

# 实验四

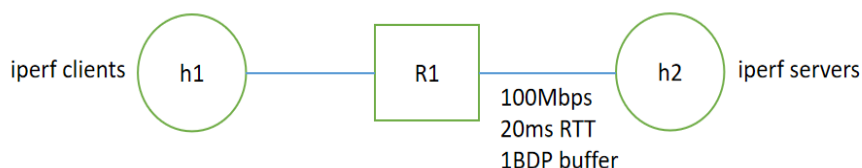
## 实验目的

设计并实现一种基于丢包和延迟的TCP拥塞控制机制

- 能够在Linux 4.15+内核环境下编译成模块，并可以加载到内核中
- 相比于Cubic/BBR算法，所设计的拥塞控制机制具有更好的收敛速度和公平性

## 实验说明

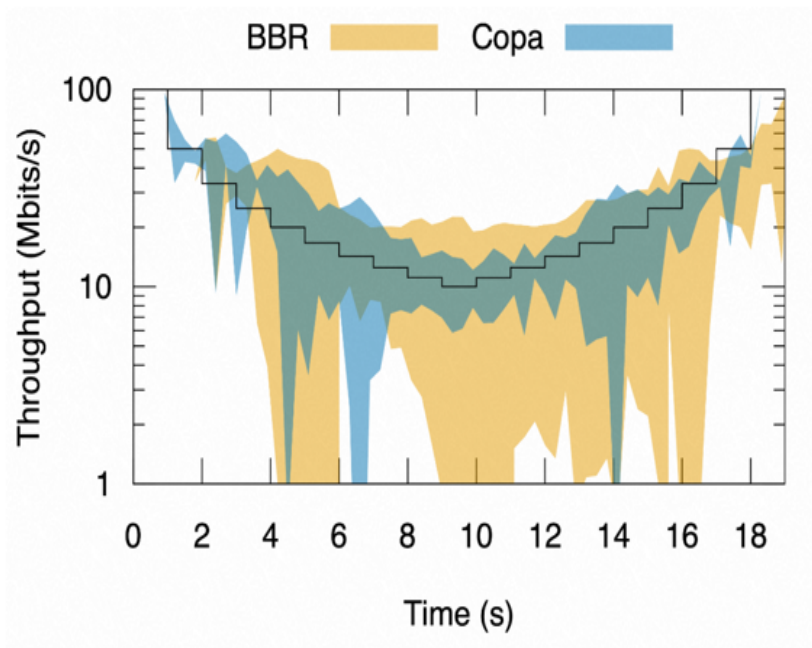
### 实验内容



- 实验拓扑：h1-R1-h2 直连，在 R1-h2 路径上设置速率限制
- 公平性和收敛性测试过程：在 h1 上每隔时间 t 启动 iperf 启动一条 TCP 流，直到有 n 条流同时发送。之后每隔时间 t 结束一条流，直到所有流停止发送
- 测试期间使用 tcpdump 抓包，供后续公平性分析使用

## 公平性指标

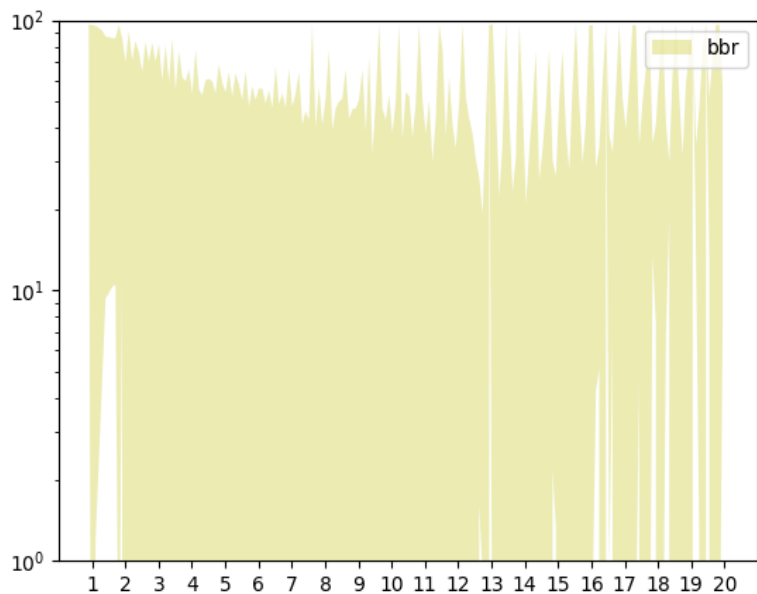
在“06-网络传输”的PPT上给出了一个公平性指标  $\max(\sum_i X_i)^2 / (\sum_i X_i^2)$ ，在假设网络资源被充分利用的情况下（即  $\sum_i X_i = c$ ，且  $c$  为定值），优化公平性等价于  $\min \sum_i X_i^2$ ，也等价于  $\min \frac{1}{n} \sum_i (X_i - \frac{c}{n})^2$ 。而这个式子的物理含义就是在某个时间刻，最小化所有流的吞吐量的方差。注意，我们假设了网络资源被充分利用，因此所有流的吞吐量的平均值也应该越贴近理论值越好。



表现在图中就是某个算法代表的区域的中心越贴近黑色线，且上下间距越薄越好。途中的蓝色区域代表的copa算法就比黄色区域代表的bbr算法要好。

## 对Cubic协议的改进

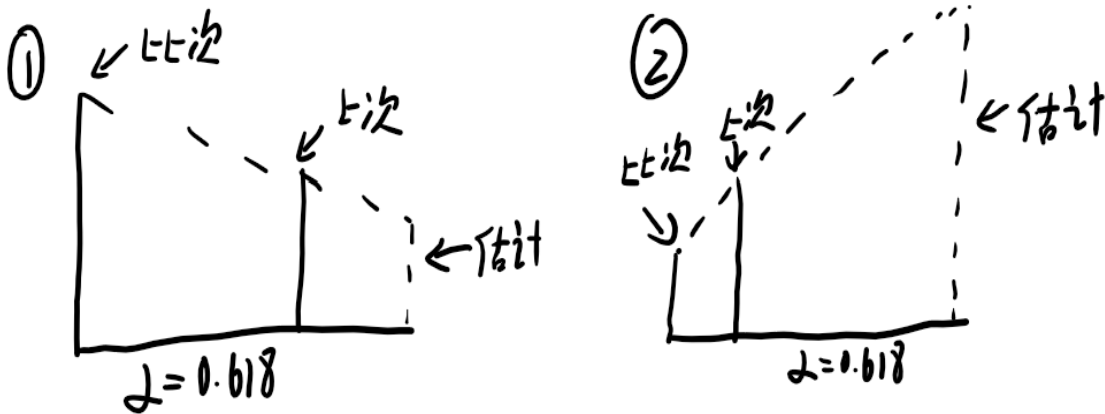
由于bbr算法存在“用户增加RTT获得更大吞吐率”的问题，且在实验中发现其公平性非常差（如下图）。同时，我们也不打算将延迟作为拥塞信号（考虑到bbr算法类似的公平性问题），因此我们考虑改进Cubic算法。



我们观察吞吐量的理想值（黑色线），发现有一下两个特质：

1. 可以分为两个近似线性的阶段：先降低后升高。
2. 降低的速率逐渐变慢，升高的速率逐渐变大。

我们基于这两点来改进Cubic。在Cubic中， $W_{max}$ 起到丢包后避免 congestion 的作用，用的是上一次丢包时 congestion window 的大小。但实际上，我们可以用之前几次丢包时 congestion window 的大小来估计  $W_{max}$  的取值。对应于特质的第一点，我们考虑基于线性的模型：用上上次和上次的丢包时 congestion window 估计  $W_{max}$ 。对应于特质的第二点，我们分两种情况考虑：上上次大于上次的丢包时 congestion window，或者上上次小于上次的丢包时 congestion window。系数就取黄金比例 0.618，估计的示意图如下：



对应的代码如下：

```
static u32 bictcp_recalc_ssthresh(struct sock *sk)
{
    const struct tcp_sock *tp = tcp_sk(sk);
    struct bictcp *ca = inet_csk_ca(sk);

    ca->epoch_start = 0; /* end of epoch */
    if (tp->snd_cwnd < ca->last_max_cwnd) { // 上上次小于上次的丢包时 congestion window
        ca->last_max_cwnd = (1618 >> 3) * tp->snd_cwnd - (618 >> 3) * ca->last_max_cwnd;
    }
    else { // 上上次大于等于上次的丢包时 congestion window
        ca->last_max_cwnd = (2618 >> 3) * tp->snd_cwnd - (1618 >> 3) * ca->last_max_cwnd;
    }
    if (ca->last_max_cwnd <= 0) {
        ca->last_max_cwnd = tp->snd_cwnd; }

    return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

## 将协议加载进内核

这一步我们完成在Linux 5.4.0 内核环境下编译成模块，且加载到内核中。这一部分查阅了大量资料和链接，见参考。

我们首先将基于cubic的代码 tcp\_chg.c 文件放到服务器中，接着用如下文件编译成模块 tcp\_chg.ko：

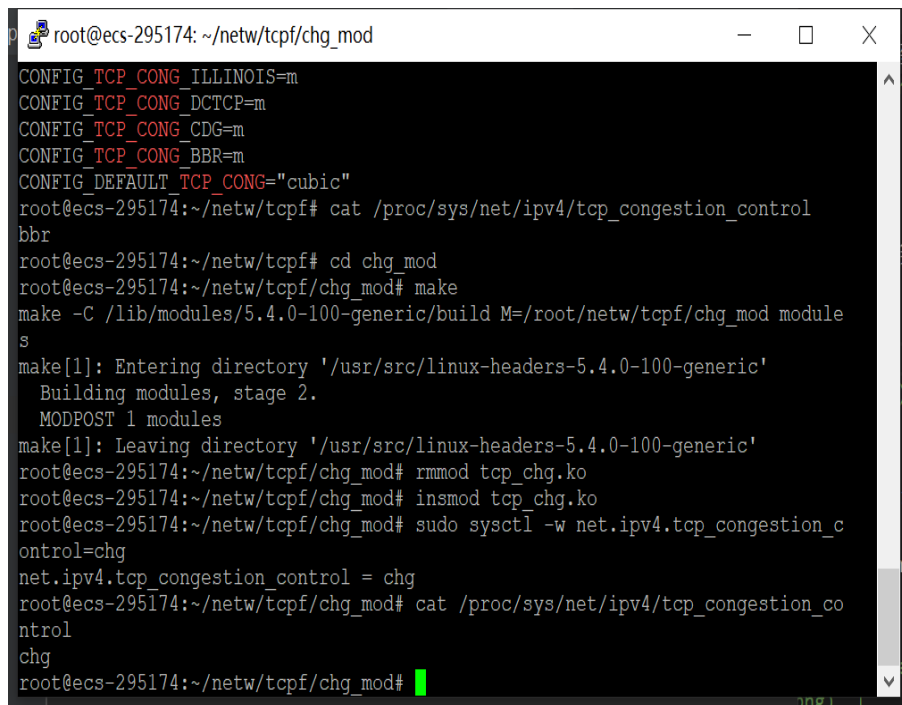
```
CONFIG_MODULE_SIG=n
obj-m += tcp_chg.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

然后将 tcp\_chg.ko 模块加载到内核中（sudo insmod tcp\_chg.ko），接着设置当前的拥塞控制协议，然后检查当前的拥塞控制协议。

完整命令如下：

```
cd netw/tcpf/chg_mod
make
sudo rmmod tcp_chg.ko
sudo insmod tcp_chg.ko
sudo sysctl -w net.ipv4.tcp_congestion_control=chg
cat /proc/sys/net/ipv4/tcp_congestion_control
```

正常的话，屏幕输出如下：可见此时的拥塞控制协议为 chg（我们的协议）。

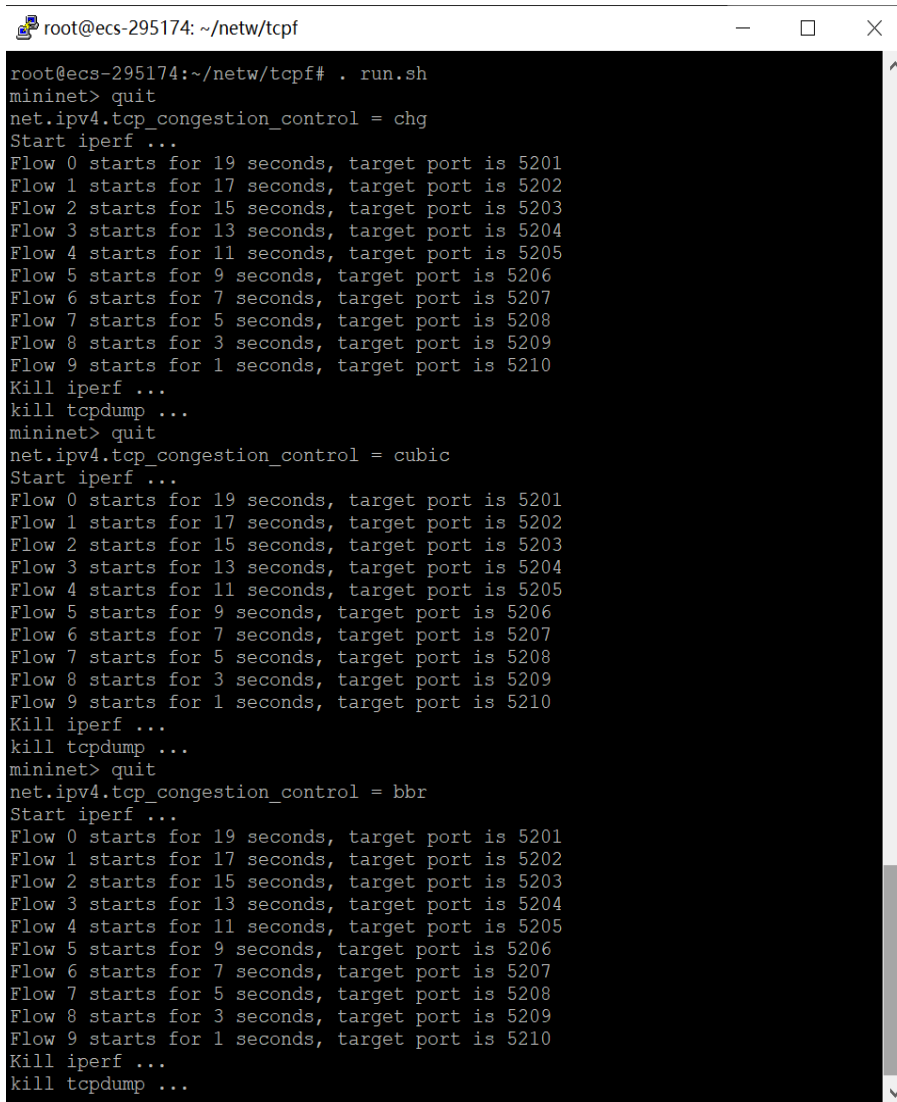


```
root@ecs-295174: ~/netw/tcpf/chg_mod
CONFIG_TCP_CONG_ILLINOIS=m
CONFIG_TCP_CONG_DCTCP=m
CONFIG_TCP_CONG_CDG=m
CONFIG_TCP_CONG_BBR=m
CONFIG_DEFAULT_TCP_CONG="cubic"
root@ecs-295174:~/netw/tcpf# cat /proc/sys/net/ipv4/tcp_congestion_control
bbr
root@ecs-295174:~/netw/tcpf# cd chg_mod
root@ecs-295174:~/netw/tcpf/chg_mod# make
make -C /lib/modules/5.4.0-100-generic/build M=/root/netw/tcpf/chg_mod module
s
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-100-generic'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-100-generic'
root@ecs-295174:~/netw/tcpf/chg_mod# rmmod tcp_chg.ko
root@ecs-295174:~/netw/tcpf/chg_mod# insmod tcp_chg.ko
root@ecs-295174:~/netw/tcpf/chg_mod# sudo sysctl -w net.ipv4.tcp_congestion_c
ontrol=chg
net.ipv4.tcp_congestion_control = chg
root@ecs-295174:~/netw/tcpf/chg_mod# cat /proc/sys/net/ipv4/tcp_congestion_co
ntrol
chg
root@ecs-295174:~/netw/tcpf/chg_mod#
```

接下来就是运行公平性实验了，命令如下：

```
python3 topo.py -c cubic
python3 topo.py -c chg
python3 topo.py -c bbr
python3 pcap_analyse_tool.py
```

屏幕输出如下，可见三种协议都正常运行了：



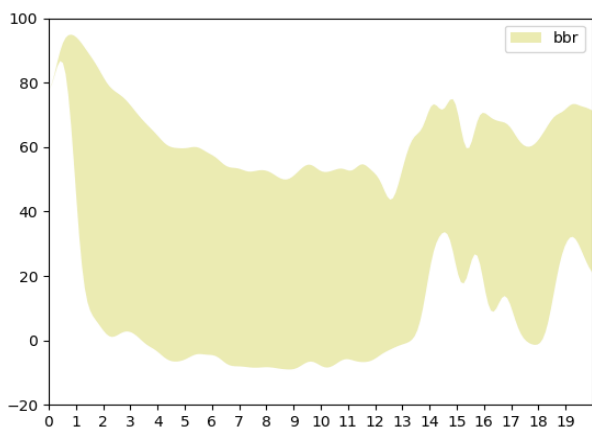
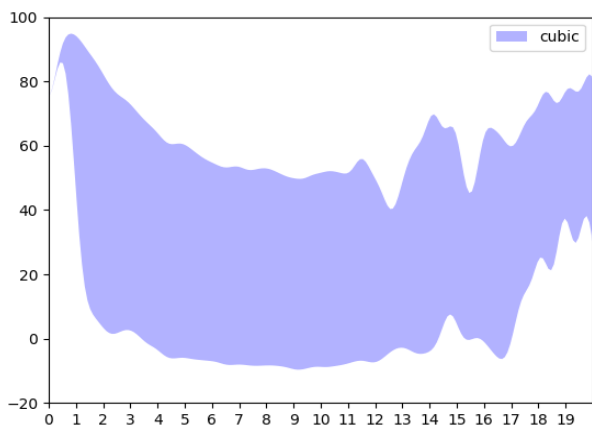
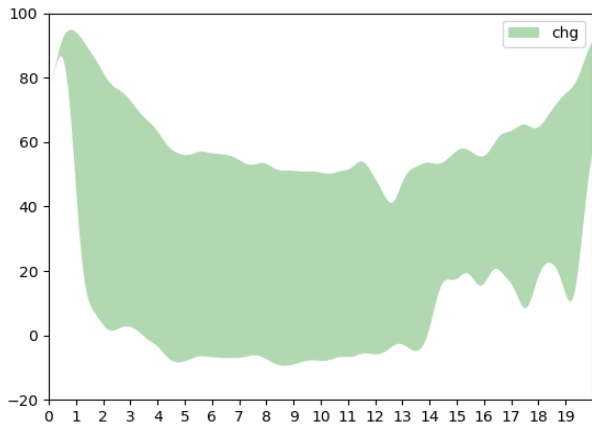
```
root@ecs-295174: ~/netw/tcpf
root@ecs-295174:~/netw/tcpf# . run.sh
mininet> quit
net.ipv4.tcp_congestion_control = chg
Start iperf ...
Flow 0 starts for 19 seconds, target port is 5201
Flow 1 starts for 17 seconds, target port is 5202
Flow 2 starts for 15 seconds, target port is 5203
Flow 3 starts for 13 seconds, target port is 5204
Flow 4 starts for 11 seconds, target port is 5205
Flow 5 starts for 9 seconds, target port is 5206
Flow 6 starts for 7 seconds, target port is 5207
Flow 7 starts for 5 seconds, target port is 5208
Flow 8 starts for 3 seconds, target port is 5209
Flow 9 starts for 1 seconds, target port is 5210
Kill iperf ...
kill tcpdump ...
mininet> quit
net.ipv4.tcp_congestion_control = cubic
Start iperf ...
Flow 0 starts for 19 seconds, target port is 5201
Flow 1 starts for 17 seconds, target port is 5202
Flow 2 starts for 15 seconds, target port is 5203
Flow 3 starts for 13 seconds, target port is 5204
Flow 4 starts for 11 seconds, target port is 5205
Flow 5 starts for 9 seconds, target port is 5206
Flow 6 starts for 7 seconds, target port is 5207
Flow 7 starts for 5 seconds, target port is 5208
Flow 8 starts for 3 seconds, target port is 5209
Flow 9 starts for 1 seconds, target port is 5210
Kill iperf ...
kill tcpdump ...
mininet> quit
net.ipv4.tcp_congestion_control = bbr
Start iperf ...
Flow 0 starts for 19 seconds, target port is 5201
Flow 1 starts for 17 seconds, target port is 5202
Flow 2 starts for 15 seconds, target port is 5203
Flow 3 starts for 13 seconds, target port is 5204
Flow 4 starts for 11 seconds, target port is 5205
Flow 5 starts for 9 seconds, target port is 5206
Flow 6 starts for 7 seconds, target port is 5207
Flow 7 starts for 5 seconds, target port is 5208
Flow 8 starts for 3 seconds, target port is 5209
Flow 9 starts for 1 seconds, target port is 5210
Kill iperf ...
kill tcpdump ...
```

为了更好地看清楚结果，我们对原图像进行一个平滑（pcap\_analyse\_tool.py 文件）：

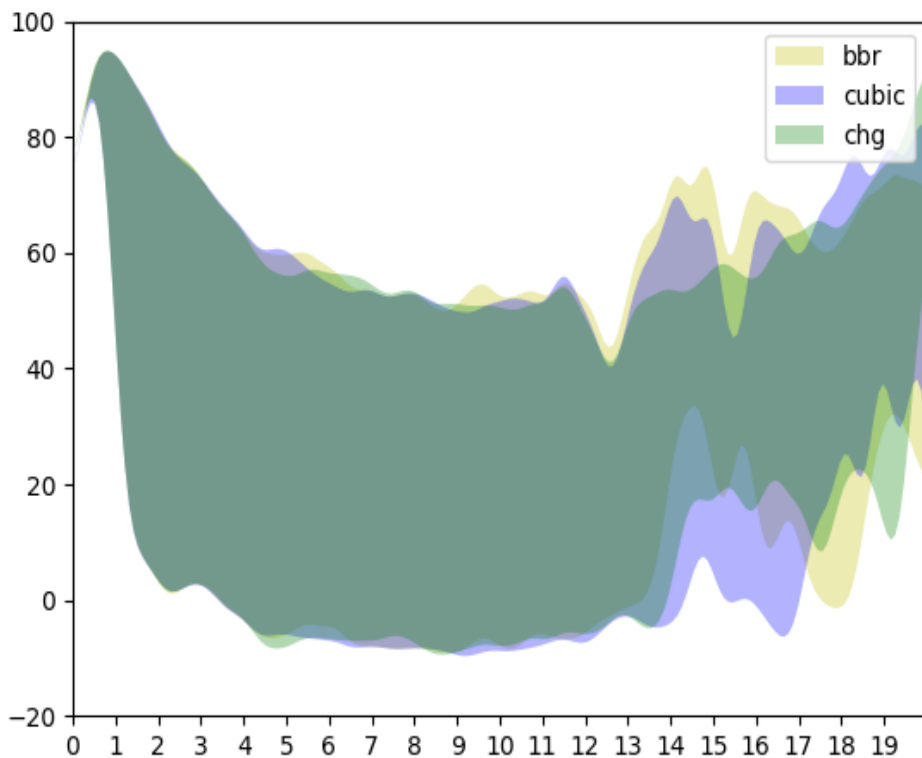
```
# ax.fill_between(timeline,mean_tpts-std_tpts,mean_tpts+std_tpts,alpha=alpha,
# color=color,linewidth=0.0,label=str(alg))
ax.fill_between(timeline, gaussian_filter1d(mean_tpts - std_tpts, sigma=3),
                gaussian_filter1d(mean_tpts + std_tpts, sigma=3), alpha=alpha,
                color=color, linewidth=0.0,label=str(alg))
```

## 实验结果和分析

三种协议表现如下：



对比如下:



bbr、cubic和我们的算法chg在时间前半段分布大致差不多，但是在后半段chg要比bbr、cubic上下区间薄。且三者的中心位置相近。

在前面我们已经提到，公平性由方差体现，在图中表现为上下间距越薄越好。因此我们可以得出结论：在实验给定的情境下，chg在**公平性**上比bbr、cubic好。

此外，也可以看出chg比bbr、cubic曲线更为光滑，这意味着chg在**收敛速度**上优于bbr、cubic。

**效率**由图像的上下分布的中间线和理想值之间的距离衡量，三者大体相同。

## 参考

- [Linux Kernel Source](#)
- [net/ipv4/tcp\\_bbr.c](#)
- [net/ipv4/tcp\\_cubic.c](#)
- [调试Linux Kernel任意模块](#)
- [The Linux Kernel Module Programming Guide](#)
- [How to compile Linux kernel modules](#)
- [关于Tcpdump抓包总结](#)
- [module verification failed: signature and/or required key missing](#)
- [TCPTuner](#)
- [启用新的tcp拥塞控制算法-bbr](#)