**网络实验报告 exp5**

**2020K8009929032 刘耕印**

1. 重现bufferbloat问题

在TCP拥塞控制算法中，采取的拥塞控制策略是维护一个拥塞窗口，发送数量不能大于拥塞窗口大小。然而，当路由器buffer过大时，发送端往往不能及时得到丢包信息，从而不会控制拥塞窗口大小，导致拥塞情况的大量发生，这就是bufferbloat问题。当bufferbloat问题发生时，数据传输延迟变大，产生延迟抖动，吞吐率也变差。

下面依据给定脚本，使用命令行参数将路由器队列大小上限分别设为0，50，100时，运行脚本，将产生的输出处理后作图。处理过程大概是用C++将每个文件中时间数据和对应的指标数据提取出来，组合成python list的形式，再作为matplotlib的输入运行python脚本画图。代码写得比较丑陋，为避免报告又臭又长，就不贴了。

RTT：

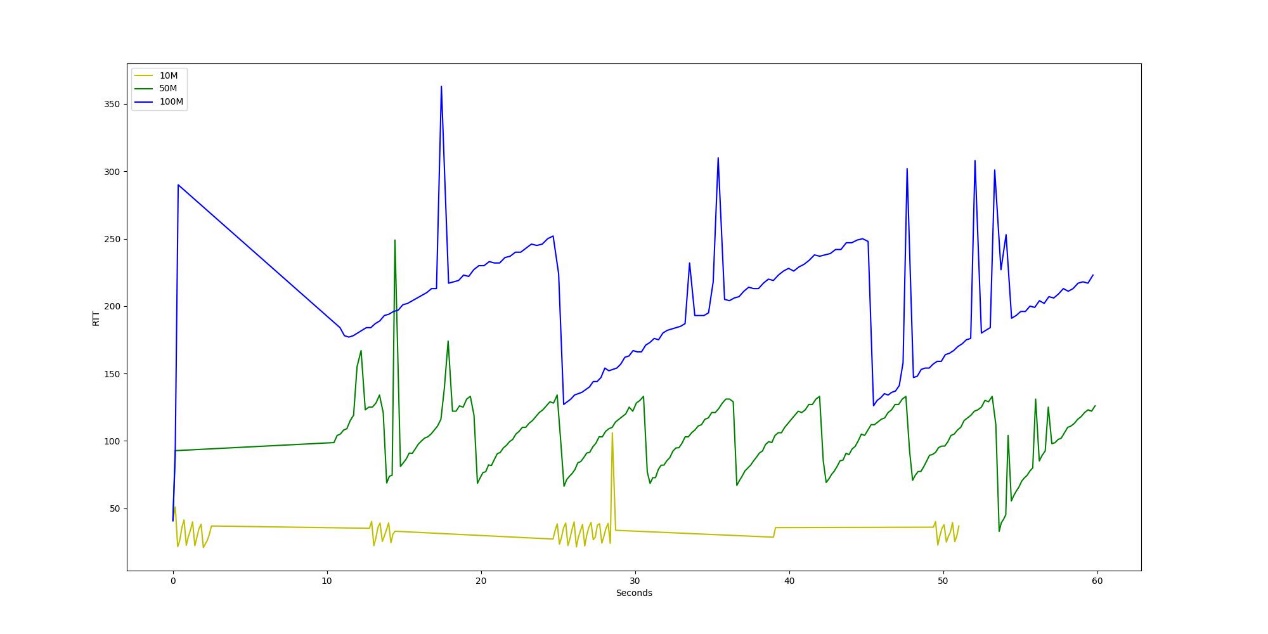


Figure RTT – time

见图例，分别是将队列上限设为10，50，100所绘制出的图线。可以看到，队列上限越大，延迟抖动就越严重。队列上限为100时，延迟有时能够达到300ms以上，对于许多实时性要求较高的工作（如fps游戏）是不可接受的。

QLEN：

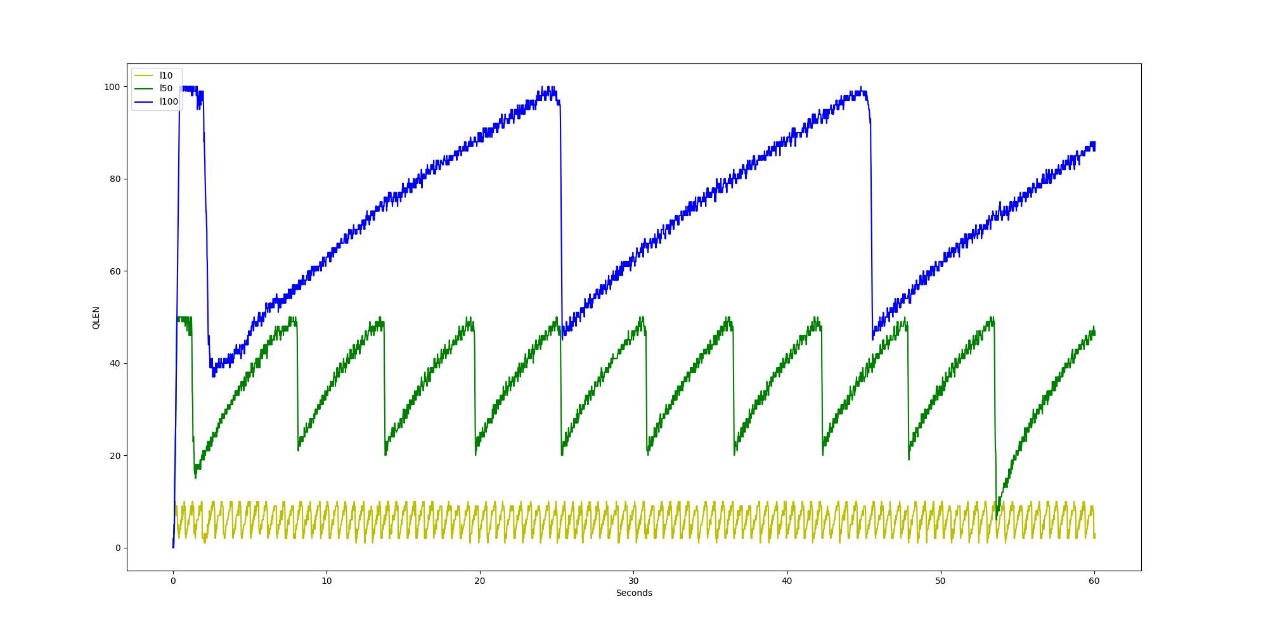


Figure QLEN -- time

成功复现了比较典型的队列长度变化过程。队列长度先上升到峰值，此时TCP拥塞控制开始起作用，发送速率变缓，队列长度迅速减小。没有直接反映出什么问题。

CWND：

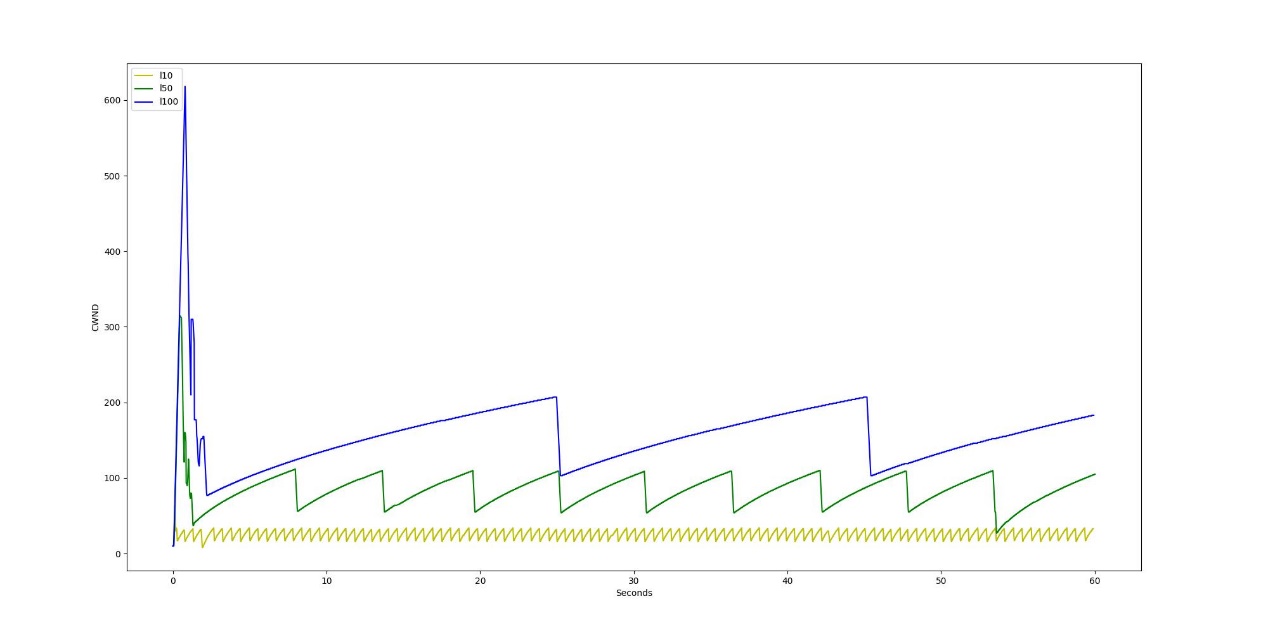


Figure CWND – time

上图展示了发送方拥塞窗口的变化过程。一开始是一个相当剧烈的波动，拥塞窗口大小达到队列上限的约6倍后，开始迅速回落，之后是一系列较小的波动，峰值约有2倍的差距。这些波动很好地体现了TCP拥塞控制算法的不足：检测到拥塞后才控制拥塞。

IPERF\_BW：

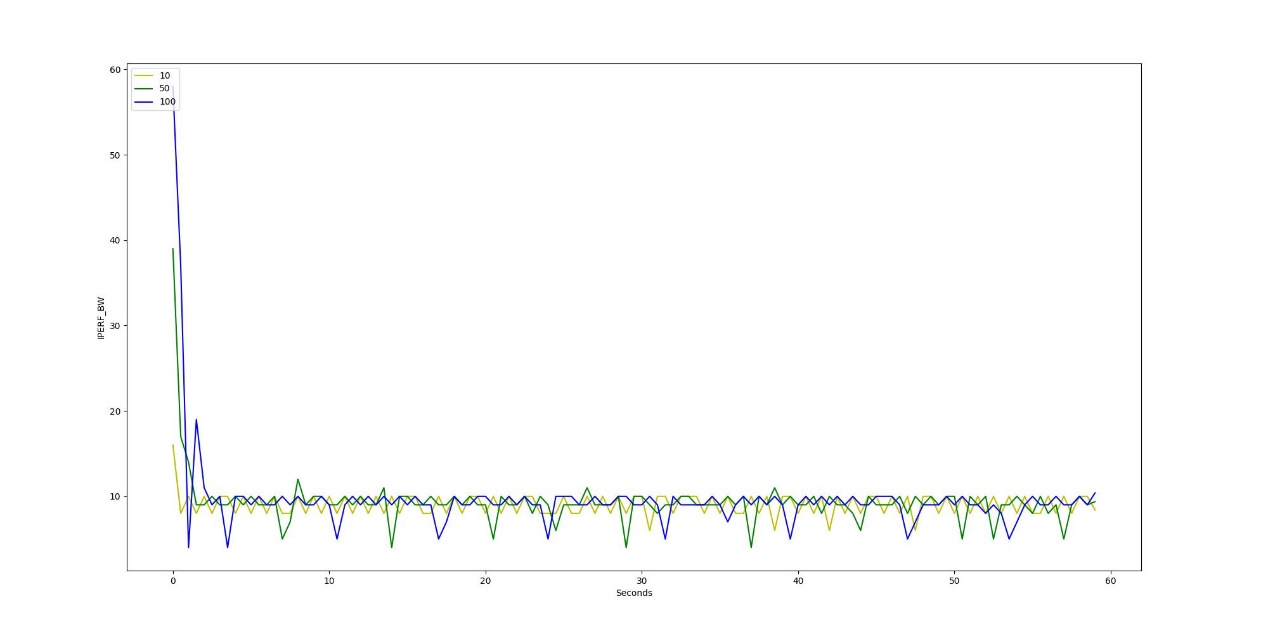


Figure BW – time

可以看到，吞吐率从峰值迅速跌落，并维持在一个较低的水平。这也是bufferbloat的影响之一。

以上四张图表成功复现了bufferbloat问题。

1. 解决bufferbloat问题

在另一个脚本中，通过命令行参数指定taildrop，red和codel三种方法来尝试解决bufferbloat问题。taildrop不加特殊处理，red按计算的概率主动丢包，codel限制包排队时间。将三种方法的延迟曲线绘图如下，其中纵坐标采取对数坐标：

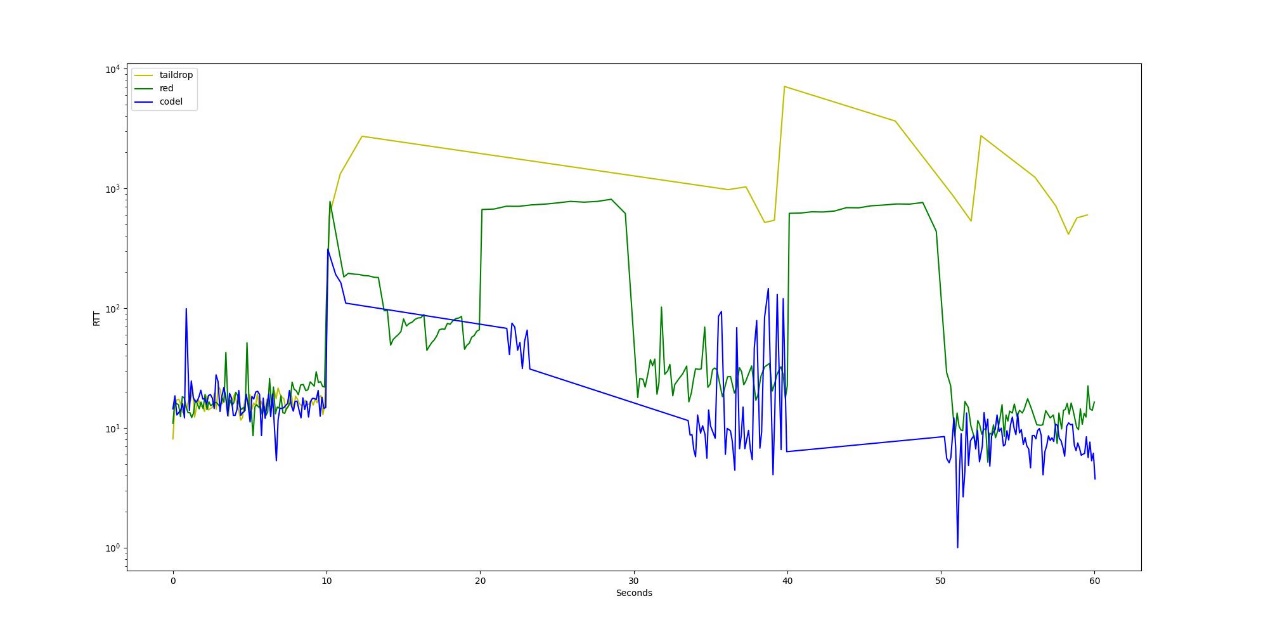


Figure 三种方法

效果和课件上的标准效果有些出入，图形并不是很平整，推测是本地环境不规范导致。但是可以看出，三种方法中，虽然都有波动，但red和codel的效果远好于taildrop，同时codel的效果好于red，可能是red的参数配置不佳。

本次的实验报告正文比较短，但工作时长还是不少的，数据处理起来着实花了一番工夫。

调研：两种新型拥塞控制

BBR和HPCC

传统的拥塞控制算法是基于数学上收敛的模型，而没有探测并结合实际情况以达到最优策略。比如，CoDel算法的本意是消除“谨慎探测——过激降速“循环所带来的锯齿状图线（把图线所围面积看作效果，可以发现效率比较低），但它并没有做到。于是出现了BBR算法，采集真实的带宽和RTT数据以便调整窗口。BBR算法并不会把队列缓存当作是接收端的处理能力的一部分——在稳定过程中，缓存基本上是空的。BBR采集到不排队的RTT作为其最小RTT，在一段时间内，坚持此最小RTT不变。根据这个RTT以及历史最大带宽，BBR算法可以控制数据包基本不排队（通过发送pacing rate），也就基本上不丢包，消除了锯齿状的图形，bufferbloat自然也解决了。另外，当多于一个流到来时，BBR算法检测不到最小RTT，会进入probeRTT状态重新采集最小RTT，最终退出此状态时所采集到的最小值不会包含排队时延。因此可以继续保证不排队。

HPCC算法是一种最新型的拥塞控制算法，它在网络上各个路由器中部署专用硬件，以试图实时精确地获知网络各个部分的拥塞情况。网络设备可以向源结点专门发送通知，通知该处的拥塞情况；同时路由器本身也有能力通过阻断某些数据流来在路由器处进行拥塞控制，脱离了传统算法只能在端设备处进行拥塞控制的局限性。