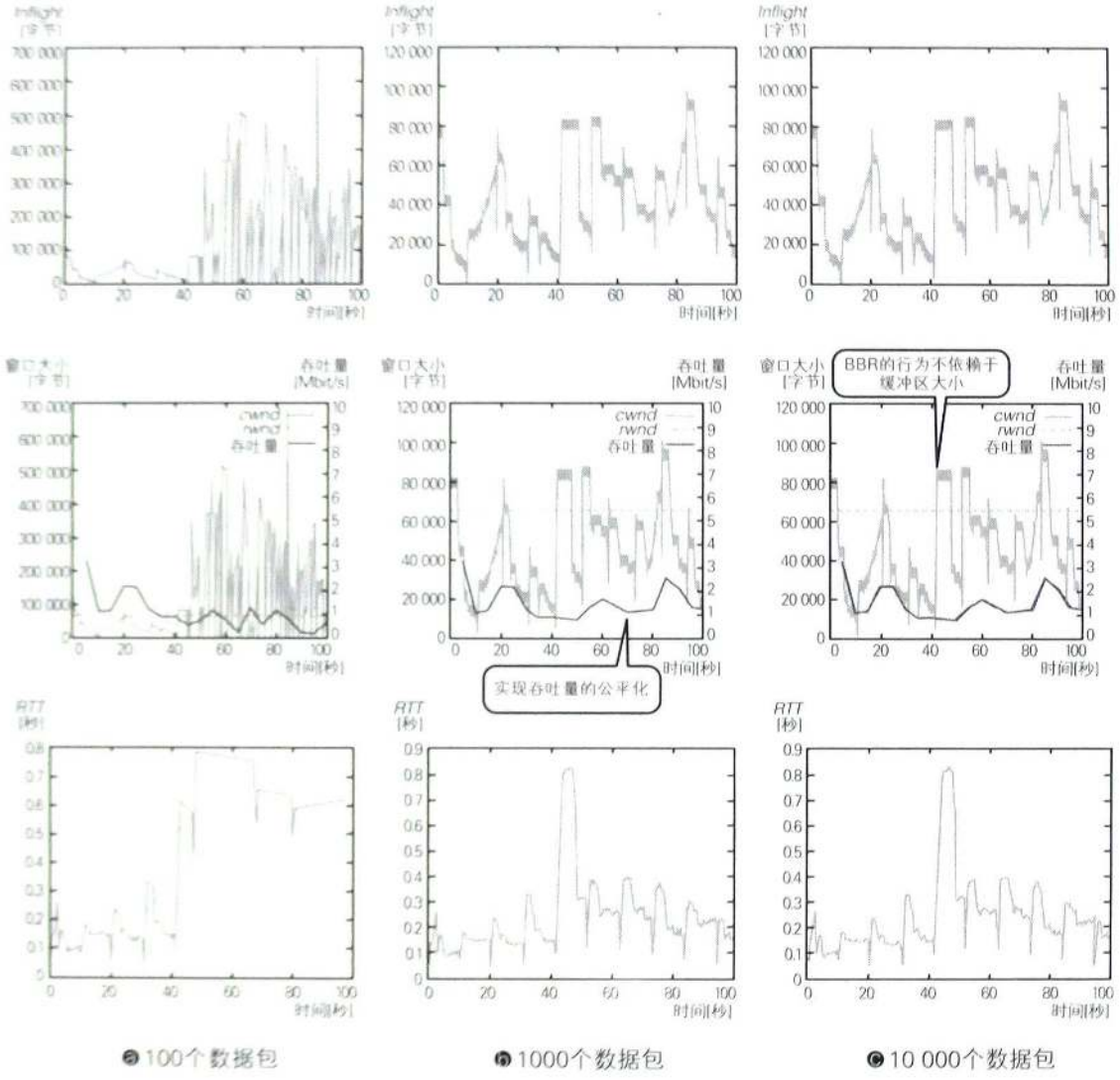


图6.11 模拟结果（实验15）

模拟运行结果 几乎完全公平地共享吞吐量

首先，缓冲区大小分别在1000和10 000个数据包的情况下，BBR的行为是一样的。这主要是因为缓冲区大小绰绰有余，大部分缓冲区没有被使用。但是，当缓冲区大小是100个数据包时，如果所有网络流的数据包都爆发性地到达，有时就会出现缓冲区溢出的情况，这一点和有1000或10 000个数据包时的情况有所不同。

此外，网络设备端并没有特意进行公平性方面的控制，但 BBR 网络流之间会公平合理地共享网络带宽，各个 BBR 网络流可以达到几乎相同的吞吐量，这可以说是 BBR 的一大特点。以 CUBIC 为首的其他拥塞控制算法，只要其拥塞窗口大小先增大，吞吐量就也会增多。而之后新加入的网络流便会被之前的网络流影响，拥塞窗口大小无法增大。

对此，BBR 设计了暂时减少数据发送量的 probeRTT 阶段，此阶段会将拥塞窗口大小暂时减小，通过重新调整来减少上述先到者的优势。在重新调整之后，各个网络流的 *RTprop* 和 *BtlBw* 估算值基本上相等，因此吞吐量更容易实现公平化。

与 CUBIC 的共存

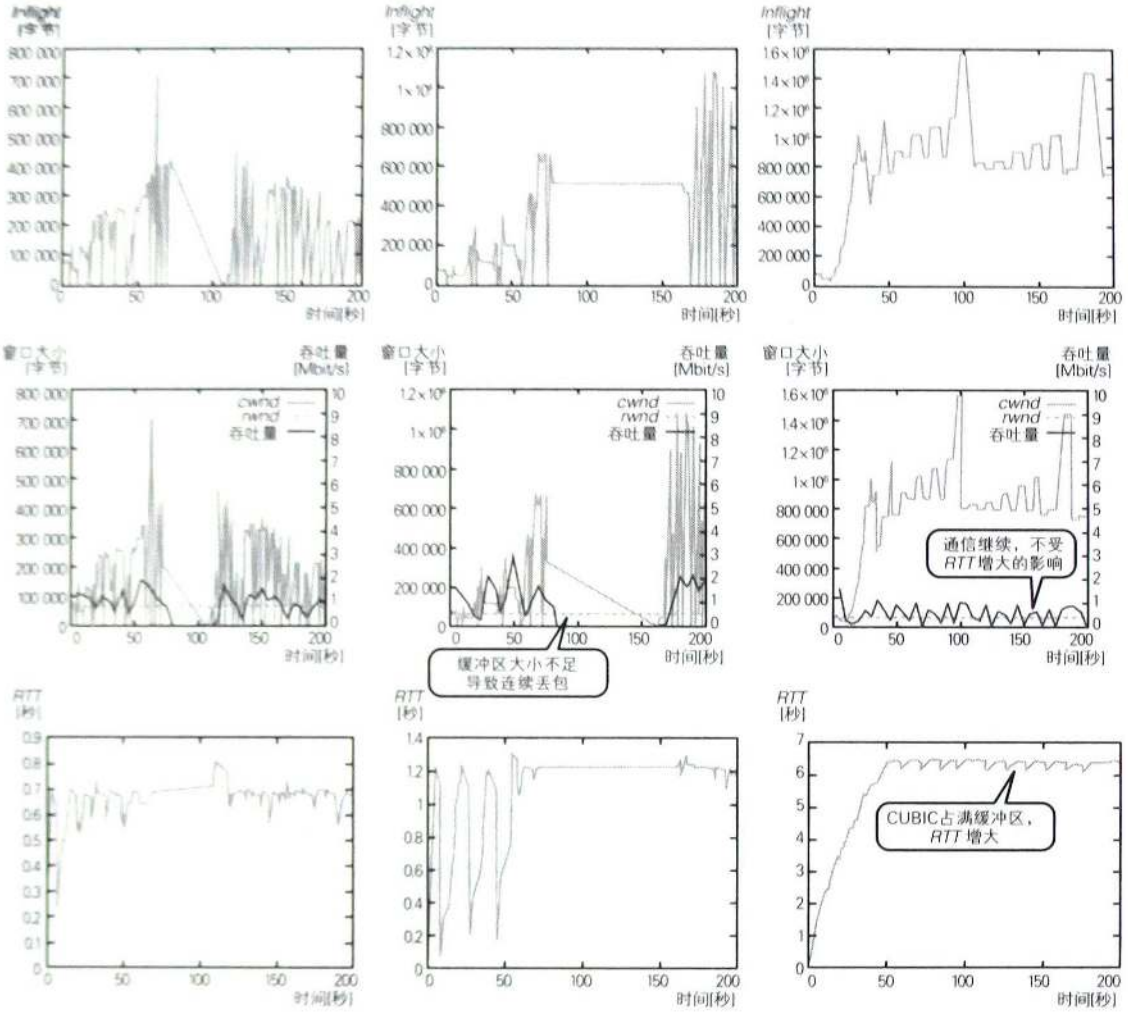
6.2 节曾介绍了 Vegas 存在的一个问题，即当 Vegas 与基于丢包的拥塞控制算法在一起时，其吞吐量会掉到接近 0 的水平。那么如果换成 BBR，又是什么情况呢？让我们通过模拟来确认一下。这里设置一个与实验 15 类似的模拟条件，只将发送节点 ❶ 的网络流设置为 BBR，其他的网络流设置为 CUBIC，然后来观察 BBR 网络流的表现。

这里将本次的模拟条件称为“实验 16”。打开 ns-3 的根目录，输入以下命令来运行该实验。

```
$ ./scenario_6_16.sh
```

※保存位置：data/chapter6目录下（测试数据：06_xx-sc16-*.data，图表：06_xx-sc16-*.png）

模拟结果如图 6.12 所示。



❷100个数据包　　❸1000个数据包　　❹10 000个数据包

图 6.12 模拟结果（实验 16）

模拟运行结果　与基于丢包的拥塞控制共存时的情况

在本次的模拟条件下，CUBIC 网络流占主要地位，这些网络流会将缓冲区占满，因此缓冲区越大，时延也就越大。其中，Vegas 会被淘汰，变得几乎无法通信；与之相对，BBR 则可以继续通信，还能达到一个近乎公平的吞吐量（网络流之间平分瓶颈链路带宽时的值）。但是，在缓冲区较小时，中间会有一段时间无法通信。究其原因，主要是受到了数据包不断被废弃的影响。

此外，在这次的模拟条件下，瓶颈链路带宽就只有 10 Mbit/s，缓冲

区时延引起的 *RTT* 增大所带来的最大吞吐量较小的问题并没有暴露出来。如果瓶颈链路带宽或者缓冲区大小中的任一项较大，且 CUBIC 等基于丢包的拥塞控制算法占主导地位，那么缓冲区时延就会增大。此时，即使使用 BBR 也无法规避吞吐量下降的问题。换句话说，为了防止缓冲区膨胀问题出现，就需要提高使用 BBR 类拥塞控制算法的网络流的比例。

长肥管道下的表现

现在，我们已经确认了低速链路中 BBR 的表现，接下来将通过模拟实验看一下 BBR 在上一章提到过的宽带、高时延环境（长肥管道）下的适应性。

这里将本次的模拟条件称为“实验 17”，模拟条件与上一章的实验 1 基本一致，拥塞控制算法设置为 BBR。打开 ns-3 的根目录，输入以下命令来运行实验 17。

```
$ ./scenario_6_17.sh
（※保存位置：data/chapter6目录下（测试数据：06_xx-sc17-*.data，图表：06_xx-sc17-*.png））
```

模拟结果如图 6.13 所示。这里显示的是拥塞窗口大小、吞吐量、*inflight* 和获取的 *RTT* 的数据。

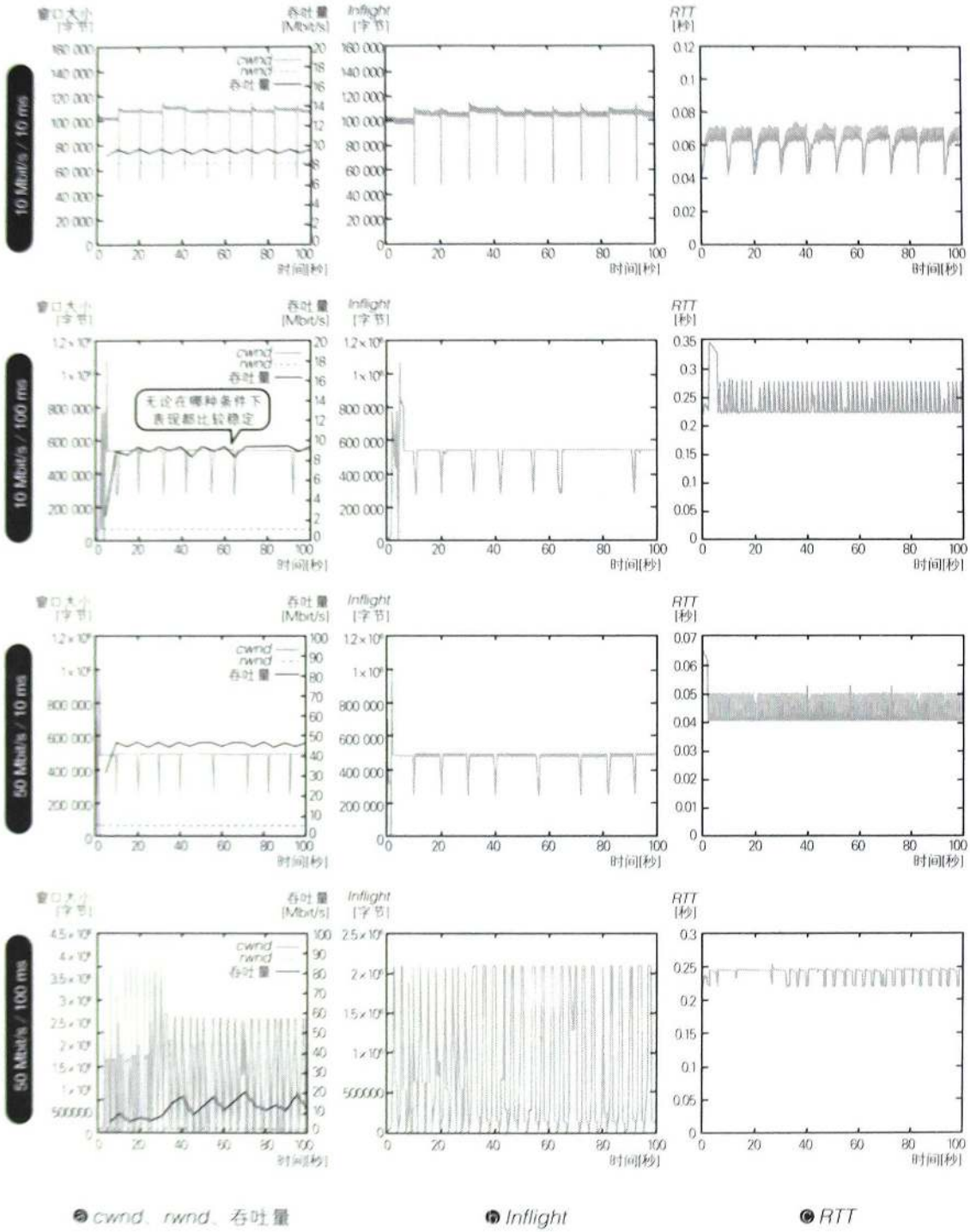


图 6.13 模拟结果（实验 17）

模拟运行结果（大致）维持了高且稳定的吞吐量

从结果可以看出来，在使用BBR时，无论在哪一种环境下，与上一章的结果相比，都能维持一个高且稳定的吞吐量。这主要是因为BBR与基于丢包的拥塞控制算法不同，它不会持续增大拥塞窗口大小，因此也就

不会发生丢包。

但是，在有些情况下，BBR 并不能完美地与 CUBIC 和 NewReno 共存，原因就是 BBR 并非在所有情况下都能有理想的表现。BBR 目前还是比较新的拥塞控制算法，因此今后想必也会有针对更多环境的验证和改良。此外，今后网络环境很可能进一步变化，与之相对的新拥塞控制算法也极有可能出现。持续跟上时代的脚步并学习新技术无疑十分重要。

6.6 小结

本章结合模拟实验介绍了近些年来逐渐暴露出来的缓冲区膨胀现象、过去基于丢包的拥塞控制算法受此影响出现的问题，以及新亮相的基于延迟的拥塞控制算法 BBR。这里简单地回顾和总结一下本章内容。

近些年来存储成本逐渐降低，路由器和交换机等网络设备上搭载的缓冲区存储容量不断增大。一方面，随着网络设备中缓冲区的增大，丢包就更不容易出现，换句话说，这带来了“爆发耐性增加”的好处；另一方面，随着缓冲区的增大，数据包堆积在缓冲区中，也使得队列时延增大。此问题最终导致的时延增大和吞吐量下降的现象便是缓冲区膨胀。

人们过去一直使用的 NewReno 和 CUBIC 等基于丢包的拥塞控制算法，由于以丢包作为拥塞的指标，所以只要不出现丢包（ = 缓冲区溢出），就会一直增大拥塞窗口大小，这很容易导致缓冲区时延增大。

与之相对，基于延迟的拥塞控制算法使用 *RTT* 作为判断网络拥塞状态的指标。换句话说，一旦 *RTT* 增大，就认为原因是链路上的队列时延增大，于是当 *RTT* 较小时就增大拥塞窗口大小，而当 *RTT* 较大时就减小拥塞窗口大小。这其中最为典型的算法就是 Vegas 拥塞控制算法。

然而，以 Vegas 为首的基于延迟的拥塞控制算法积极性不强，当其与基于丢包的拥塞控制算法共存时很容易被淘汰。为了解决上面的问题，谷歌于 2016 年 9 月又发布了名为 BBR 的基于延迟的拥塞控制算法。目前

BBR的使用非常广泛，Linux中已默认支持它。

BBR认为过去的基于丢包的拥塞控制算法以丢包为契机检测拥塞，这种做法过于迟钝，因此它致力于维持“数据包即将堆积在缓冲区中但还没开始堆积”的临界状态，此时既能充分利用网络带宽，又没有缓冲区时延。为了达到这种理想状态，BBR监测数据发送量和*RTT*的值，把控两者之间的关系，同时调节数据发送速度，以在网络最大可处理的范围内提高吞吐量。

不仅如此，从本章的模拟结果可以看出，BBR可以作为大部分情况下的拥塞控制算法。不过，BBR目前仍然是比较新的拥塞控制算法，因此可以想象，它今后一定会面临很多挑战，也会迎来很多改进。此外，如目前看到的一样，今后随着技术的进步，网络环境一定会继续变化，想必也会有新的问题浮出水面。换句话说，倘若今后网络环境继续发生变化，一定会有与新变化对应的新技术出现。

因此，下一章将介绍近些年来出现的，以及将来可能会出现的以TCP为中心的技术及社会环境，探讨随之而来的各类问题。此外，下一章还会介绍TCP相关的研究动向。

参考资料

- Steven Low, Larry Peterson, Limin Wang. Understanding TCP Vegas: Theory and Practice [R]. Prinston University Technical Reports, TR-616-00, 2000.
- Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, et al. BBR: Congestion-Based Congestion Control [C]. ACM Queue, vol.14, no.5, p.50, 2016.
- Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, et al. BBR Congestion Control [R]. Google Networking Research Summit, 2017.