在 Linux 中，毫无疑问是现今主流的拥塞控制算法之一。

我们从本章内容可以看出，即使是性能卓越的拥塞控制算法，随着技术的进步和网络环境的变化，有时也会不断暴露出新的问题。此外，近些年来，随着存储成本的降低和其他一些因素的变化，路由器等网络设备中安装的缓冲区存储容量也呈现出增大的趋势。在这种环境下，如果使用基于丢包的拥塞控制算法，就会出现一种情况，即只要不发生丢包，拥塞窗口大小就会无限增大，因此网络链路中的缓冲区会被逐步占满，然后队列时延就会增大，最终导致 *RTT* 增大或吞吐量减小。下一章将详细介绍这一问题，以及专门为了解决这一问题而开发的新算法。

参考资料

- 甲藤二郎 . 広帯域高遅延環境における TCP の課題と解決策 [J]. 電子情報通信学会 . 知識の森，3群4編 2-1，2014.
- Sangtae Ha，Injong Rhee，Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant [C] ACM SIGOPS operating systems review，vol.42，no.5，pp.64-74，2008.
- Tom Kelly，Scalable TCP: Improving Performance in Highspeed Wide Area Networks [C]. ACM SIGCOMM Computer Communication Review，Vol 33，No 2，pp 83-91，2003.
- 《面向大型拥塞窗口的高速 TCP》(RFC 3649) .
- 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.

第 6 章

BBR 算法

检测吞吐量与RTT的值，调节数据发送量

近些年来，在存储成本不断下降和通信速度不断提升的背景下，路由器等网络设备中的缓冲区存储容量不断增大。随之而来的，便是包含 CUBIC 算法在内的那些过去的拥塞控制算法，在缓冲区时延增大的情况下出现了吞吐量下降的问题。

针对此问题，谷歌在 2016 年 9 月发布了新的拥塞控制算法 BBR。它属于基于延迟的拥塞控制算法，以 *RTT* 作为指标增减拥塞窗口大小。如今，BBR 的应用范围很广，不仅默认搭载在 Linux 系统中，还在 Google Cloud Platform（GCP）中被广泛使用，可以说是目前主流的拥塞控制算法之一。

本章将结合模拟实验，详细介绍缓冲区增大给 CUBIC 带来的影响，以及 BBR 算法和其具体性能。

6.1 缓冲区增大与缓冲区时延增大

存储成本下降的影响

由于网络设备上安装的缓冲区存储容量增大，所以缓冲区时延增大所导致的吞吐量下降问题逐渐暴露出来。本节将介绍缓冲区时延增大给 TCP 通信带来的影响。

网络设备的缓冲区增大

近些年来，路由器、交换机等网络设备上安装的缓冲区存储容量不断增大。究其原因，主要是存储成本不断下降。此外，大部分人认为存储容量越大越好，而且这一点不仅限于网络设备。这种思潮也是引发这一现象的原因之一。

网络设备的缓冲区增大，好处就是更不容易出现网络丢包。换句话说，即使网络设备瞬间收到大量数据包，也可以将这些数据包存储在缓冲区中，并按顺序发送出去。这里，我们将这种爆发性网络流量下更不容易出现丢包的特性称为爆发耐性。

如果缓冲区过小，一旦爆发性流量到达设备，就很容易超出缓冲区大小的上限，使缓冲区溢出，并发生丢包。此时，只要到达的网络流（TCP 网络流）使用的是之前介绍的基于丢包的拥塞控制算法，吞吐量就会在每次发生丢包时减小。即使拥塞窗口大小很小，如果网络流碰巧与其他的网络流同时到达网络设备，那么对于网络设备来说，这种情况同样可以视作爆发性的网络流量，也会导致数据包被废弃。因此，即使网络流的传输速率很低，也很容易因为拥塞控制算法的过激反应而导致吞吐量出现不必要的下降。

如果增大缓冲区，就不容易发生此类问题，网络通信也会更加稳定。图 6.1 便是缓冲区大小与网络丢包之间的关系示意图。

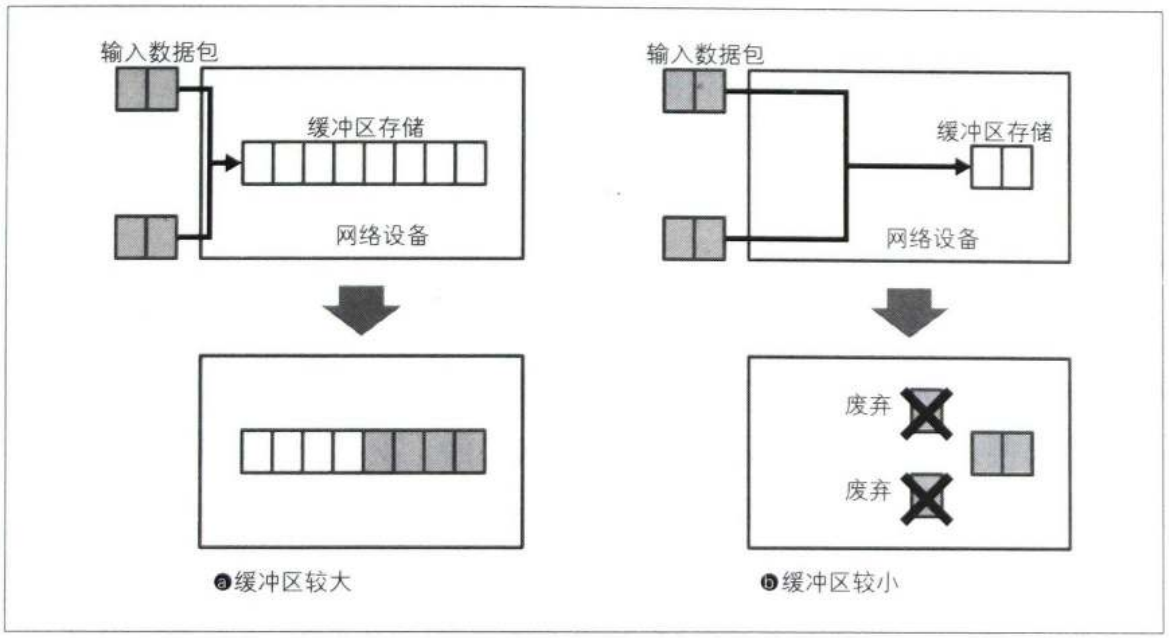


图 6.1 缓冲区大小与丢包（数据包废弃）

缓冲区膨胀 缓冲区增大所带来的危害——时延增大

虽然增大缓冲区可以减少丢包，但缓冲区容量并非越大越好。增大缓冲区也会带来其他的危害，那就是缓冲区时延增大。由缓冲区时延增大带来的通信数据包的端到端时延增大的现象称为缓冲区膨胀（bufferbloat，缓冲区时延增大），此问题从 2009 年开始逐渐被广泛关注。

根据上一章介绍的知识，通信数据包的端到端时延如图 6.2 所示，包括链路上的网络设备的**处理时延**、在各个节点的缓冲区存储上的转发等待时间即**队列时延**，以及节点之间链路的信号传输所必需的物理上的**传播时延**。

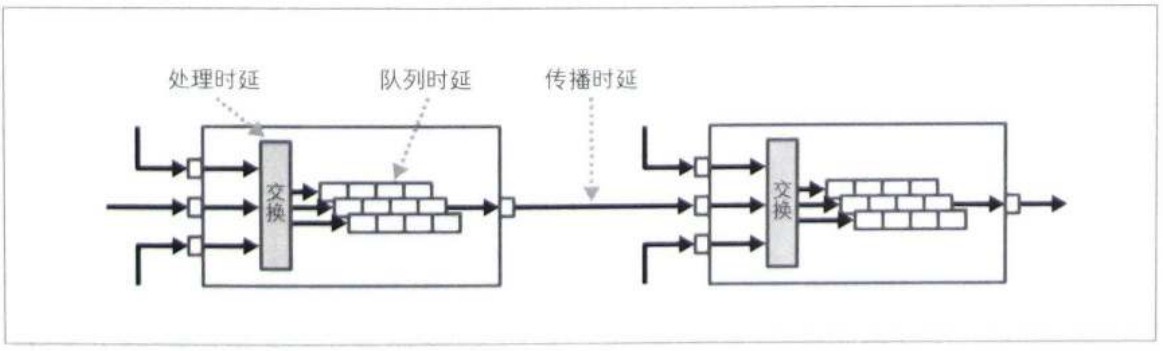


图 6.2 端到端时延的构成

当缓冲区容量增大，等待的数据包的数量就会增加，也就是说，在存储上的转发等待时间——队列时延就会增大。随之而来的便是 *RTT* 的增大，最终的结果是 TCP 网络流的吞吐量下降。

——AQM 与 RED 算法

为了控制缓冲区膨胀现象，许多方法被提出并实现。AQM（Active Queue Management，主动队列管理）是其中具有代表性的技术。AQM 算法的思路便是在缓冲区容量即队列被填满之前，主动地开始丢包处理，以防止出现缓冲区溢出的现象。

RED（Random Early Detection，随机早期检测）是最著名的 AQM 算法。RED 算法时刻监测队列长度，当发现队列长度超过一定的阈值之后，就按照提前设置好的概率开始丢弃输入数据包。队列长度越长，丢包率就会设置得越大，以防止出现缓冲区溢出。在使用 RED 时，发送包量较大的网络流将以更高的概率丢弃数据包。与尾丢包（tail drop，在缓冲区溢出时丢弃数据包的策略）相比，这种方法更为公平。此外，还有加权 RED 算法等多种 AQM 算法。

AQM 是进行数据包中转的网络设备端的算法技术，不过，我们也能很容易地想象到，发送数据包的流量生成方，即运行 TCP 拥塞控制算法的一方，肯定也有相应的算法对策。在开始介绍控制缓冲区膨胀的拥塞控制算法之前，笔者将简单介绍一下过去的基于丢包的拥塞控制与缓冲区膨胀之间的关系。

基于丢包的拥塞控制与缓冲区膨胀的关系

首先，我们来回顾一下基础知识。TCP 要发送新数据，就必须先从接收方接收到 ACK。这是 TCP 的基本行为，与拥塞控制算法完全无关，同时也是 TCP 与 UDP 的不同。如果是 UDP，那么发送节点只要持续不断地发送数据即可，完全不会受链路上缓冲区时延等的影响。但是，UDP 无法保证数据一定能到达目的地。

TCP 一边通过接收方发送回来的 ACK 确认数据已经成功到达目的地，

一边发送数据。因此，如图 6.3 所示，随着缓冲区时延的增大，数据到达所需要的端到端时延会增大，而 ACK 的到达也会随之推迟，最终的结果就是数据的发送间隔变长，吞吐量下降。

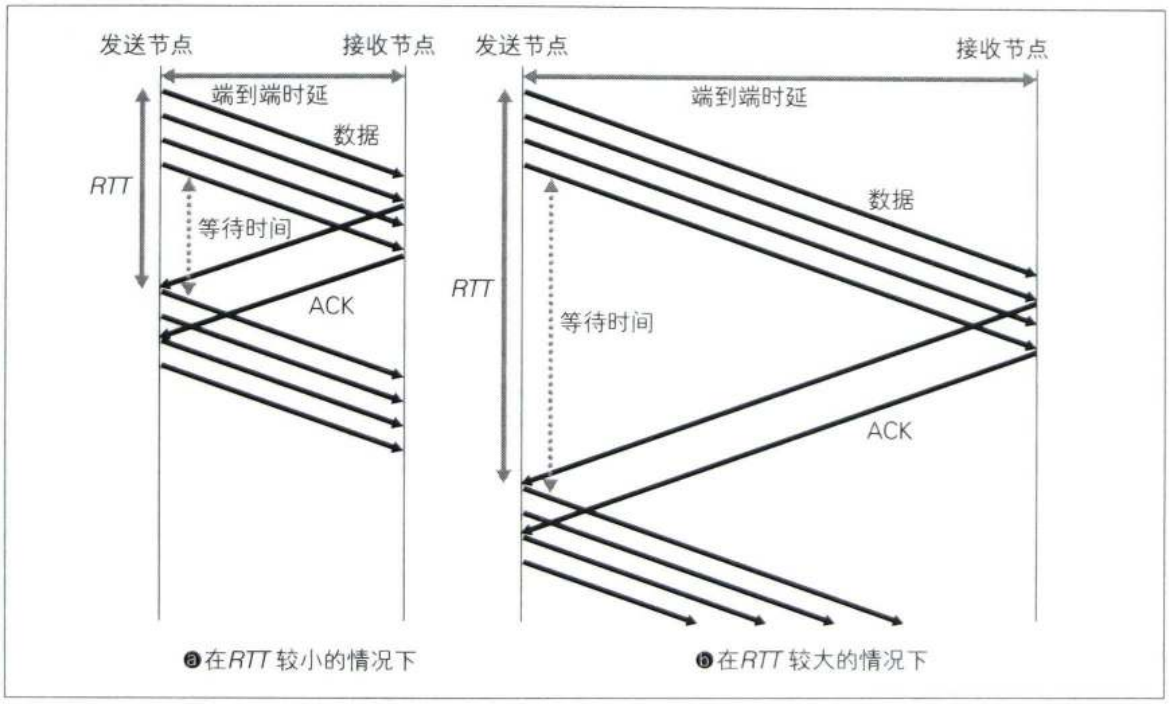


图 6.3 RTT 与吞吐量的关系

根据缓冲区容量与 *RTO*（超时重传时间）的大小关系，甚至可能会出现数据包在链路上的缓冲区中停留的时间超过 *RTO*，导致数据重传的情况。而且，基于丢包的拥塞控制算法正是以丢包作为判断拥塞的指标。因此，只要不发生丢包，就会一直增大拥塞窗口大小。

近些年来，随着数据传输可靠性（错误少）的提升，缓冲区溢出成了数据丢包的主要原因。也就是说，在使用基于丢包的拥塞控制算法时，只要不发生丢包（不出现缓冲区溢出），数据发送量就会一直增加。

一方面，如果网络链路上的缓冲区较小，就会因缓冲区溢出而频繁发生丢包；另一方面，随着缓冲区增大，缓冲区不再容易溢出，但是发出的数据包会将网络中的各台设备的缓冲区填满，带来的影响是 *RTT* 增大。

总的来说，TCP 网络流本身是引起缓冲区膨胀的原因，同时缓冲区膨胀又会反过来影响 TCP 网络流自身。考虑到因为超时而进行重传等情况，可以断言，基于丢包的拥塞控制算法更容易使时延增大。

缓冲区增大给CUBIC带来的影响 通过模拟来确认

接下来，我们通过模拟来看一下缓冲区时延增大给TCP吞吐量带来的影响。下面将使用基于丢包的拥塞控制算法中的代表性算法CUBIC。我们在修改网络上的缓冲区大小的同时，收集相应情况下的拥塞窗口大小、*RTT*和吞吐量的数据。模拟条件如图6.4所示。这里，我们将本次的模拟条件称为“实验11”。

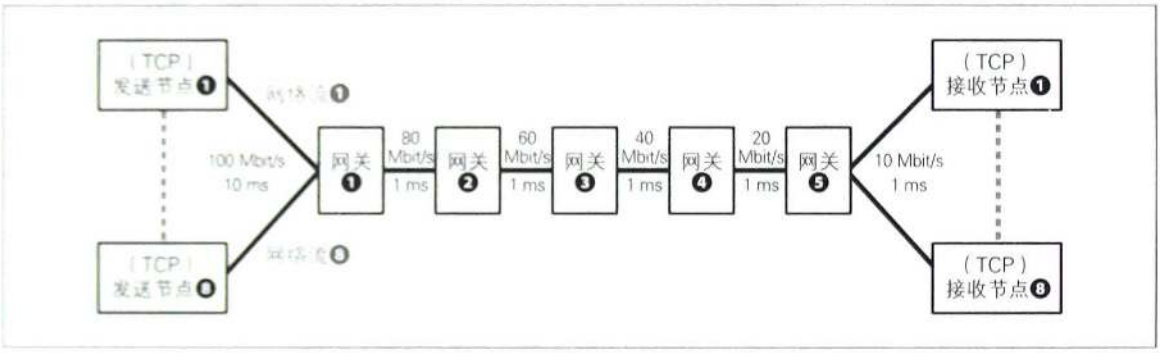


图6.4 模拟条件（实验11）

本次实验为了模拟网络上的缓冲区时延，组建了一个经过多个网关节点（gateway node）的网络拓扑[1]结构。这里，我们是为了方便才使用“网关节点”这个说法的，大家可以将其想象为网络上的路由器等设备，应该会更方便理解。

随着发送节点向接收节点不断发送数据包，模拟环境上的链路速度会降低，数据包会滞留在各个网关节点的发送队列中，这便导致了缓冲区时延。这里模拟的便是“互联网上多个网络流汇合之时，每个网络流可占用的带宽会变小”的现象。为了减少模拟的时间，这里将发送节点个数控制在8个，这样就可以不用准备更多的网络流。

实验11具体要收集的数据与上一章一样，即拥塞窗口大小（*cwnd*）、吞吐量（基于到达接收节点的数据量，每5秒计算一次）、拥塞状态和*RTT*，共4项。将接收窗口大小（*rwnd*）设置为65 535字节，打开窗口扩大（window scaling）选项，然后把代表最大数据包长度的*MTU*设置为

1 其英文为network topology，指的是网络的逻辑和物理上的形态。

1500 字节，把测试时间设置为 100 秒。为了确认缓冲区增大的影响，这里分别将各个网关节点的最大队列长度设置为 100、1000、10 000 个数据包，并分别基于这 3 种环境进行模拟。

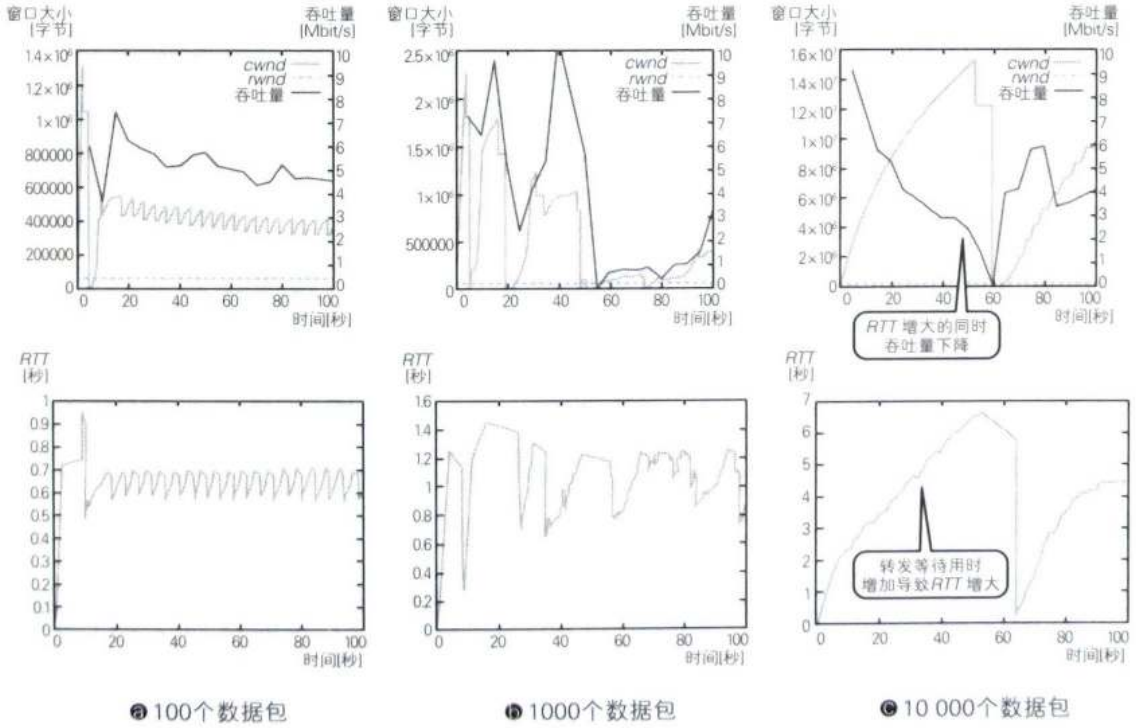
打开 ns-3 的根目录，通过以下命令来运行实验 11。

```
$ ./scenario_6_11.sh
```

※保存位置：data/chapter6目录下（测试数据：06_xx-scl1-*.data；图表：06_xx-scl1-*.png）

模拟运行结果 随着缓冲区增大，RTT 显著增大

模拟运行结果如图 6.5 所示。这里，我们首先将目光集中到发送节点❶发送的 TCP 网络流的拥塞窗口大小和吞吐量上。从图中可以一目了然地看到，随着网络上的缓冲区增大，*RTT* 显著增大。特别是在队列长度为 10 000 的情况下，*RTT* 甚至到了 7 秒左右，这对吞吐量的影响无疑非常大。此外，本次模拟中各个网络节点的队列设置为 DropTail 方式，因此需要注意，由于没有特意对网络流之间的公平性进行控制，所以最终可能链路带宽并不会被公平地使用。



ⓐ100个数据包　ⓑ1000个数据包　ⓒ10 000个数据包

图 6.5 模拟结果（实验 11）

综上所述，基于丢包的拥塞控制算法只要仍有缓冲区，就会不断地增加数据发送量。正因为有这个特点，这类算法在缓冲区比较大的情况下才会出现缓冲区时延增大的问题。

6.2 基于延迟的拥塞控制

以RTT为指标的算法的基本情况和Vegas示例

在TCP的拥塞控制算法中，**基于延迟的拥塞控制算法**使用*RTT*作为指标。接下来，笔者就以具有代表性的基于延迟的拥塞控制算法Vegas为例，介绍一下其基本的行为。

3种拥塞控制算法和如何结合环境选择算法

从前面介绍的内容来看，TCP的拥塞控制算法大致上分为3种（图6.6）。具体来说，就是以丢包作为拥塞指标的**基于丢包的拥塞控制**、以*RTT*为指标的**基于延迟的拥塞控制**，以及两者结合的**混合型拥塞控制**。

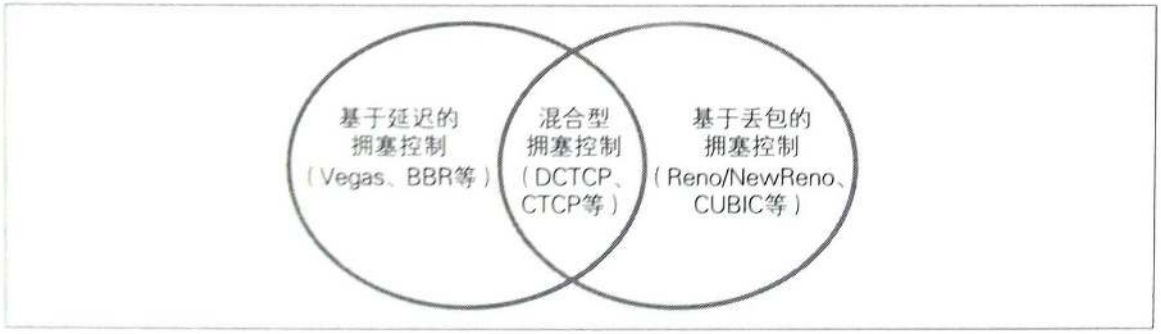


图6.6 拥塞控制算法的种类

必须注意，这些算法各有各的特征，所适应的环境也各有不同，我们无法简单地说哪种算法更加优秀。

例如，进行复杂拥塞控制的算法通常能以更小的粒度增减拥塞窗口大小，但是此类算法存在许多问题，例如理论上难以解析，无法预测其下一

步的行为，以及在性能较低的设备上甚至无法运行等。此外，如上一章所述，NewReno 不适合宽带、高时延环境，这也是一个非常有代表性的例子。上一章介绍了与现有 TCP 的亲和性较差的 HighSpeed 与 Scalable 算法，假如能够为它们准备一个不存在其他拥塞控制算法的局域网环境，那么这两种算法便可以活用自己在扩展性方面的优势，成为拥塞控制算法中有力的竞争者。

综上所述，选择算法需要结合实际的使用环境，因此我们必须详细地了解各个算法的特点。上一章介绍了上述 3 种拥塞控制算法中基于丢包的拥塞控制算法。那么，本章就来介绍基于延迟的拥塞控制算法。

基于延迟的拥塞控制的基本设计思路

RTT 的增大与队列时延的增大

前文已经介绍过了，TCP 每次收到 ACK 时就会计算 *RTT* 的值。基于延迟的拥塞控制会将这个 *RTT* 值作为判断网络拥塞状态的指标，当发现 *RTT* 比较大时，就认为拥塞状态已经恶化，然后调整拥塞窗口的大小。

RTT 是通过合并计算往返的端到端时延而得到的。端到端时延的构成要素如第 207 页的图 6.2 所示。在这些之前已经介绍过的端到端时延的构成要素中，节点链路间传输信号所必须的物理上的**传播时延**和传输链路上网络设备中的**处理时延**基本上保持不变[①]。与之相对，在内存存储上的转发等待时间，也就是**队列时延**，会受缓冲区上能存储多少数据的影响而大幅变化。由于从输出链路流出的数据，其转发速度保持稳定，所以在此速度下，单位时间内输入的数据量如果变多，那么无法及时输出的数据就会缓存在内存中，并不断累积，这就会导致转发新到达的数据包要花费更多的时间。

也就是说，基于延迟的拥塞控制正是利用了以上性质，认为链路上 *RTT* 增大的原因正是**队列时延增大**。因此，此类拥塞控制算法的基本逻辑就是在 *RTT* 较小时增大拥塞窗口大小，而在 *RTT* 较大时减小拥塞窗口大小。

迄今为止，已经有许多种基于延迟的拥塞控制算法被提出来和使用。其中具有代表性的算法包括 Vegas、面向宽带环境开发的 FAST TCP 等。

① 严格来说，根据设备和条件不同，会有少许变化。

此外，本章将详细介绍的BBR也属于这种类型。接下来，本书将以具有代表性的算法Vegas为例，结合模拟实验和收集的数据详细地观察一下其拥塞窗口大小的变化情况。

Vegas的拥塞窗口大小的变化情况

下面将介绍Vegas拥塞窗口大小控制算法[①]的要点。设时刻t的拥塞窗口大小为$w(t)$，那么时刻$t+1$的拥塞窗口大小可以用公式6.1来表示。

$$w(t+1)=\begin{cases} w(t)+\dfrac{1}{D(t)} & \left(\dfrac{w(t)}{d}-\dfrac{w(t)}{D(t)}<\alpha\right) \\ w(t)-\dfrac{1}{D(t)} & \left(\dfrac{w(t)}{d}-\dfrac{w(t)}{D(t)}>\alpha\right) \\ w(t) & (else) \end{cases} \qquad \text{（公式6.1）}$$

在上面的公式中，d是往返的传播时延，$D(t)$代表实际测试到的RTT。也就是说，$w(t)/d$是期望吞吐量，$w(t)/D(t)$是时刻t的实际吞吐量，拿它们的差值与阈值α进行比较，然后根据结果来增减$w(t)$的值。此外，阈值α是提前设置的参数。如果网络上的缓冲区时延非常小，期望吞吐量与实际吞吐量的差值就会比α小，此时就如公式6.1所示，每个RTT让拥塞窗口大小增大1。而当拥塞比较严重，缓冲区时延增大时，期望吞吐量与实际吞吐量的差值会大于α。此时，每个RTT让拥塞窗口大小减小1，以此控制数据包的发送。接下来，我们通过模拟来确认一下Vegas的拥塞窗口大小的变化情况。

这里使用与之前的实验11一样的模拟条件，用Vegas代替CUBIC进行实验。我们将本次的模拟条件称为“实验12”。打开ns-3的根目录，输入以下命令来运行实验12。

```
$ ./scenario_6_12.sh
```

（保存位置：data/chapter6目录下［测试数据：06_xx-sc12-*.data，图表：06_xx-sc12-*.png］）

① Steven Low, Larry Peterson, Limin Wang. Understanding TCP Vegas: Theory and Practice [R]. Printson University Technical Reports, TR-616-00, 2000.

模拟运行结果 RTT 和吞吐量都与缓冲区大小无关

模拟运行结果如图 6.7 所示。从结果我们可以一眼看出，在使用 Vegas 的情况下，*RTT*、吞吐量完全与缓冲区大小无关，并保持一定值不变。*RTT* 的值也只有 50 ms 左右，比较小。在这次的模拟条件下，传播时延被设置为往返 30 ms，因此缓冲区时延总共为 20 ms 左右。

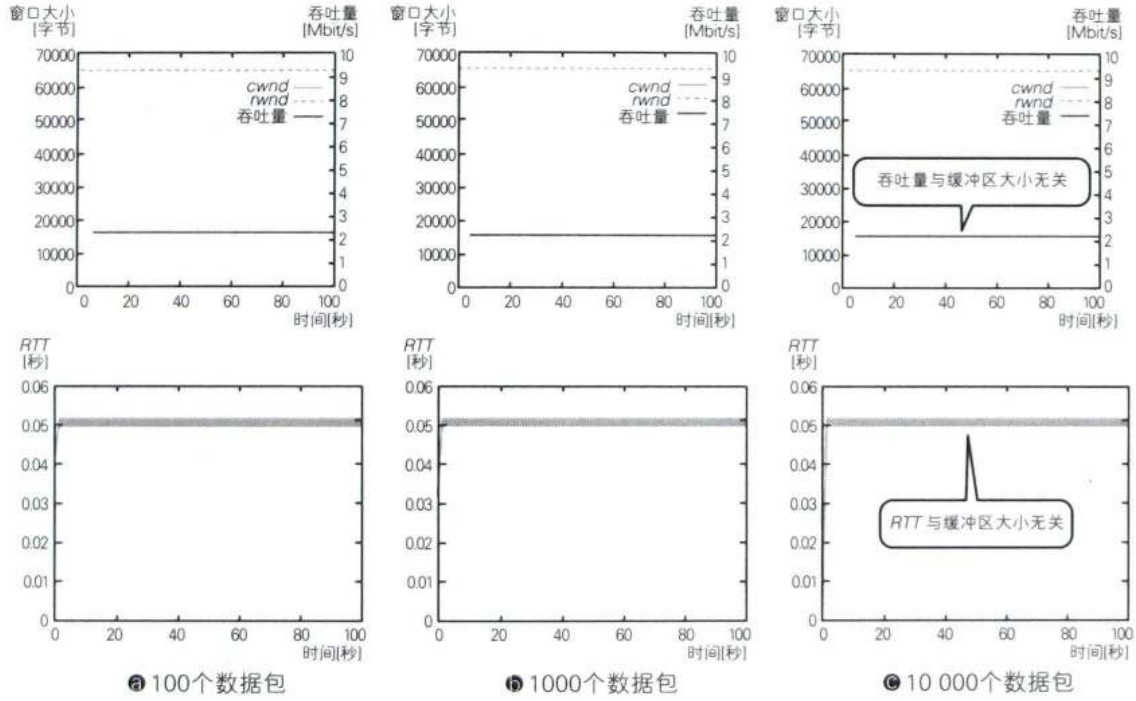


图 6.7 模拟结果（实验 12）

综上所述，在缓冲区时延稍微增大时，通过控制拥塞窗口大小的增大防止拥塞进一步恶化，便可以保持吞吐量稳定。

过去的基于延迟控制的问题 积极性过差，容易被淘汰

按“惯例”，接下来我们看一下 Vegas 存在的问题。Vegas 在理想的环境下确实可以不发生数据丢包，而且还可以保持稳定、低时延，实现高吞吐量。这一点可以从刚才的模拟结果中看到。

然而很遗憾，互联网这种牵涉非特定的多台主体设备的环境，其实很多是非理想的环境。举个具体的例子，Vegas 的积极性非常低，当 *RTT* 增大之后，它很快就会减小拥塞窗口大小，因此就非常容易被基于丢包的拥塞

控制算法淘汰，简而言之就是很难与这些算法共存。况且，过去一直被广泛使用的算法是基于丢包的拥塞控制算法 NewReno，而之后替代它的算法是 CUBIC。这样看下来，想要在互联网中实际使用 Vegas 确实很难。

接下来，我们通过模拟来确认一下上述 Vegas 的问题。这里使用与之前的实验 11 几乎一样的模拟条件，同时只将发送节点❶的网络流设置为 Vegas，其他的网络流设置为 CUBIC，然后观察 Vegas 的行为。这里将本次的模拟条件称为“实验 13”。打开 ns-3 的根目录，输入以下命令来运行实验 13。

```
$ ./scenario_6_13.sh
```

※保存位置：data/chapter6目录下（测试数据：06_xx-sc13-*.data，图表：06_xx-sc13-*.png）

模拟运行结果 与基于丢包的拥塞控制算法很难共存

模拟运行结果如图 6.8 所示。在这种条件下，占据主要地位的 CUBIC 网络流会将缓冲区用光，所以 *RTT* 会随着缓冲区大小一起增大，最终变成与实验 11 一样的结果。这是 CUBIC 本身的特点，无法解决，然而此时由于其他网络流的数据堆积在队列上，Vegas 会因此受到 *RTT* 变化的影响而作出反应，不断减小拥塞窗口大小。

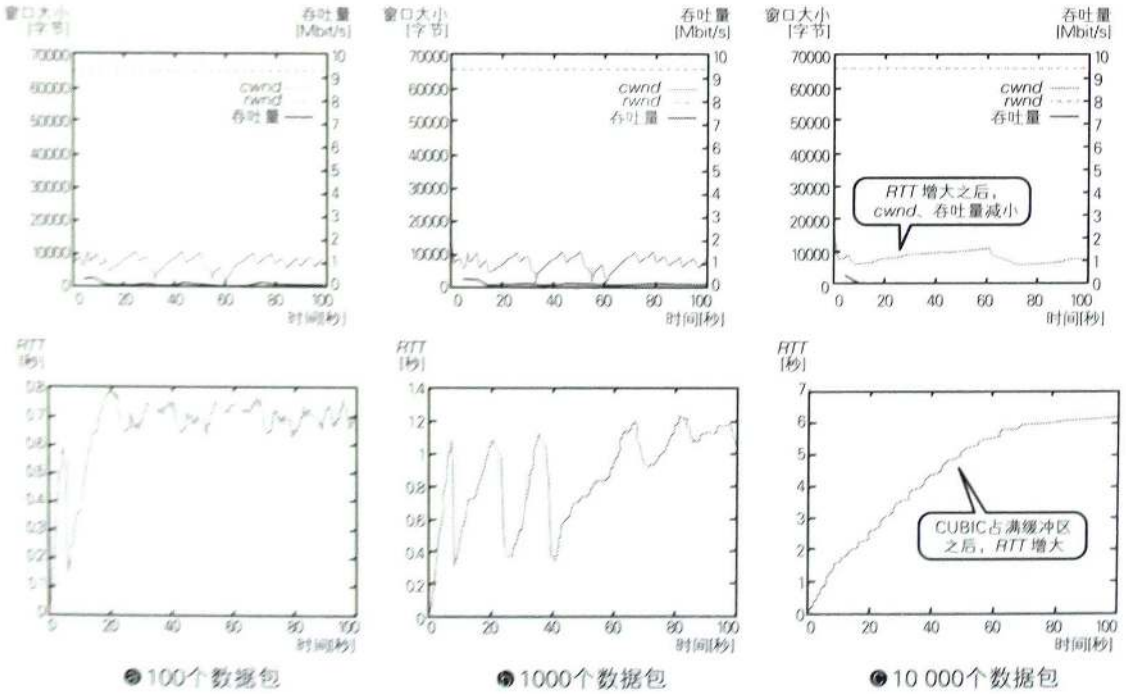


图6.8 模拟结果（实验 13）

最终，拥塞窗口大小降低到 1 万以下，吞吐量降到接近 0 的水平。综上所述，从模拟结果可以看出，Vegas 会受到其他网络流的大幅影响，与基于丢包的拥塞控制算法尤其难以共存，实用性很差。

6.3 BBR 的机制

把控数据发送量与 RTT 之间的关系，实现最大吞吐量

BBR（Bottleneck Bandwidth and Round-trip propagation time，瓶颈带宽和往返传播时延）是近些年开发出来的基于延迟的拥塞控制算法，目前已经默认搭载在 Linux 中，并被广泛使用。毫无疑问，它已经是当前主流的拥塞控制算法之一。

BBR 的基本思路

BBR 与之前介绍的 Vegas 一样，属于基于延迟的拥塞控制算法。

谷歌在 2016 年发布 BBR[①] 之后，Linux 内核也在 4.9 版本以后开始支持它，随后 BBR 在 Google Cloud Platform 等平台上也被广泛使用，引发了很大的关注。此外，谷歌在 YouTube 等服务中也开始使用 BBR，且有报告指出，BBR 实现了较高的吞吐量，并使 *RTT* 减少了 50% 以上[②]。

BBR 的基本思路是，过去主流的基于丢包的拥塞控制算法以发现丢包为契机来判断发生了拥塞的做法过于迟钝，因此它将“数据包被缓存之前”，也就是“网络带宽被占满，但没有缓冲区时延”的状态作为理想状态。

然而，由于无法直接获取链路上网络设备的状态，所以 BBR 只能一直监视吞吐量和 *RTT* 的值，一边把控数据发送量与 *RTT* 之间的关系，一

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.

② Neal Cardwell，Yuchung Cheng，C. Stephen Gunn，et al. BBR Congestion Control [R]. Google Networking Research Summit，2017.