数据包队列管理实验

2020年11月15日

蔡润泽

本实验 `Github` 地址

实验内容

实验内容一

重现Bufferbloat结果

- h1(发送方)在对h2进行iperf的同时，测量h1的拥塞窗口值(cwnd)、r1-eth1的队列长度(qlen)、h1与h2间的往返延迟(rtt)
- 变化r1-eth1的队列大小，考察其对iperf吞吐率和上述三个指标的影响

实验内容二

解决BufferBloat问题

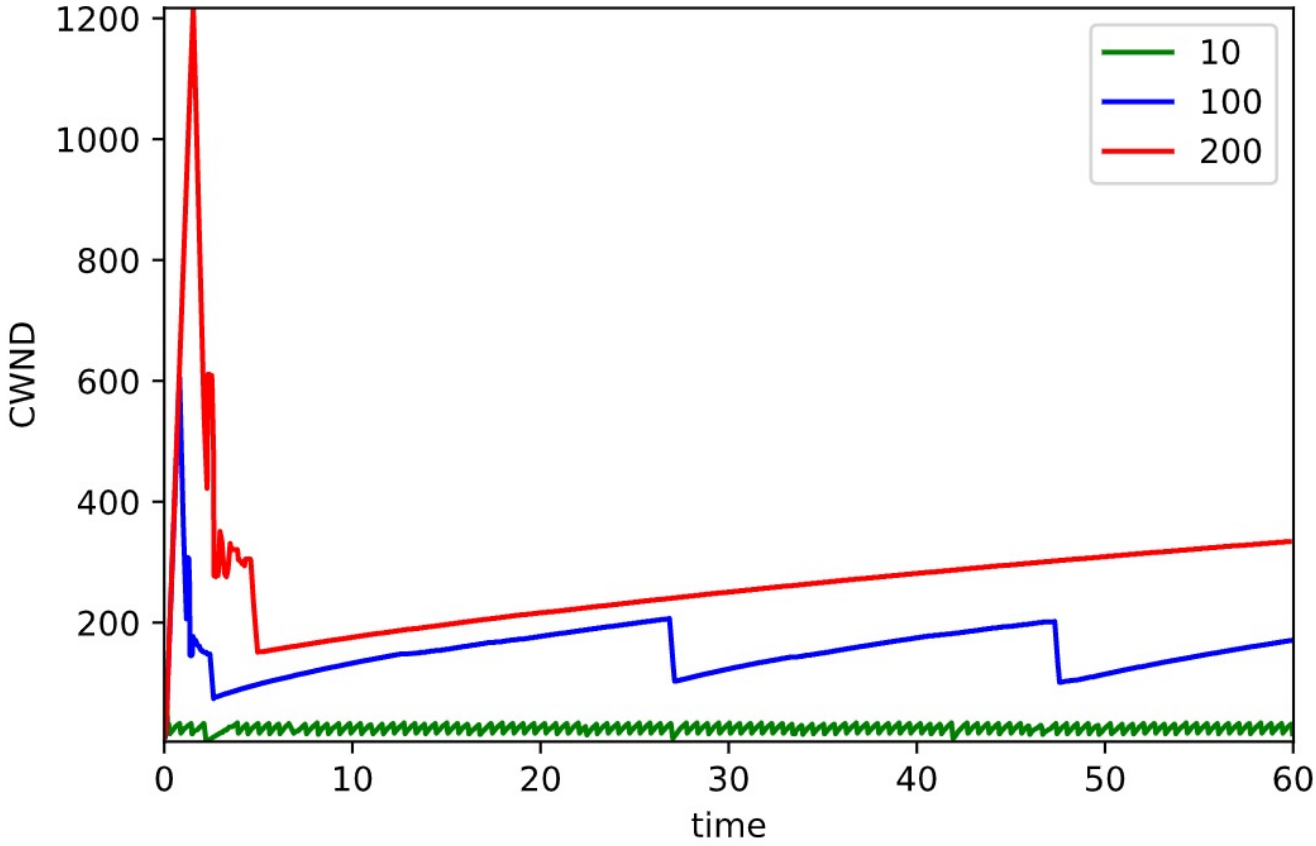
- Tail Drop
- RED
- CoDel

数据处理及结果

重现Bufferbloat结果

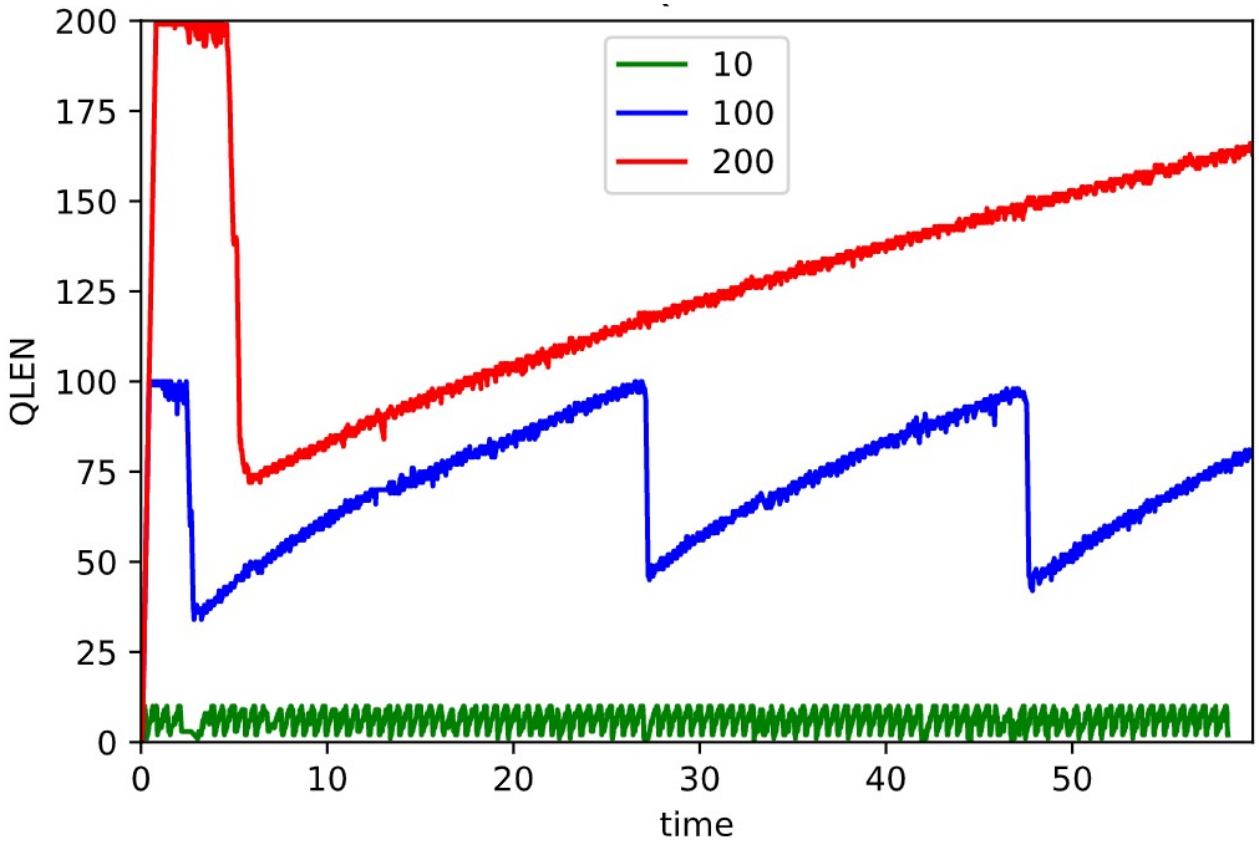
处理的可视化脚本位于 `10-bufferbloat/reproduce_data.ipynb` 中，实验根据获取的数据，利用Matplotlib画图，得到了下列结果。

CWND



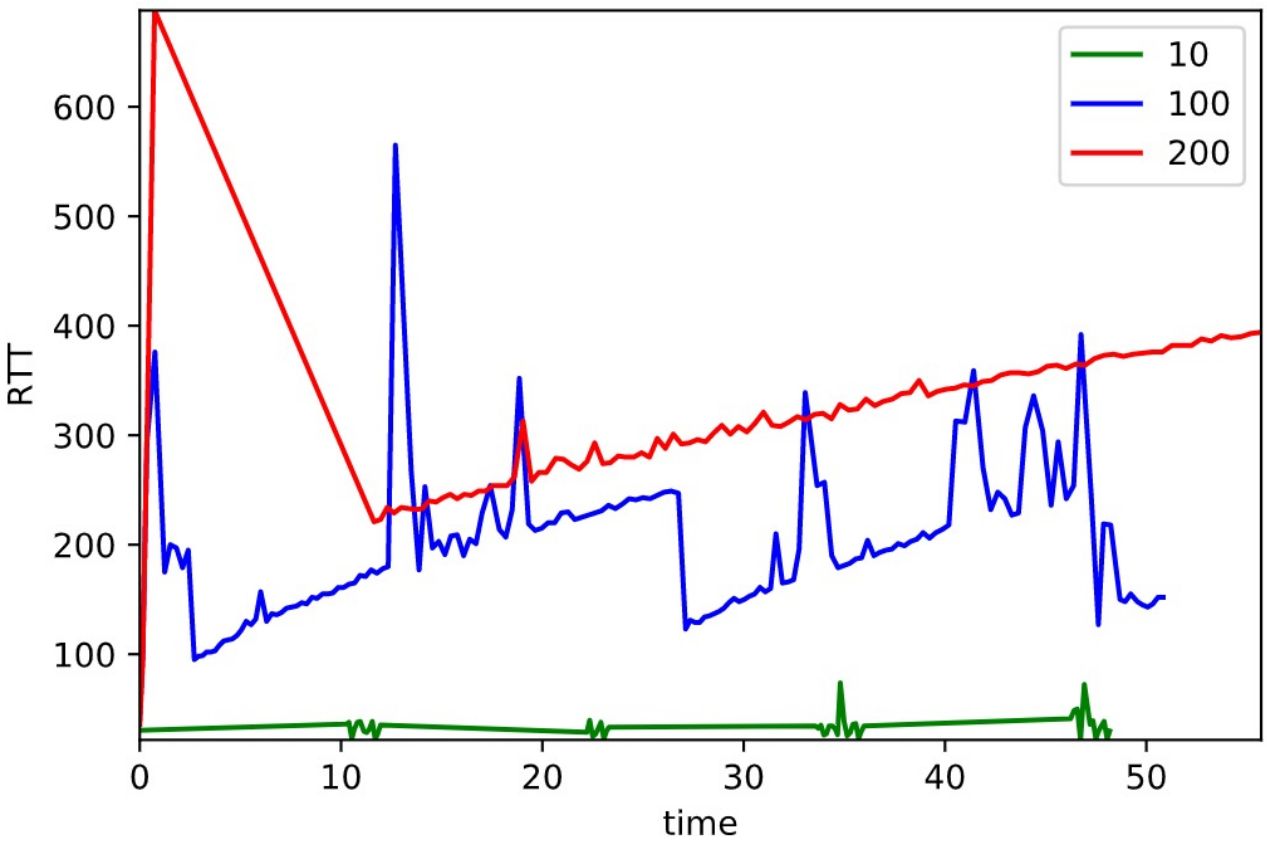
上图为拥塞窗口大小随时间变化的曲线。测试刚开始时，队列大小为 `100` 的拥塞窗口迅速增长达到 `600` 的峰值，队列大小为 `200` 的的拥塞窗口迅速增长达到 `1200` 的峰值。可见该实验环境下，拥塞窗口峰值大小为队列大小的两倍。
接着拥塞窗口迅速回落到队列大小一致的水平，接着再次增长。拥塞窗口随收包增长，随丢包下降。另外可以看到大队列后续的波动较小队列不明显。

QLEN



上图为即时队列长度随时间变化的曲线。队列大小为 `100` 的曲线的上界为 `100` 。测试刚开始时队列快速上升达到峰值，当队列满时，直接丢弃新接收的包，由于发送速率骤降，使得队列长度降低至最大队列长度的40%左右。随后一直这次循环下去。

RTT



上图为RTT随时间变化的曲线。测试刚开始时，由于接收队列的快速增长，网络延时急剧增加，随后也随着队列缩短而减少。之后的时间内，延时随着队列长度的变化而变化，波动明显。另外可以看到大队列后续的波动较小队列不明显。

iperf 吞吐率

队列大小`200`时，iperf Transfer成功的带宽平均大约为29.4Mb/s，峰值为37.7 Mb/s，有`2/3`的时间区间会下降到0。

队列大小`100`时，iperf Transfer成功的带宽平均大约为15Mb/s，峰值为37.7 Mb/s，有`1/3`的时间区间会下降到0。

队列大小`10`时，iperf Transfer成功的带宽平均大约为9Mb/s，峰值为12.6 Mb/s，最低为6.9Mb/s。

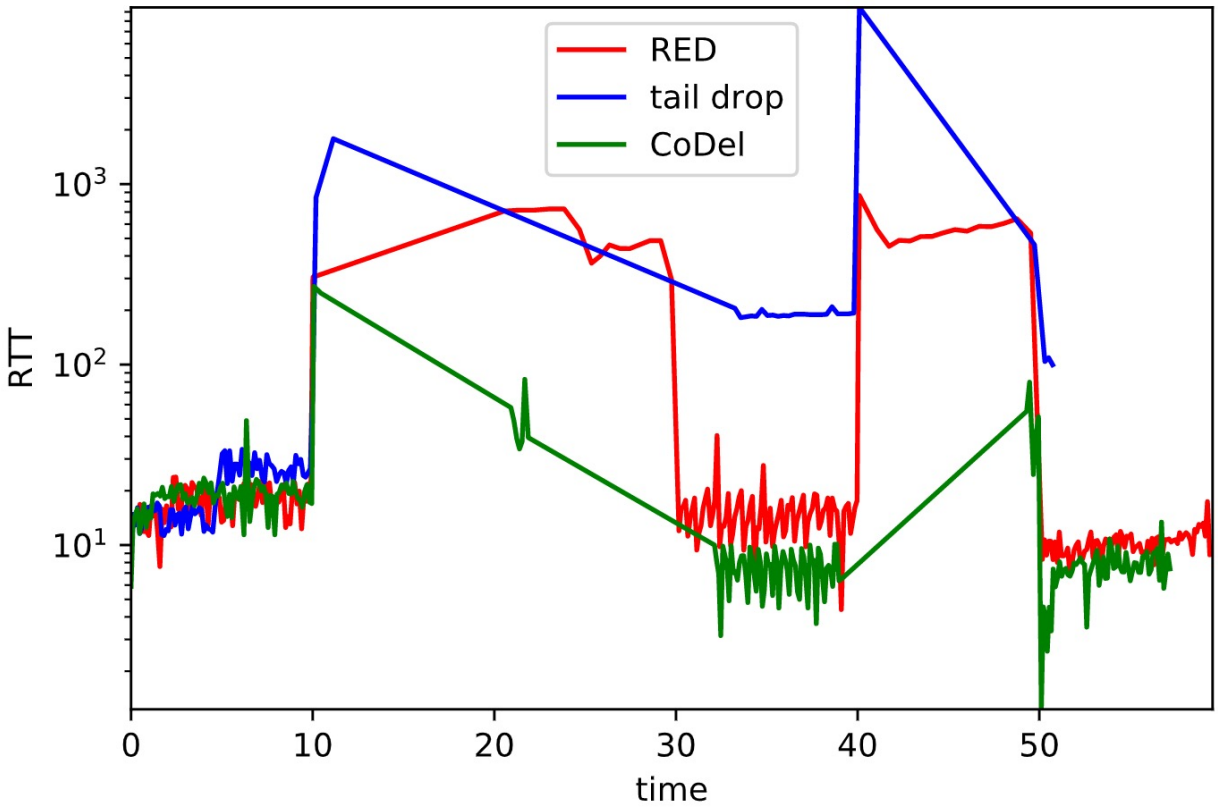
因此可以看出队列越长，iperf Transfer 为0的可能性越大，但是Transfer成功时的带宽也越大。

总结分析

接收队列的长度受到拥塞窗口的影响，因其长度取决于发送方的发包速率，而发包速率又同时受到滑动窗口和拥塞窗口的制约。因此，当拥塞窗口较大时，可以实现更高的发送速率，接收队列的增长也会随之加快。但是从实验结果中可以看到随着拥塞窗口增大，接收队列增速反而减缓，这可能是因为时延增大引起的。

另外，随着接收队列增长，数据包的接收延时会线性增长。当队列长度很长时，会产生BufferBloat问题。tail drop机制可以减少延时。但tail drop启动时，网络会产生BufferBloat问题，而且这一问题会随着最大接收队列长度的增加而增加。但减小最大队列长度也会导致不停地重启，从而影响传输效率。还可能引发TCP Incast问题。因此需要改进此方法来解BufferBloat问题。

解决BufferBloat问题



上图中出现了三种方式来BufferBloat问题的曲线。

可以看出，tail drop的队列延时远高于RED和CoDel。其中CoDel的性能表现最佳。

另外值得注意的是，tail drop曲线一直出现“尖端”情况，和ppt中稳定的一段峰值不同，这是因为Mininet环境下仿真环境的差异导致的。在Mininet中，maxq参数实际上是一个模拟延迟和丢包的批处理大小，与真实的队列大小有区别，只是近似仿真。因此结果存在差异。

调研分析

BBR

BBR算法主要有以下特点：

即时速率的计算

计算一个即时的带宽，该带宽是BBR一切计算的基准，BBR将会根据当前的即时带宽以及其所处的pipe状态来计算pacing rate以及cwnd，这个即时带宽计算方法的改进是BBR简单、高效的根源。BBR运行过程中，系统会跟踪当前为止最大的即时带宽。

RTT的跟踪

BBR可以获取非常高的带宽利用率，是因为它能探测到带宽的最大值以及rtt的最小值，这样计算出来的BDP就是目前为止TCP管道的最大容量。BBR的目标就是达到该最大的容量。这个目标驱动了cwnd的计算。在BBR运行过程中，系统会跟踪当前为止最小RTT。

BBR pipe状态机的维持

BBR算法根据互联网的拥塞行为有针对性地定义了4中状态，即STARTUP，DRAIN，PROBE_BW，PROBE_RTT。BBR通过对上述计算的即时带宽bw以及rtt的持续观察，在这4个状态之间自由切换，相比之前的所有拥塞控制算法，其改进在于其拥塞算法不再跟踪系统的TCP拥塞状态机，而是用统一的方式来应对pacing rate和cwnd的计算。

结果输出-pacing rate和cwnd

BBR的输出并不仅仅是一个cwnd，更重要的是pacing rate。在传统意义上，cwnd是TCP拥塞控制算法的唯一输出，它只规定了当前的TCP最多可以发送多少数据，它并没有规定怎么把这么多数据发出去。在Linux的实现中，在忽略接收端通告窗口的前提下，Linux会把cwnd一窗数据全部突发出去，而这往往会造成路由器的排队，在深队列的情况下，会测量出rtt剧烈地抖动。BBR在计算cwnd的同时，还计算了一个与之适配的pacing rate，该pacing rate规定cwnd指示的一窗数据的数据包之间，以多大的时间间隔发送出去。

通过上述特点的控制BBR能很好地调度资源，从而解决Bufferbloat问题

HPCC

HPCC是对现有的拥塞控制的一种替代方案。它可让网络中的报文稳定的、以ms级的延迟传输。

当前主流的拥塞控制算法主要依赖于端的信息（e.g.丢包信息，RTT）做拥塞控制，而HPCC则运用了网络设备提供的细粒度负载信息而全新设计了拥塞控制算法。

拥塞算法控制使其解决Bufferbloat问题的核心。

传统拥塞控制主要通过在端上调节流量，以维持网络最佳平衡状态。发送方根据网络承载情况控制发送速率，以获取高性能，避免拥塞崩溃导致网络性能下降几个数量级，并在多个数据流之间产生近似最大化最小流的公平分配。发送方与接收方确认包、包丢失以及定时器情况，估计网络拥塞状态，从而调节数据流的发送速率，这被称为网络拥塞控制。

而HPCC的核心理念是利用精确链路负载信息直接计算合适的发送速率，而不是像现有的 TCP 和 RDMA 拥塞控制算法那样，通过`慢启动`等方法，迭代探索合适的速率；HPCC 速率更新由数据包的 ACK 驱动，而不是像 DCQCN 那样靠定时器驱动。