数据包队列管理实验报告

李昊宸

2017K8009929044

（一）BufferBloat复现实验

一、实验内容

1. 调研BufferBloat的成因

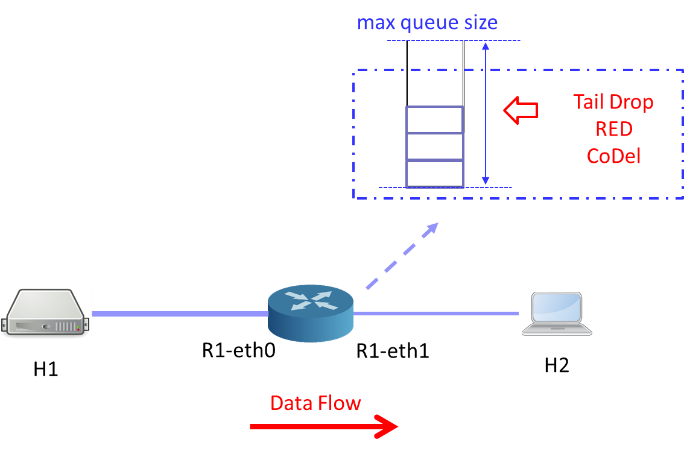
2. 使用reproduce\_bufferbloat.py脚本复现BufferBloat过程

3. 解释现象

二、实验流程

1. 搭建实验环境

实验用到的脚本reproduce\_bufferbloat.py工作图如下：



图一 连接建立拓扑图

通过在python内部调用addHost生成三个主机，其中中间主机r1充当路由器的角色。h1到r1之间通道为5ms延迟，r1到h2间通道也为5ms延迟，带宽10mbps，但对其规定了最大队列数maxq。

使用Tcp\_probe工具测试h1与h2之间的TCP连接质量，加载方式： # modprobe tcp\_probe port=5001 full=1。数据保存在cwnd.txt中。

调用utils.py中的工具qmon监测队列长度，数据保存在qlen.txt中。

使用iperf工具测试h1与h2之间的连接质量（ping），数据保存在ping.txt中。

2.启动脚本

sudo python reproduce\_bufferbloat.py -q 10

sudo python reproduce\_bufferbloat.py -q 50

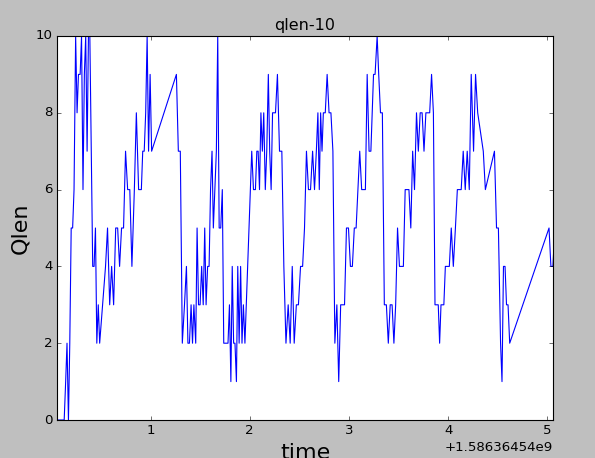
sudo python reproduce\_bufferbloat.py -q 100

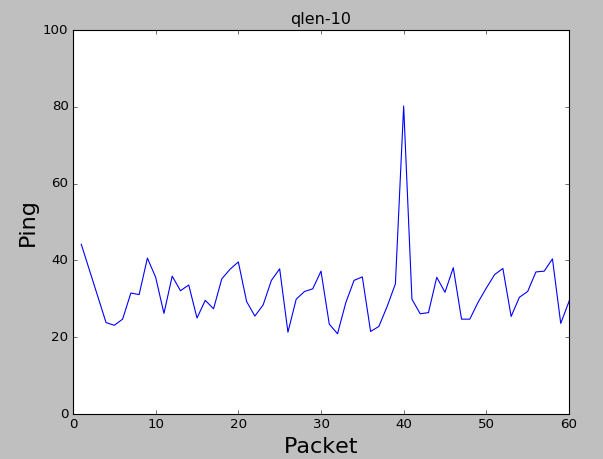
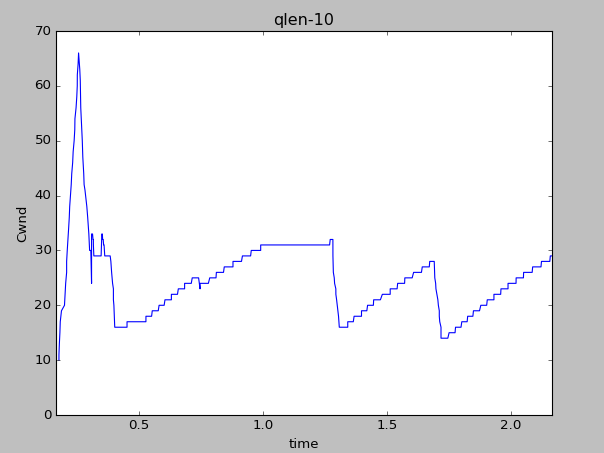
-q后的传参表示设置最大队列大小

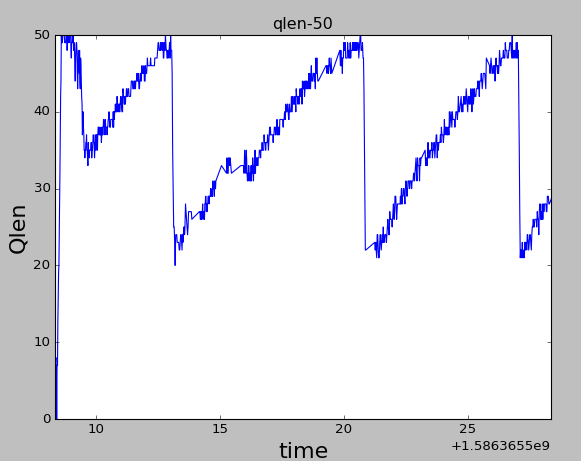
3.绘制折线图

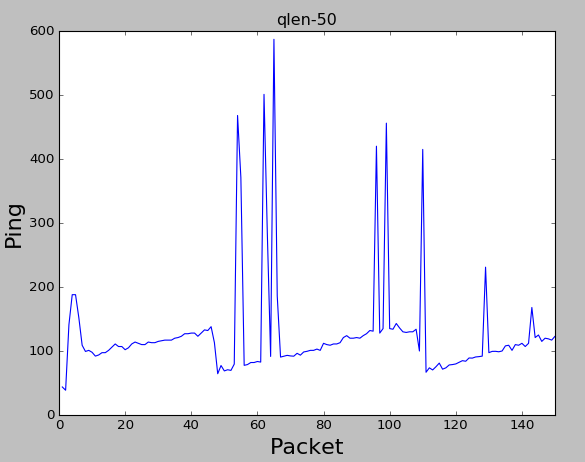
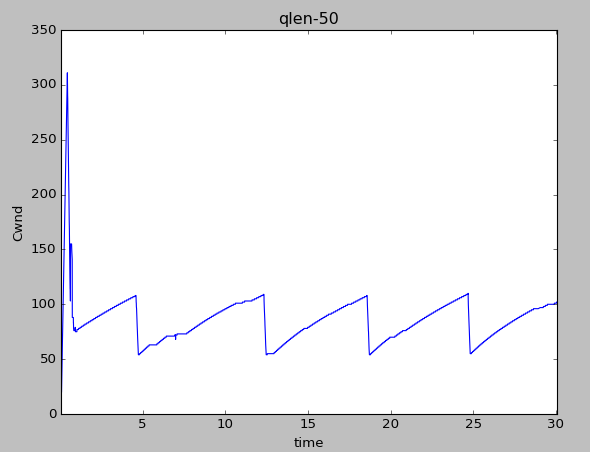
用python脚本处理得到的数据，分别绘制三个不同队列长度下的Cwnd-time，Qlen-time和Ping-packet折线图。

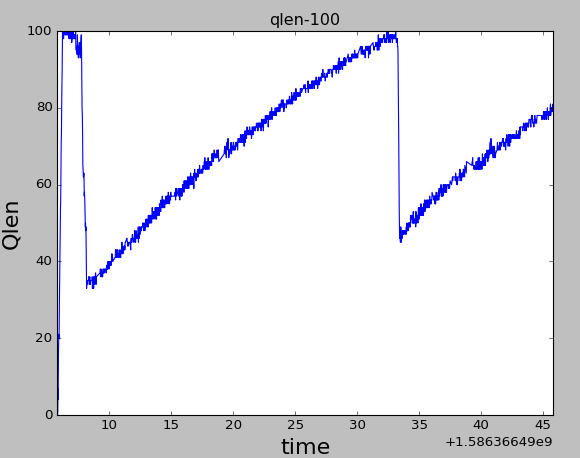
三、实验结果及分析

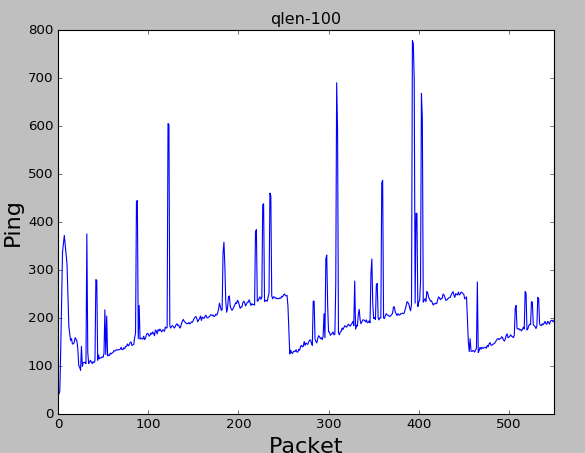
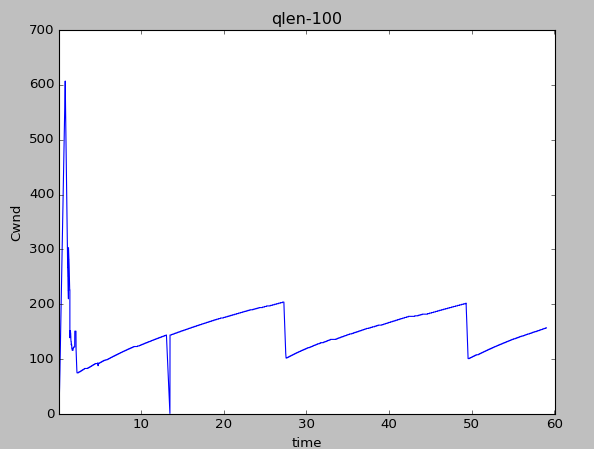
1. 实验结果



图二 最大队列为10时的Qlen-time（左），Ping-packet（中） 和Cwnd-time（右）折线图



图三 最大队列为50时的Qlen-time（左），Ping-packet（中） 和Cwnd-time（右）折线图



图四 最大队列为100时的Qlen-time（左），Ping-packet（中） 和Cwnd-time（右）折线图

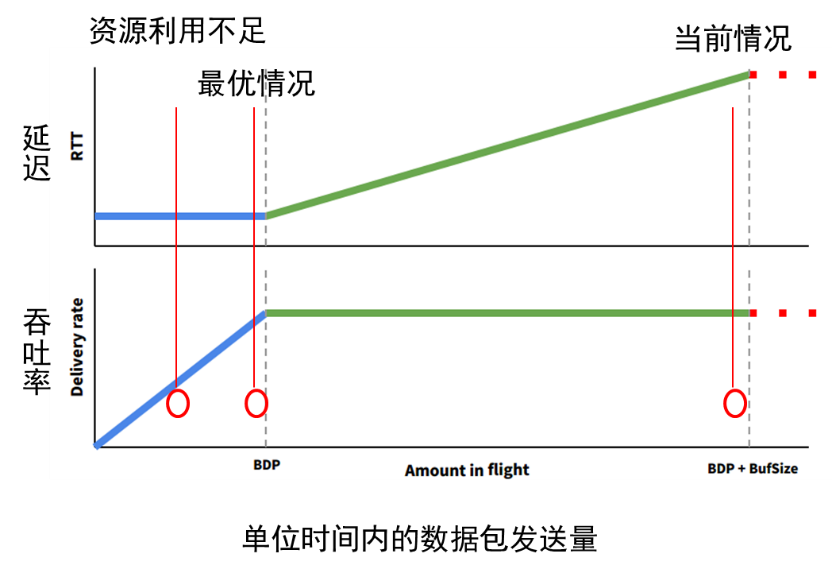
2. 实验分析

BufferBloat：BufferBloat是指数据包在缓冲区中存留时间过长引起的延迟过大问题。通常发生在：

1. 网络负载较重的时间段，（不会一直持续）

2. 边缘网络，（该部分的队列大小更容易被错误配置）

3. 3G/4G网络，（运营商为了提升QoS等更容易部署大量队列）



图五 BufferBloat的产生本质

BufferBloat的产生本质：在吞吐率达到BDP之前，线路未被填满，延迟主要为传播延迟，故RTT维持在稳定的较低水平。当吞吐率达到BDP之后，如果继续增大单位时间内的数据包发送量，线路已经被填满，故多出来的数据包就会进入缓冲区缓存，等到前面的数据到达后继续发送，此时延迟为传播延迟加排队延迟。因为直到缓冲区满线路发生丢包，才会回馈信息，告诉发送方减缓发送速率，所以如果缓冲区特别大，进入到缓冲区的数据包就要经历相当长的排队时间，导致ping值迅速升高。

复现的三种折线图分别体现了三种概念：Qlen，Ping和Cwnd。下面逐一阐述这三种概念。

Qlen：Queue length，队列长度，表示中介主机的缓冲区当前拥有缓存的个数，也即正排队等待发送的队列长度。

观察图二图三图四的左图，可以发现，起初缓存区为空，TCP协议采用慢启动方式逐渐加快发包速度，导致Qlen曲线不断上升，此阶段表示从h1发送数据到r1的速度高于r1向h2发送数据的速度。

当Qlen等于Max queue size发生丢包时，h2会向h1反馈报文，要求h1降低发送速率，随后h1发送数据到r1的速度低于r1向h2发送数据的速度，Qlen曲线下降。

当Qlen下降到一定程度（低于缓冲区）后，r1向h1发送数据的速率会试探性提高，随后便重复以上过程。

可以发现，当maxq为10时，波动周期为0.6s；maxq为50时，波动周期为7.5s；maxq为100时，波动周期为25s。波动周期与maxq成平方正比关系。

Ping：Packet Internet Groper，因特网包探索器。通常情况下我们所指的ping值应该说的是RTT。RTT，Round-Trip Time，往返时延。

观察图二图三图四的中图，可以发现，起初缓存区为空，TCP协议采用慢启动方式逐渐加快发包速度，这段时间内单位时间发送量还未达到BDP，故RTT维持在较低水平。

随着h1发送速度不断提高，h1到r1的发送速度超过r1到h2的发送速度，Qlen从0开始逐渐增长，反映在RTT曲线上为逐渐升高的部分。当缓冲区即将被填满时，RTT曲线到达峰值，此时Ping达到正常情况下的最大值。

随后当Qlen等于Max queue size的过程中，会出现一些packet RTT值非常大的极端现象。这是因为当Qlen等于Maxq时，会发生丢包，部分packet被丢弃，故无法到达h2，导致Ping不通。此时h2察觉丢包，发回报文要求h1减缓发送速率。

h1向r1发送速率低于r1向h2发送速率，Qlen逐渐降低，之前积累的packet也逐渐被发送出去，链路维持在较为畅通的状态，体现在RTT曲线上为迅速下降后趋于最低点，最低点的理论值为传播时延。

当Qlen下降到一定程度（低于缓冲区）后，r1向h1发送数据的速率会试探性提高，RTT曲线在平稳过后也开始上升，随后便重复以上过程。

Cwnd：Congestion Window，拥塞窗口。拥塞窗口的设计由两部分算法完成：“慢速启动”算法和“拥塞避免”算法。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。发送方会维持一个拥塞窗口,刚开始的拥塞窗口和发送窗口相等,一般开始均设置1，然后我们每收到一个确认,就让拥塞窗口大小变为原来的两倍，接着发送分组也是原来的两倍，以此类推。

当窗口值等于16（慢开始门限ssthresh初始值），然后我们开始采用“拥塞避免”的策略，即不在以2倍的方式增加，而是转变为每次加1的方式。

直到网络拥塞时，让新的慢开始门限值变为发生拥塞时候的值的一半，将拥塞窗口置为1，然后让它再次重复，这时一瞬间会将网络中的数据量大量降低。

也就是说：

1）当cwnd<ssthresh时，使用慢开始算法。

2）当cwnd>ssthresh时，改用拥塞避免算法。

3）当接收方cwnd=ssthresh时，慢开始与拥塞避免算法任意。

4）无论当前处于慢开始算法还是拥塞避免算法，当出现网络拥塞时，就将ssthresh置为当前发送窗口的一半，然后将拥塞窗口置为1。

这样做的目的是要迅速减少主机发送到网络中的分组数，使得发生 拥塞的路由器有足够时间把队列中积压的分组处理完毕。

观察图二图三图四的右图可以发现，Cwnd的变化趋势与Qlen和RTT的变化趋势同步。当Qlen值较低时，队列较空，packet的排队时延较短，Ping较低，发送的包可以迅速得到反馈，此时如果Cwnd还处于较低水平，就执行慢开始迅速增长到ssthresh的水平。到达之后，执行拥塞避免算法，Cwnd呈线性增长，这与图中线性上升的现象一致。

当Qlen即将到达Maxq时，Ping值到达极大，并且伴有丢包事件发生。发送端检测到Ping不通，立即将Cwnd设置为1，设置慢开始门限ssthresh为当前发送窗口大小的一半，重新慢开始算法。此时由于发送瞬间停止，Buffer中的数据包迅速被发送出去，此时线路会处于良好的状态，在图中表现为Ping和Qlen的极小值。随后不断重复以上过程。

此外需要说明的是，使用tcp\_probe监控流量，需使用命令# modprobe tcp\_probe port=5001 full=1，监测端口5001的连接，每个数据包到达时都记录相关参数。

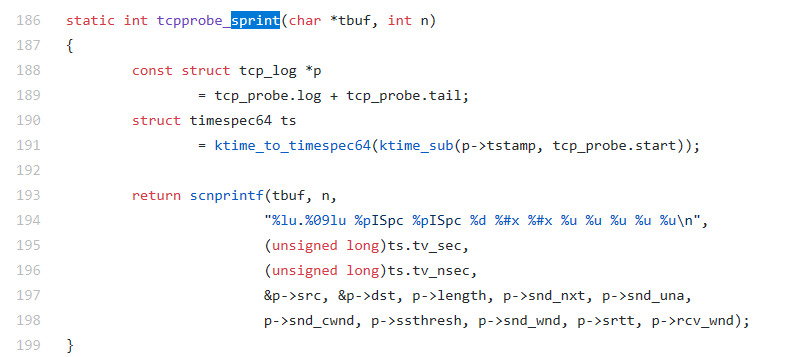
GitHub上Linux仓库已经不再留有v4.9版本了，经后续寻找我找到了Linux 4.9版本的源码并fork到了我的GitHub账号下。有需要的可以访问

<https://github.com/Therock90421/linux-intel-4.9>

有了源码就可以参考tcp\_probe.c中抓取数据包中的参数。

路径：

内容：



图六 tcp\_probe打印的参数

从左到右，依次为：时间（秒）[ts.tv\_sec].时间（微秒）[ts.tv\_nsec]，源地址：端口[&p->src]，目的地址：端口[&p->dst]，发送长度[p->length]，下一发送序列号[p->snd\_nxt]，已确认序列号[p->snd\_una]，发送端拥塞窗口[p->snd\_cwnd]，慢启动门限[p->ssthresh]，发送窗口[p->snd\_wnd]，平滑往返延迟[p->srtt]，接收窗口[p->rcv\_wnd]。

（二）BufferBloat解决实验

一、实验内容

1. 根据mitigate\_bufferbloat.py脚本，复现出三种不同策略下BufferBloat问题的解决情况

2. 画出对比折线图

二、实验流程

1. 搭建实验环境

实验用到的脚本mitigate\_bufferbloat.py结构与上一实验较为类似，设置h1与r1之间带宽100Mbps，最大队列数为100；h2与r1之间带宽设置为100mbps。与上一实验不同的是，由于本次我们主动采用不同的算法解决BufferBloat问题，需要调用set\_qdisc\_algo函数。函数内容如下：

def set\_qdisc\_algo(net, algo):

algo\_func\_dict = {

'taildrop': [],

'red': ['tc qdisc add dev r1-eth1 parent 5:1 handle 6: red limit 1000000 avpkt 1000'],

'codel': ['tc qdisc add dev r1-eth1 parent 5:1 handle 6: codel limit 1000']

}

if algo not in algo\_func\_dict.keys():

print '%s is not supported.' % (algo)

sys.exit(1)

r1 = net.get('r1')

for func in algo\_func\_dict[algo]:

r1.cmd(func)

虽然mininet已经封装了tc功能的大部分算法，但是red和codel机制未封装，需要手动配置（见algo\_func\_dict中的red和codel）

另外，为了模拟真实情况，现实情况的带宽会不断波动，脚本中也给出了带宽动态变化函数并在运行时调用。代码如下：

def dynamic\_bw(net, tot\_time):

h2, r1 = net.get('h2', 'r1')

start\_time = time()

bandwidth = [100,10,1,50,1,100]

count = 1

while True:

sleep(tot\_time/6)

now = time()

delta = now - start\_time

if delta > tot\_time or count >= 6:

break

print '%.1fs left...' % (tot\_time - delta)

h2.cmd('tc class change dev h2-eth0 parent 5:0 classid 5:1 htb rate %fMbit burst 15k' % bandwidth[count] )

r1.cmd('tc class change dev r1-eth1 parent 5:0 classid 5:1 htb rate %fMbit burst 15k' % bandwidth[count] )

count += 1

return

2. 启动脚本

sudo python mitigate\_bufferbloat.py -a taildrop

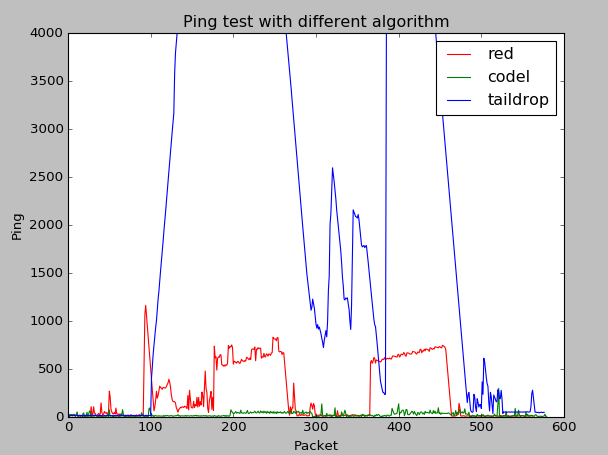
sudo python mitigate\_bufferbloat.py -a red

sudo python mitigate\_bufferbloat.py -a codel

-a表示参数为选择何种算法

三、实验结果及分析

1. 实验结果



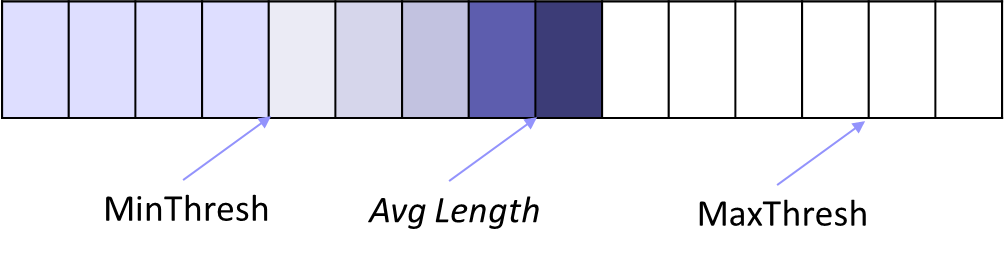
图七 动态带宽下不同算法的每个包往返延迟时间

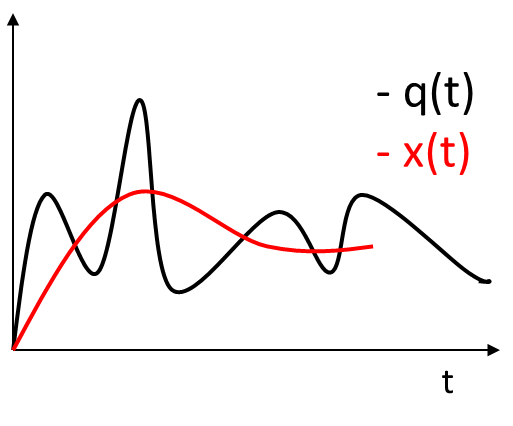
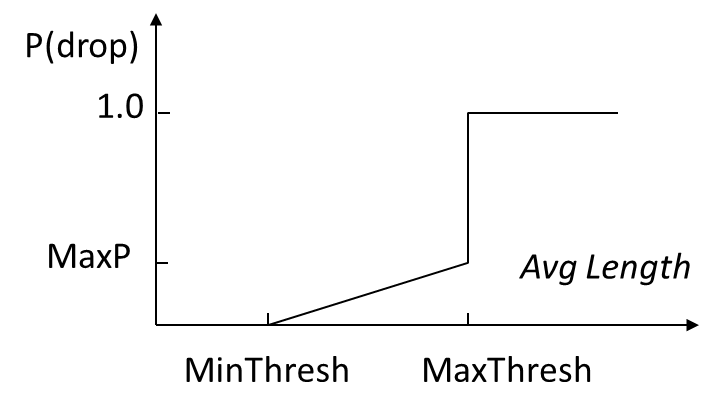
2. 实验分析

首先介绍下使用到的三种算法：

TailDrop算法：尾部丢弃算法，默认队列管理机制。该算法的核心为，排队策略配合FIFO使用（First in first out，先进先出策略），当队列满时，将新到达的数据包丢弃。算法原则为使中间设备的功能尽可能简单，由端设备负责拥塞控制，是最简单、应用最广泛的队列管理机制。但存在的主要问题是当出现拥塞时反馈有后延时性，即当接收方开始丢包并通知发送方减慢速率时，网络状态已经很拥塞了，该算法没有主动避免拥塞的功能。

RED算法：Random Early Detection，随机早期检测算法。算法的设计思路为，在队列满之前，就开始主动概率性丢包，丢包概率与队列长度正相关。



。

图八 RED算法示意

首先使用平滑函数计算平均队列长度x(t):x(t) = (1-Wq)\*x(t-T) + Wq\*q(t)。

随后根据平均队列长度x(t)的大小和两个阈值之间的关系进行概率丢包。

操作的对象均为下一个新到达的分组。

早期的RED算法只设置一个阈值，这导致流量会在阈值附近产生不断抖动，修改后的RED采用双阈值的缓冲区间，一定程度上缓和了抖动程度。

该算法的主要问题是参数设置调优很困难，设置不当可能导致性能十分低下。

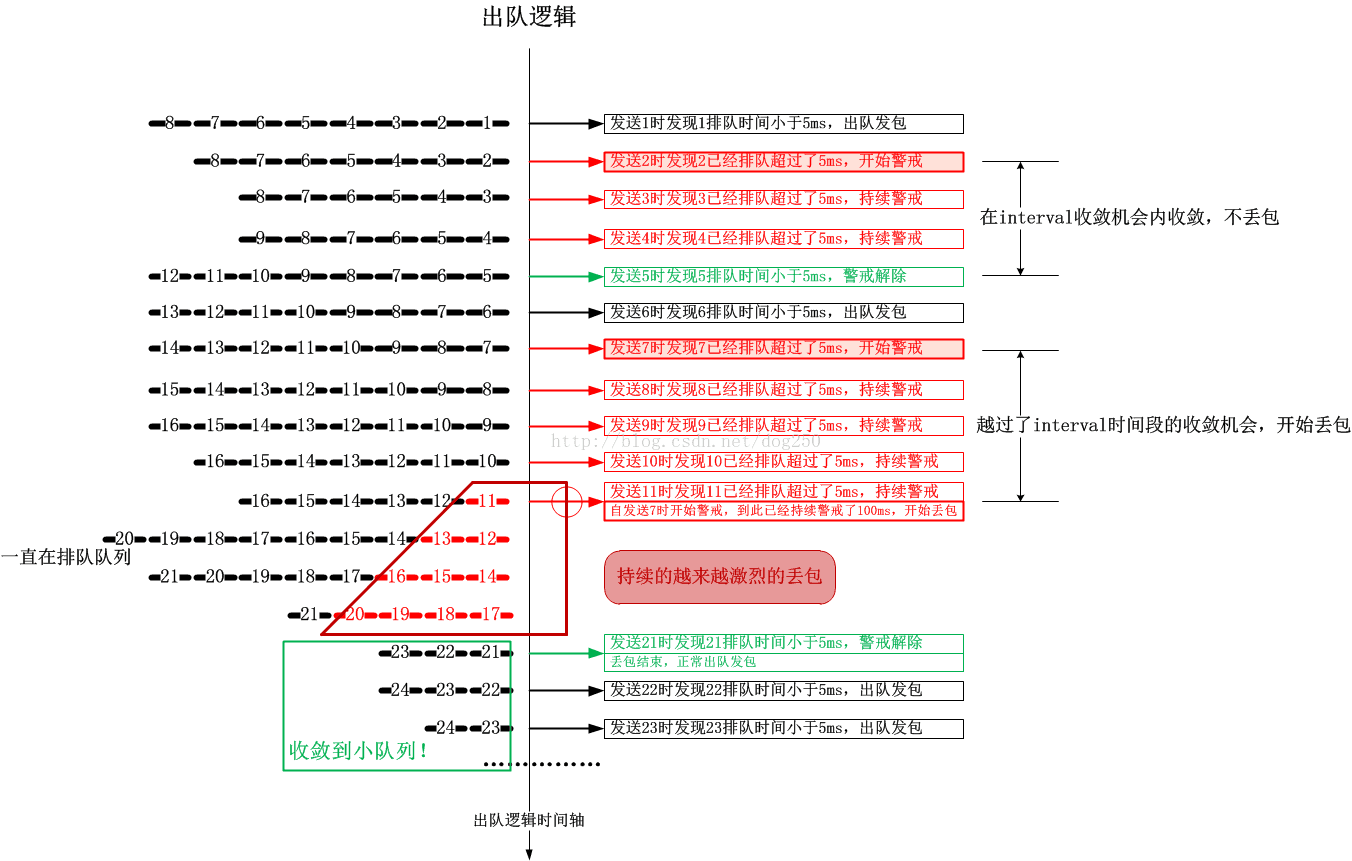
CoDel算法：Controlled Delay，控制时延算法。该算法的核心思想为控制数据包在队列中的时间，而非队列长度。也就是说使用数据包在队列中的停留时间作为度量指标，简单来说包出队时，发现包的停留时间超过target值时，就将该数据包丢弃。

CoDel算法运行需要两个参数：

target：理想情况下，数据包的最长排队延时

interval：在触发CoDel开始丢包之前，持续出队的数据包排队时间超过target的最长忍耐时间

也就是说，CoDel算法没有严格限制数据包在队列里的时间，即排队延时不超过target，而是给予了一个interval的窗口。在窗口期间如果数据包的的延迟较高，仍然可以发送出去，但是会使CoDel处于警戒状态，该状态期间如果发送端察觉RTT升高，主动降低发送速率，那么后续进入队列的数据包数量减少，出队时的排队延时降低到target之下，此时CoDel解除警戒，过程中没有丢包。但是如果在interval时间耗尽之前仍没有数据包满足排队延时小于target，那么接下来就丢弃所有排队延时超过target的包（并且丢包速度愈来愈快），直到再出现小于target的数据包出队时解除警戒，重新开始发包。



图九 CoDel运行示意图

观察实验结果图，起初h1到r1，r1到h2之间的带宽都为100Mbps，还没有出现丢包的情况，故三种算法都维持在相似的较好水平。

在第100个包时，h2和r1间数据通路带宽降低到10Mbps，r1结点的队列突发拥塞。

对于taildrop算法（蓝色），短时间内大量的拥塞数据包因无法入栈而直接被丢弃，导致RTT急剧升高，随后h1结点察觉丢包，开始降低发送速率，RTT逐步下降，体现在曲线上为在100packet附近的突发上升以及后续的逐渐下降后平稳。

对于RED算法（红色），突发的拥塞数据包迅速填满Buffer，但是由于Qlen超过了MAX thresh，以概率1开始丢包，h1很快就检测到了丢包，迅速降低速度，体现在曲线上为100packet附近出现小幅迅速上升后又迅速回落，证明了RED主动拥塞控制的有效性和前瞻性。

对于CoDel算法（绿色），一直稳定在较好水平。是因为出现从出现拥堵的第一个包开始，CoDel就能检测的到，然后100ms后开始进行越来越激烈的丢包，也就是说h1结点最早在拥堵出现的100ms后就能发现问题，然后降低发包速率，综合来看CoDel算法比RED算法可以更早的让发送端意识到拥堵的问题，并且CoDel可以更快的清空信道，有更快的收敛速度。

（三）附录：工具记录

一、更改默认启动内核

1. 将两个安装包拷贝到虚拟机任意位置

2. sudo dpkg -i 00-linux-image-4.9.48-networking\_4.9.48-1\_amd64.deb

sudo dpkg -i 00-linux-headers-4.9.48-networking\_4.9.48-1\_amd64.deb

3. sudo nano /etc/default/grub，使用nano编辑器打开grub文件

4．修改GRUB\_DEFAULT=0语句，0 修改为 你想要启动的内核。如果知道你要修改的核的名字或代号，直接修改即可，随后跳到步骤10

如果不知道，跳转到步骤5。

5. 将GRUB\_TIMEOUT\_STYLE=hidden语句用#注释掉。（对于不同Ubuntu版本该语句的写法不同，Ubuntu 16.04版本中是GRUB\_HIDDEN\_TIMEOUT=0 ，类似的注释掉该行）

6. 保存grub缓冲区，执行sudo update-grub

7. 执行sudo reboot，然后等待重启

8. 重启过程中会看到启动时进入grub界面。我的界面如下：

Ubuntu

Ubuntu高级选项

memtest

…(记不清了)

grub菜单界面索引从0开始，所以默认启动0即启动Ubuntu主系统内核。Ubuntu高级选项对应的索引为1。选中该项，进入如下菜单：

Ubuntu，Linux 4.15.0-45-generic

Ubuntu，with Linux 4.15.0-45-generic （upstart）

Ubuntu，with Linux 4.15.0-45-generic （recovery mode）

Ubuntu，Linux 4.9.48-networking

Ubuntu，with Linux 4.9.48-networking （upstart）

Ubuntu，with Linux 4.9.48-networking （recovery mode）

我需要加载的是Ubuntu，Linux 4.9.48-networking，索引为3，选中它然后等待系统启动。

9. sudo nano /etc/default/grub，使用nano编辑器打开grub文件，修改GRUB\_DEFAULT=0语句为GRUB\_DEFAULT=”1> 3”。注意在>后面有一个空格，故使用双引号”。如果想取消开机加载grub选项，将之前加的注释号删去即可。

10. 保存grub缓冲区，执行sudo update-grub

11. 执行sudo reboot，然后等待重启

12. 重新启动后，在终端输入uname -r，查看版本号。如果执行正确的话你会看到Linux 4.9.48-networking。

二、更改文件夹所有者

在执行完python脚本后，由于以sudo模式启动，生成的文件夹所有者为root，其他用户没有访问修改的权力。

用到语句：chown [选项] [更改目标所有者][:[更改目标组]] 文件名

或：chown [选项] –reference=参考文件 文件名

选项：-c 如果文件权限确实被更改，就显示更改信息

-f 忽略大部分错误信息（除用法错误外）

-v 显示详细的信息（包括符号链接）

-h 更改符号链接（只对该链接做变更，而不变更链接指向的文件的所有权），但如果未加-h执行时遇到了符号链接，则变更链接指向的文件的所有权而不改变链接的所有权。

-R 递归的更改其下子文件的属性。在指定了-R后还可以继续指定-H，-L，-P：

H：如果命令行参数是指向目录的符号链接，就遍历之

L：遍历遇到的所有符号链接

P：（default）不会遍历任何符号链接

例：chown root /qlen 将 /qlen的属主更改为“root”

chown root：staff /u 和上面类似，但同时也将其属组更改为“staff”

chown -hR root /qlen 将 /qlen 及子目录下所有文件的属主更改为“root”

补充更改文件权限：chmod [选项] [<权限范围>+/-/=<权限设置>]

选项：-c 如果文件权限确实被更改，就显示更改信息

-f 忽略大部分错误信息（除用法错误外）

-v 显示详细的信息（包括符号链接）

-R 递归的更改其下子文件的属性。

权限范围：u User，即文件所有者

g Group，文件所属群组

o Other，其他用户

a All，所有用户（用处：设置u、g、o具有相同权限）

+ 增加权限

- 减少权限

= 重新赋予权限

权限设置：r 读取权限，8进制代码为4

w 写入权限，8进制代码为2

x 执行权限，8进制代码为1

- 没有权限，8进制代码为0

例如：chmod u=+r+w-x qlen.txt 表示使文件所有者者对qlen.txt拥有读写权限

等价于chmod u=rw qlen.txt

再例：chmod u=rw，g=rw，o=r qlen.txt表示属主和属组有读写权限，其他用户只读

此时用ll命令应该会看到 drw-rw-r-- 或 -rw-rw-r--

另外，也可以用chmod abc 文件名的方式同时对三类用户进行修改。abc为3位8进制数字

- 0 无权限

x 1 可执行

w 2 只写

wx 3 可写可执行

r 4 只读

rx 5 可读可执行

rw 6 读写

rwx 7 全权限

例如：chmod 777 qlen.txt即为全部用户全权限