数据包队列管理实验报告

李昊宸

2017K8009929044

（一）BufferBloat 复现实验

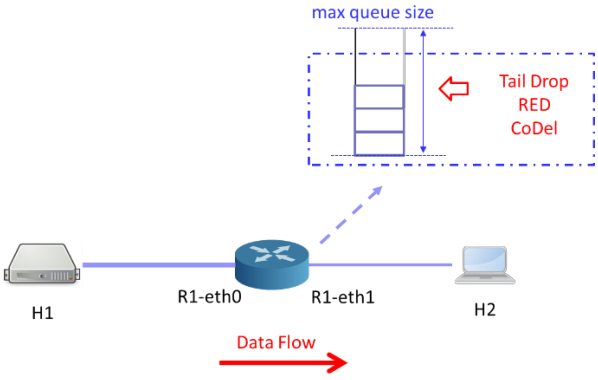
一、实验内容

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
```

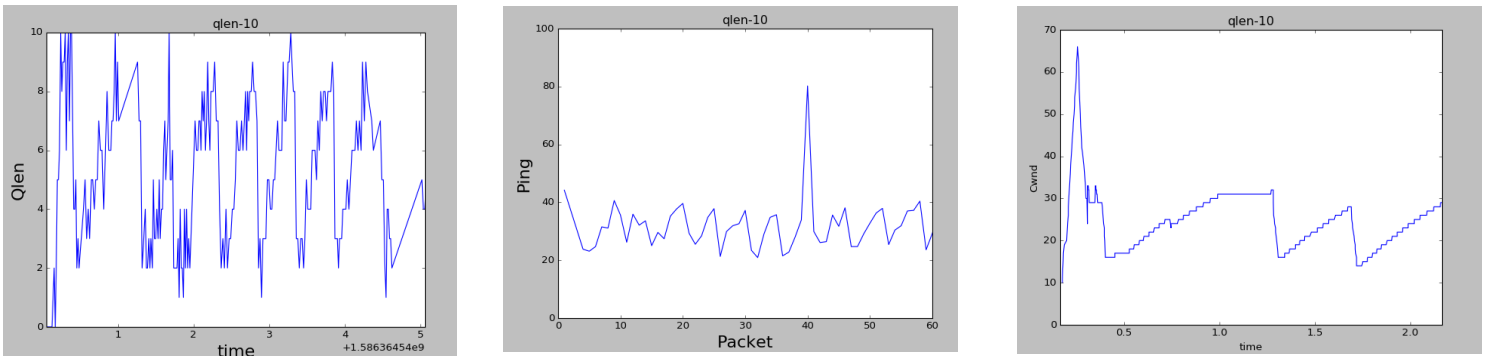
-q 后的传参表示设置最大队列大小

3. 绘制折线图

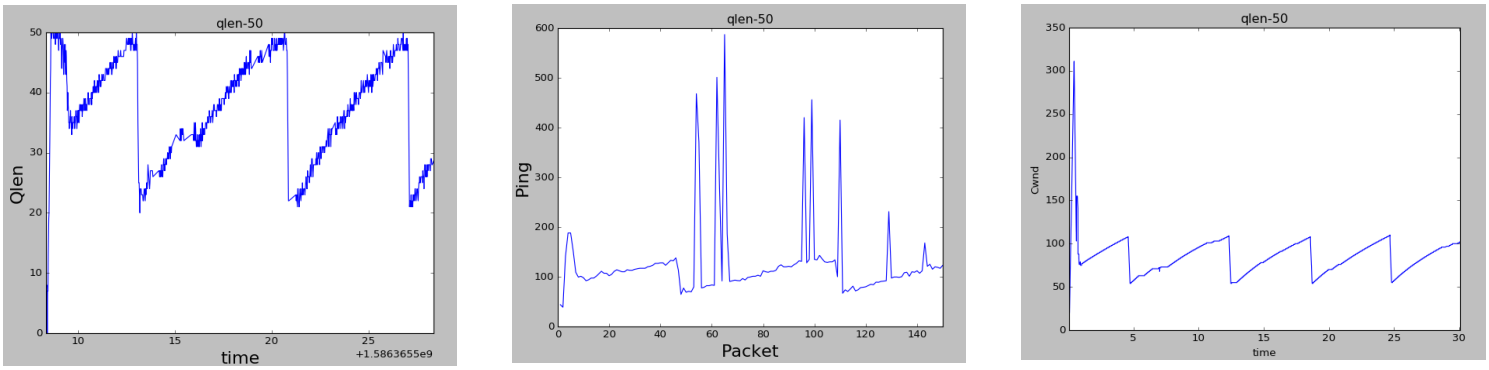
用 python 脚本处理得到的数据，分别绘制三个不同队列长度下的 Cwnd-time，Qlen-time 和 Ping-packet 折线图。

三、实验结果及分析

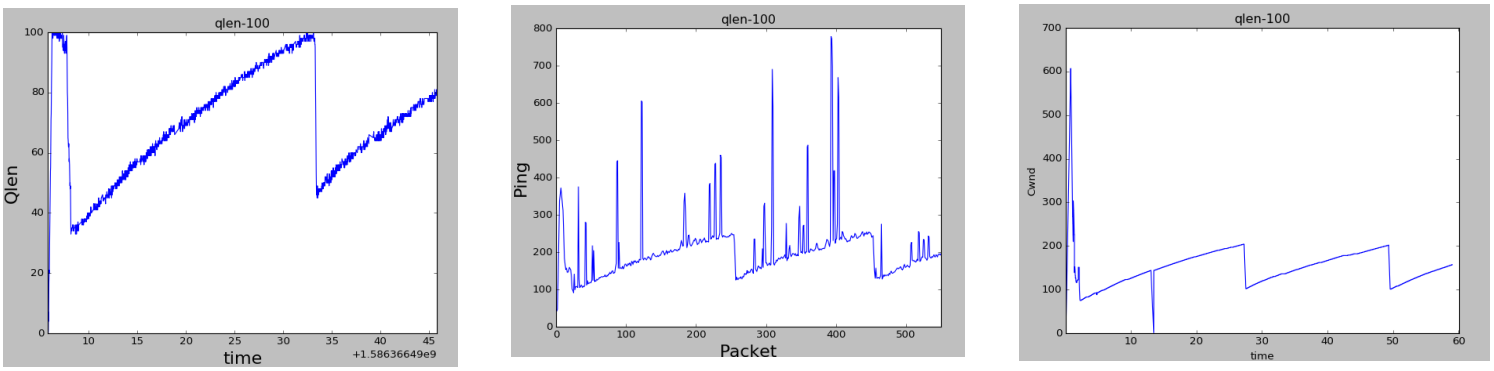
1. 实验结果



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

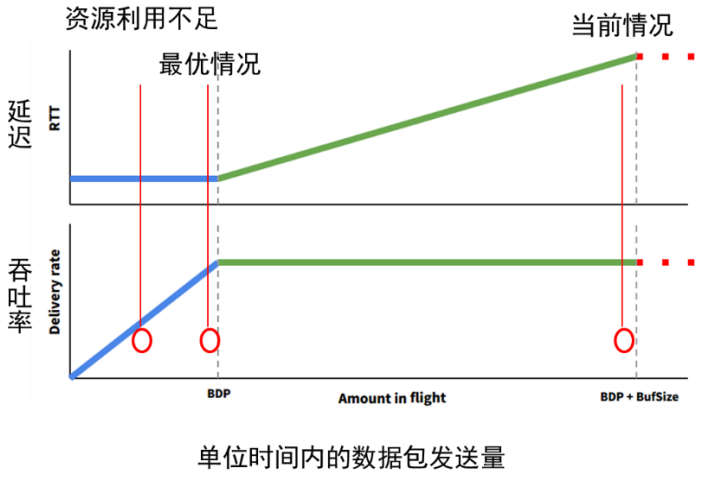


图四 最大队列为 100 时的 Qlen-time（左），Ping-packet（中） 和 Cwnd-time（右）折线图

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：Branch: master ▾ linux-intel-4.9 / net / ipv4 / tcp_probe.c

内容：

```
186  static int tcpprobe_sprint(char *tbuf, int n)
187  {
188          const struct tcp_log *p
189                  = tcp_probe.log + tcp_probe.tail;
190          struct timespec64 ts
191                  = ktime_to_timespec64(ktime_sub(p->tstamp, tcp_probe.start));
192
193          return scnprintf(tbuf, n,
194                           "%lu.%09lu %pISpc %pISpc %d %#x %#x %u %u %u %u %u\n",
195                           (unsigned long)ts.tv_sec,
196                           (unsigned long)ts.tv_nsec,
197                           &p->src, &p->dst, p->length, p->snd_nxt, p->snd_una,
198                           p->snd_cwnd, p->ssthresh, p->snd_wnd, p->srtt, p->rcv_wnd);
199  }
```

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
```

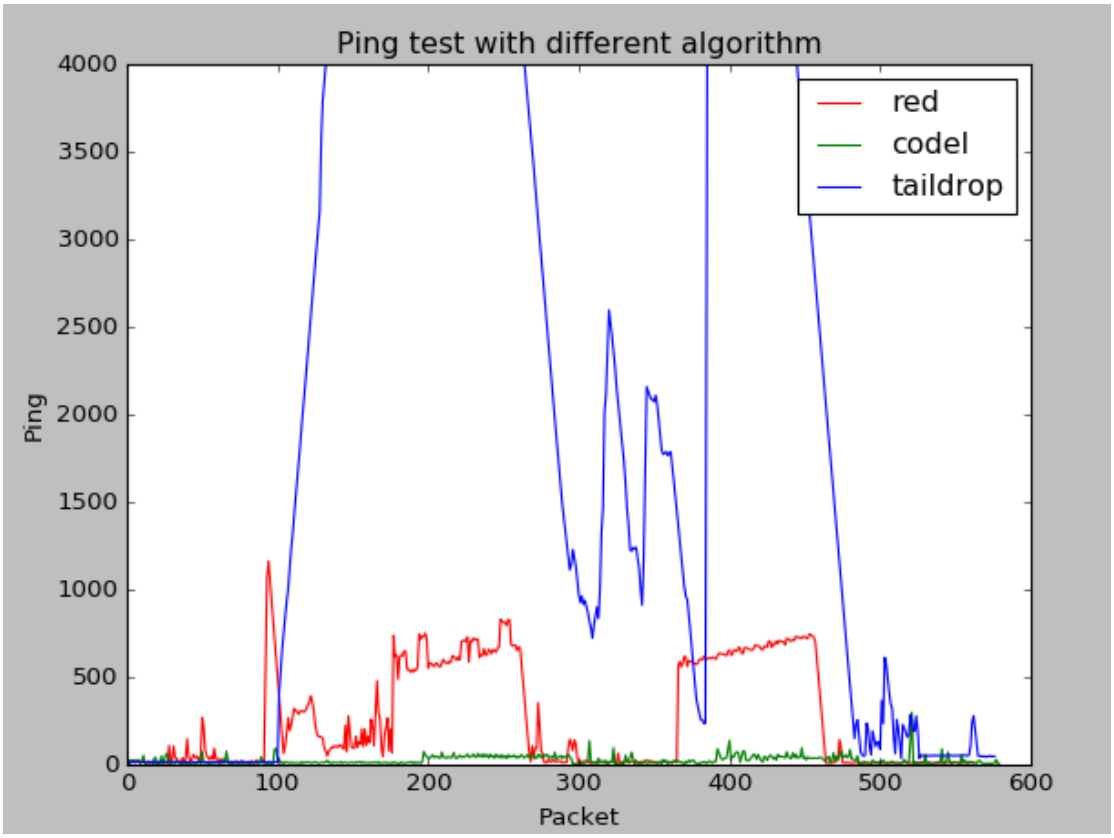
2. 启动脚本

```
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

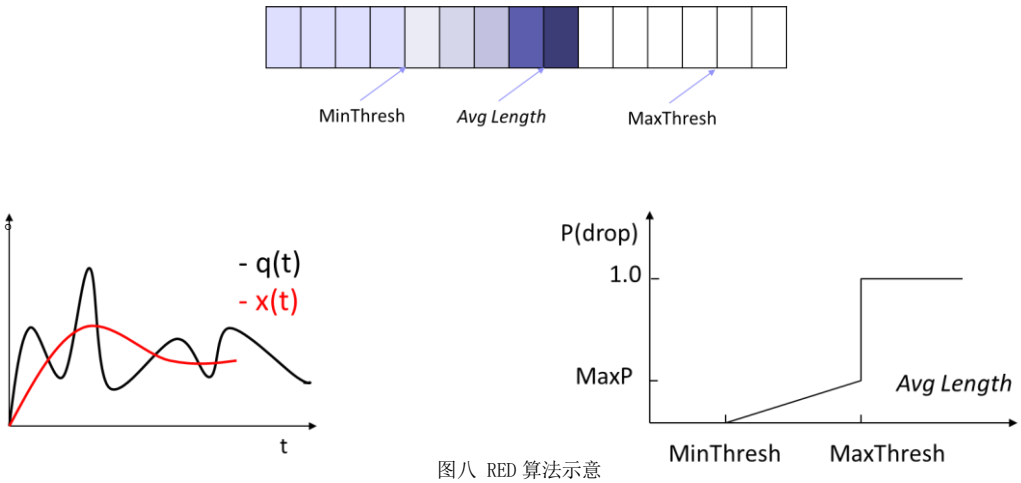
2. 实验分析

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

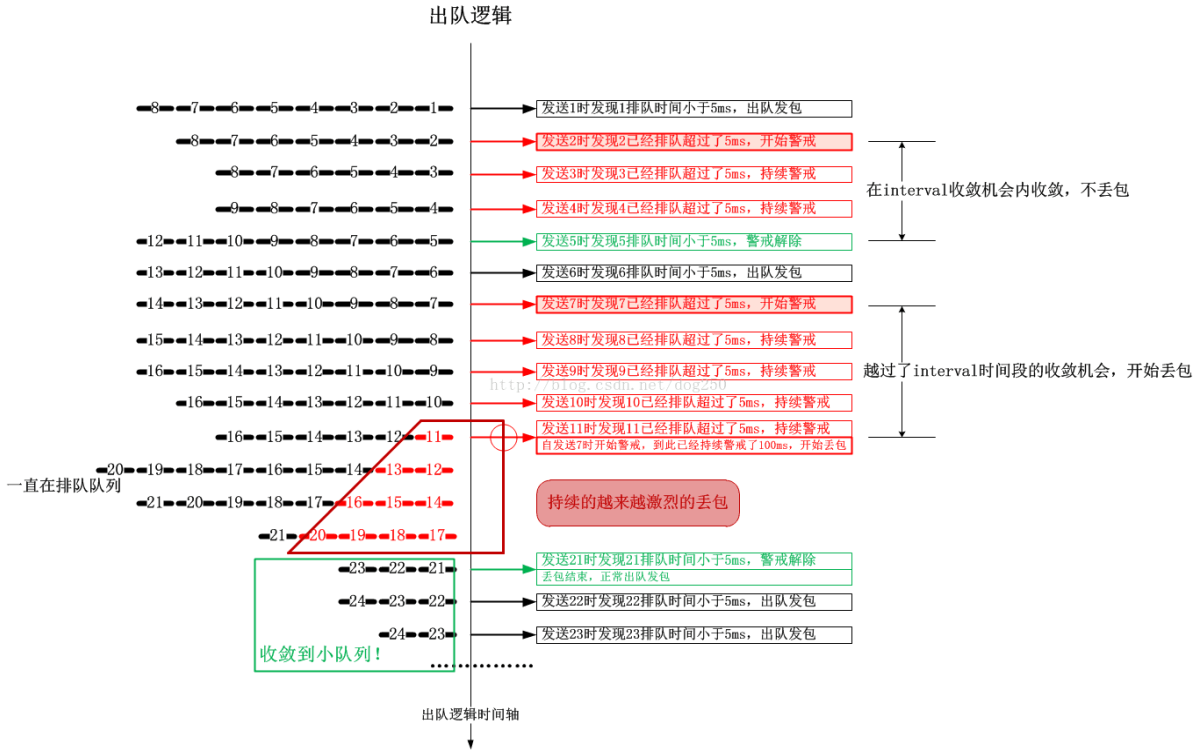
首先使用平滑函数计算平均队列长度 x(t):x(t) = (1-Wq)*x(t-T) + Wq*q(t)。
随后根据平均队列长度 x(t)的大小和两个阈值之间的关系进行概率丢包。
操作的对象均为下一个新到达的分组。

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