

计算机网络综合实验TCP拥塞控制算法改进实验 (NS-3)

姓名：姚丁钰

班级：智能2103班

学号：202107030125

一、实验目的

二、实验内容

2.1传输协议改进

2.2实验背景知识：

2.2.1慢启动和拥塞避免

2.2.2快速重传和快速恢复

2.2.3改进思路

2.2.4安装环境

2.2.5代码部署

2.2.6NS3内部实验

2.2.7NS3 TO DOCKER实验

2.2.8实验分析

三、实验总结

一些问题

一、实验目的

学习改进TCP的拥塞控制算法，通过改进其快速恢复算法部分，以提高传输的性能，并进行测试。

二、实验内容

2.1传输协议改进

实验环境：Ubuntu20.04+NS3.33+Docker

在NS3中实现对TCP NewReno拥塞控制算法的改进，改进后的算法称为TCP NewRenoPlus。测试分别为NS3内部测试和NS3 TO DOCKER测试。

2.2实验背景知识：

简要介绍 TCP NewReno 的主要算法，慢启动、拥塞避免、快速重传和快速恢复。

2.2.1慢启动和拥塞避免

在 TCP NewReno 中，当 TCP 连接开始时，慢启动算法将拥塞窗口初始化为一个报文段。然后，发送方根据窗口大小发送数据包。对于从接收方返回的每个确认数据包，拥塞窗口增加一个报文段（Segment）。慢启动会一直维持，直到拥塞窗口达到慢启动阈值（slow start threshold, ssthresh）。此时，NewReno 进入拥塞避免阶段，降低拥塞窗口的增长速率。在拥塞避免阶段，只要没有检测到网络拥塞的出现，拥塞窗口每个往返时延（Round-Trip Time, RTT）都会线性增加一个报文段。慢启动算法和拥塞避免算法的实现方式是使拥塞窗口持续增大，直到发生拥塞。此时，发送速率应减慢以解决网络拥塞。

2.2.2快速重传和快速恢复

在拥塞避免期间，接收到重复确认或重传计时器超时都会隐式地向发送方发出网络拥塞的信号。因此，发送方必须减缓传输速率。如果拥塞是由超时引起的，则将 ssthresh 设置为当前拥塞窗口的一半，将拥塞窗口设置为一个报文段，并使发送方进入慢启动阶段。如果拥塞是由重复确认引起的，则 TCP 发送方进入快速重传模式，无需等待重传计时器到期即可重传似乎丢失的数据包。然后，发送方将 ssthresh 设置为当前拥塞窗口的一半，将新的拥塞窗口设置为新的 ssthresh 加上接收到的重复确认的数量，并进入快速恢复阶段。进入快速恢复后，对于每个后续接收到的重复 ACK(Acknowledgement)，发送方将拥塞窗口逐渐增加一个报文段。快速恢复算法的基本思想是，重复 ACK 表示接收方已经接收到一些数据段，因此可以触发新的数据段传输。发送方在其拥塞窗口允许的情况下传输新的数据段。在快速恢复期间，TCP NewReno 区分“部分”ACK（partial ACK）和“完整”ACK（full ACK）。完整 ACK 确认了在快速恢复开始时还未确认的所有数据段，而部分 ACK 只确认了其中一部分数据段。在收到部分 ACK 时，NewReno 根据部分 ACK 重新传输序列中的下一个数据段，并将拥塞窗口减少一个少于部分 ACK 确认的数据段数。这种窗口减小称为部分窗口缩小，允许发送方在快速恢复的后续 RTT 中传输新的数据段。NewReno 会一直维持在快速恢复阶段，直到在快速恢复开始时未确认的所有数据段都已被确认。在收到完整 ACK 时，发送方将拥塞窗口设置为 ssthresh，退出快速恢复阶段，并进入拥塞避免阶段。

下表为TCP NewReno的伪代码

--

```

1 Slow Start
2   Initial: cwnd = 1 ;
3 For (each packet Acked)
4   cwnd++;
5 Until (congestion event, or, cwnd > ssthresh)
6 Congestion Avoidance
7   // 慢启动结束且 cwnd > ssthresh
8   Every Ack
9   cwnd = cwnd + (1 / cwnd)
10 Until (Timeout or 3 DUPACKs (Duplicate Acknowledgement))
11 Fast Retransmit
12   // 接收到3个 DUPACKs
13   Resend lost packet;
14   Invoke Fast Recovery
15 Fast Recovery
16   // 快速重传后, 不进入慢启动
17   ssthresh = cwnd / 2;
18   wwnd = ssthresh + 3 ;
19   Each DACK received;
20   cwnd++;
21   Send new packet if allow;
22   After receiving an Ack;
23 If partial Ack;
24   Stay in fast recovery;
25   Retransmit next lost packet (one packet per RTT);
26 If Full Ack;
27   cwnd = ssthresh;
28   Exit fast recovery;
29   Invoke Congestion Avoidance Algorithm;
30
31   When Timeout
32   ssthresh = cwnd / 2;
33   cwnd = 1 ;
34   Invoke Slow Start;
35

```

2.2.3改进思路

NewReno 的问题在于, 在快速恢复阶段中, 无论网络状态如何, 都会将其拥塞窗口减半。NewReno 的另一个问题是, 当没有发生丢包, 但是数据包不守序地到达接收方, 发送方可能会收到超过三个重复确认的数据包, TCP NewReno 错误地进入快速恢复状态, 并将拥塞窗口减半。由于 TCP 的拥塞窗口控制着 TCP

发送方在任何 时间内可以通过网络发送的数据包数量，因此将拥塞窗口设置为其值的一半会使 TCP NewReno 的链路利用率低下。

对 NewReno 的改进是通过修改其快速恢复算法来避免这些问题。基本思想是根据网 络中的拥塞程度来调整 TCP 发送方的拥塞窗口，以允许更多数据包传输到目的地。拥 塞窗口的调整有三个主要目标。第一个目标是利用可用的网络资源，第二个目标是尽量 减小拥塞的可能性，第三个目标是在多个连接之间提供公平的带宽分配。通过根据网络 状态采用不同的拥塞窗口大小，可以实现这些目标。

在 TCP 中，所有传输的数据包都会从发送方回到接收方，完成一次往返。RTT 的测量是 TCP 超时和重传策略的基础。随着网络流量的变化，TCP连接期间的 RTT也会发生变化。换句话说，随着网络负载增加，RTT会增加。因此，RTT可以用来反映网络的状态。基本思想的关键在于，在给定的网络状态下，通过观察 RTT的变化，修改的算法可以确定网络中的拥塞程度。在进入快速恢复算法时，检测 RTT的变化，并根据RTT的增大程度来减小拥塞窗口。具体方法是，发送方维护一个数据结构，里面保存这每个RTT值。当 TCP 进入快速恢复算法时，使用之前保存的 RTT 值计算平均 RTT (RTT_{avg})，计算公式如下

$$RTT_{avg} = \sum_{i=1}^n RTT_i / n$$

发送方随后通过计算最新的 RTT (RTT_n) 与平均 RTT (RTT_{avg}) 之间的差异，来计算 RTT 的变化量 (ΔRTT)，公式如下

$$\Delta RTT = RTT_n - RTT_{avg}$$

最后，发送方根据 RTT 的变化来减小其发送速率。换句话说，发送方通过将当前的拥塞窗口除以平均 RTT，来计算发送速率的增长因子 (Factor)，公式如下

$$Factor = cwnd / RTT_{avg}$$

随着网络流量的增加，拥塞窗口将会减小一个平均数 (Avgnum)，该平均数是增长因子和 ΔRTT 的乘积，公式如下

$$Avg_{num} = Factor * \Delta RTT$$

因此，新的拥塞窗口 ($cwnd_n$) 由两个报文段值和当前拥塞窗口与平均值 (Avgnum) 之差的最大值决定，公式如下

$$cwnd_n = \max \{2, (cwnd - Avg_{num})\}$$

在拥塞避免阶段，当TCP发送方接收到三个重复的ACK时，进入快速重传阶段，立即重传可能丢失的数据包，而不等待重传计时器超时。然后，计算新的拥塞窗口（cwndn），将sssthresh设置为两个报文段值和cwndn之间的最大值，并将cwnd设置为sssthresh值加上接收到的重复确认的数量，并继续留着快速恢复阶段。TCP发送方对每个接收到的重复确认都将cwnd增加一个报文段值，并在允许的情况下发送新的数据段。在接收到部分确认时，重传被确认的数据段并继续留在快速恢复阶段。在接收到完整确认时，它将cwnd设置为sssthresh值，并调用拥塞避免算法。如果TCP发送方通过超时检测到数据包丢失，它将sssthresh设置为两个分段和cwndn的最大值，并将cwnd设置为一个报文段值，然后进入慢启动算法。修改后的快速恢复算法如下表所示。

```
1 Modified Fast Recovery
2   Calculate cwndn;
3   cwndn = max {2, (cwnd - Avgnum)};
4   sssthresh = max {2, cwndn};
5   sssthresh = cwnd / 2;
6   cwnd = sssthresh + 3 ;
7   Each DACK received;
8   cwnd++;
9   Send new packet if allow;
10  After receiving an Ack;
11  If partial Ack;
12    Stay in fast recovery;
13    Retransmit next lost packet (one packet per RTT);
14  If Full Ack;
15    cwnd = sssthresh;
16    Exit fast recovery;
17    Invoke Congestion Avoidance Algorithm;
18
19    when Timeout;
20    Calculate cwndn;
21      cwndn = max {2 , (cwnd - Avgnum)};
22      sssthresh = max {2 , cwndn};
23    cwnd = 1 MSS;
24    Invoke slow start;
```

2.2.4安装环境

- 安装Docker

将installdocker.sh放入Ubuntu系统，在终端执行下列shell代码

```
chmod +x installdocker.sh
```

```
sudo ./installdocker.sh
```

```
1 #!/bin/env bash
2
3 #卸载旧版本
4 sudo apt-get remove docker docker-engine docker.io containerd runc
5 #卸载旧版本
6 sudo apt-get update
7 #安装 apt 依赖包
8 sudo apt-get -y install apt-transport-https ca-certificates curl software-properties-common
9 #安装GPG证书
10 curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -
11 #验证
12 sudo apt-key fingerprint 0EBFCD88
13 #设置稳定版仓库
14 sudo add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb_release -cs) stable"
15 #安装最新版本
16 sudo apt-get install docker-ce docker-ce-cli containerd.io
17
18 #参考链接
19 #https://blog.csdn.net/u012590718/article/details/125632482
```

安装docker完成，接下来是测试docker是否安装成功。

```
docker run hello-world
```

因为是在root模式下执行的脚本(sudo),所以后面都得进root模式，这里使用docker也得前面加个sudo

```
See 'docker run --help'.
ydy@ubuntu:~/Documents/NewRenoPlus/files$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

出现以上信息为docker安装成功。

- 制作自己的docker镜像

使用我给出的Dockerfile文件，在该文件所在的目录下，启动终端，运行如下指令

```
docker build -t ns3_docker .
```

注意这个.是必要的

```
ydy@ubuntu:~/Documents/NewRenoPlus/files$ sudo docker build -t ns3_docker .
[sudo] password for ydy:
[+] Building 472.2s (8/8) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 349B                                0.0s
=> [internal] load .dockerignore                                  0.0s
=> => transferring context: 2B                                       0.0s
=> [internal] load metadata for docker.io/library/ubuntu:20.04    7.1s
=> CACHED [1/4] FROM docker.io/library/ubuntu:20.04@sha256:f2034e7195f61 0.0s
=> [2/4] RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get 457.7s
=> [3/4] RUN rm -rf /var/lib/apt/lists/*                          0.4s
=> [4/4] WORKDIR /app                                             0.1s
=> exporting to image                                             6.8s
=> => exporting layers                                             6.8s
=> => writing image sha256:dc923d8cbbd49dd7e49ad878e4f40b6184c0c4d88481f 0.0s
=> => naming to docker.io/library/ns3_docker                      0.0s
```

如果安装的时候提示network error，可以执行如下指令来重启docker和Ubuntu网络

```
sudo service network-manager restart
```

```
sudo systemctl restart docker
```

镜像制作好之后

```
docker image ls
```

可以看到镜像ns3_docker信息

```
ydy@ubuntu:~/Documents/NewRenoPlus/files$ sudo docker image ls
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
ns3_docker           latest          dc923d8cbbd4   56 seconds ago 1.25GB
hello-world          latest         d2c94e258dc    7 months ago   13.3kB
```

2.2.5代码部署

NewRenoPlus包括以下文件tcp-NewRenoPlus.h、tcp-NewRenoPlus.cc、tcp-recovery-ops.h、tcp-recovery-ops.cc、tcp-socket-base.cc、wscript。

NS3内部实验包括以下文件plottcpalgo.py、compare-tcp-algorithms.sh、my-tcp-csma.cc、my-tcp-p2p.cc。

NS3 TO DOCKER实验包括以下文件setup_tap.sh、teardown_tap.sh、csma-p2p-p2p.cc、wifi-p2p-p2p.cc、testdocker.cpp。

- 部署NewRenoPlus:

将 tcp-NewRenoPlus.h、tcp-NewRenoPlus.cc 放入 ns-allinone-3.33 /ns-3.33/src/internet/model。

如果是NS3.33, 可以直接将 tcp-recovery-ops.h、tcp-recovery-ops.cc、tcp-socket-base.cc 放入 NS-3.33/src/internet/model 覆盖替换文件, wscript放入 ns-allinone-3.33 /ns-3.33/src/internet/ 覆盖替换文件。如果不是NS3.33, 建议根据我给出的代码自己修改对应文件。

改动如下:

(1) tcp-socket-base.cc: 修改了 void TcpSocketBase::EnterCwr (uint32_t currentDelivered),

修改了 void TcpSocketBase::EnterRecovery (uint32_t currentDelivered)

(2) tcp-recovery-ops.h: 增加头文件代码 #include <deque> ,

class TcpRecoveryOps : public Object 里增加函数

```
virtual void NewRenoEnterRecovery (std::deque<RttHistory>
rttHistory, Ptr<TcpSocketState> tcb, uint32_t dupAckCount,
uint32_t unAckDataCount, uint32_t deliveredBytes) = 0;
```

class TcpClassicRecovery : public TcpRecoveryOps 里增加函数

```
virtual void NewRenoEnterRecovery (std::deque<RttHistory>
rttHistory, Ptr<TcpSocketState> tcb, uint32_t dupAckCount,
uint32_t unAckDataCount, uint32_t deliveredBytes) override;
```

(3) tcp-recover-ops.cc: 增加头文件代码 #include <algorithm>

增加函数 void TcpClassicRecovery::NewRenoEnterRecovery
(std::deque<RttHistory> rttHistory, Ptr<TcpSocketState> tcb,
uint32_t dupAckCount, uint32_t unAckDataCount, uint32_t
deliveredBytes)

(4) wscript: def build(bid)函数里, obj.source的底部加上 'model/tcp-NewRenoPlus.cc',

headers.source的底部加上 'model/tcp-NewRenoPlus.h',。注意这两个英文的“,”也是要添加的。

实验文件 `setup_tap.sh`、`teardown_tap.sh`、`my-tcp-csma.cc`、`my-tcp-p2p.cc`、`csma-p2p-p2p.cc`、`wifi-p2p-p2p.cc`、`testdocker.cpp` 放入 `ns-allinone-3.33 /ns-3.33/scratch`，剩下的 `plottcpalgo.py`、`compare-tcp-algorithms.sh` 文件放入 `ns-allinone-3.33 /ns-3.33/`。

注意这些sh文件都要执行 `chmod +x` 代码来赋予权限。

至此你完成了代码的部署，接下来要 `build ns3`。

在 `ns-allinone-3.33 /ns-3.33/` 下的终端执行以下shell代码：

```
CXXFLAGS="-wno-error" ./waf configure --enable-sudo --disable-python
```

注意等号前后没有空格

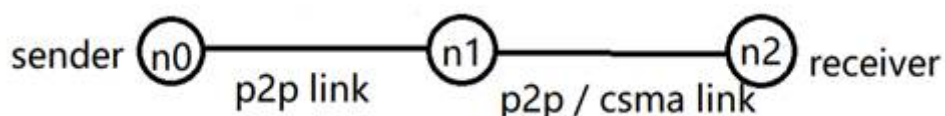
2.2.6 NS3内部实验

对比TCP NewReno 和 TCP NewRenoPlus 在 RTT、吞吐量、拥塞窗口和丢包速率的表现。

模拟时间 300 秒

丢包率 默认 1%，point to point 链路增加 10%丢包率的测试

下图为实验拓扑图。



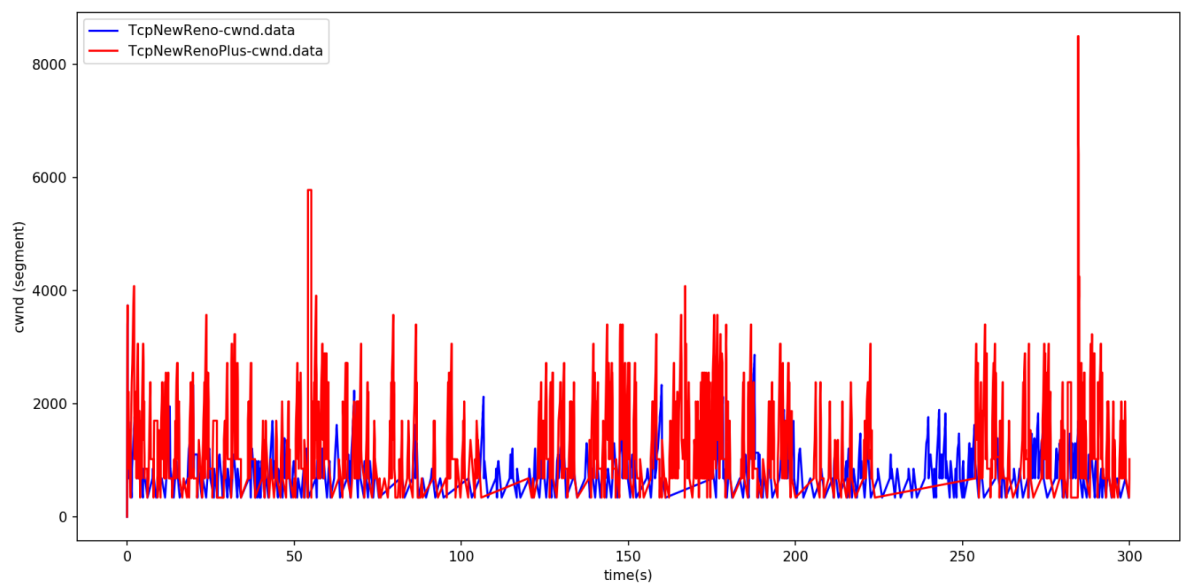
在 `ns-allinone-3.33 /ns-3.33/` 下执行 `./compare-tcp-algorithms.sh`

`compare-tcp-algorithms.sh` 内参数可自行修改，如 `my-tcp-p2p` 可改为 `my-tcp-csma`，这样你就从测试p2p改为测试csma链路。

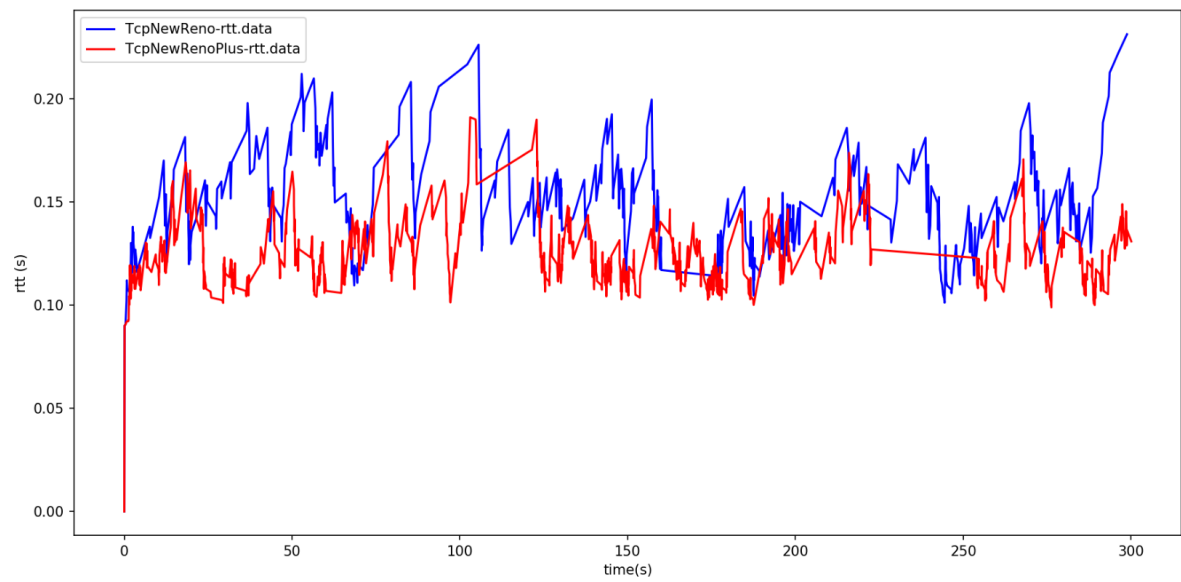
生成数据和图片在 `ns-allinone-3.33 /ns-3.33/data` 下，RTT、拥塞窗口已自动生成图片，吞吐量、丢包速率请使用wireshark打开pcap文件，随后在顶部统计->I/O图表，可以看到All Packets（吞吐量）、TCP Errors（丢包速率）。

可以看到生成了6张图片，其中有两对是相同的，所以我们只需要看4张即可。

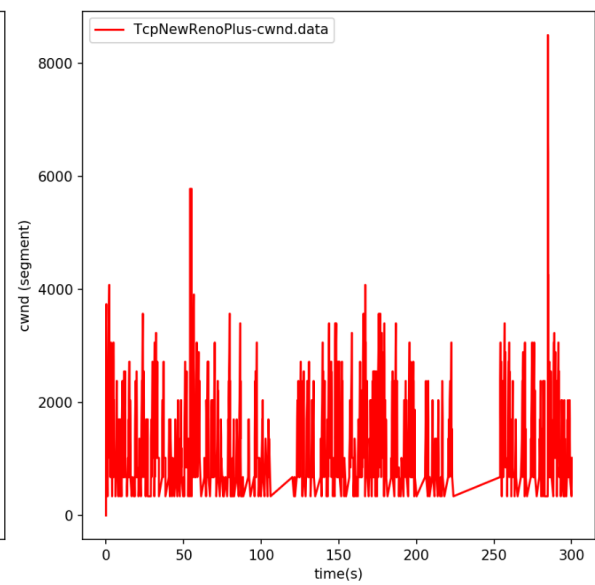
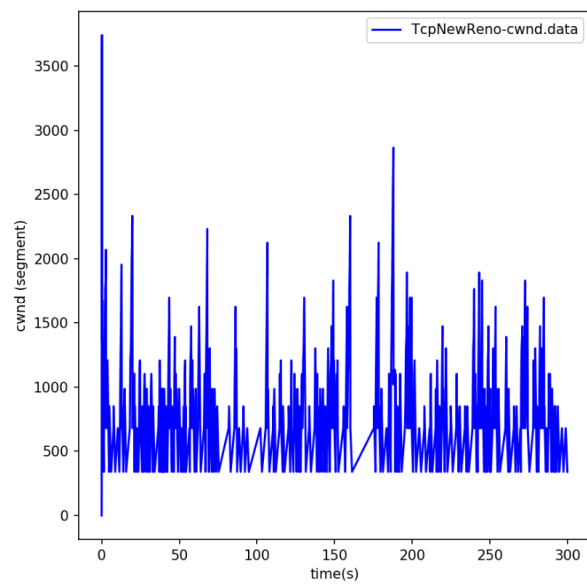
窗口大小对比：



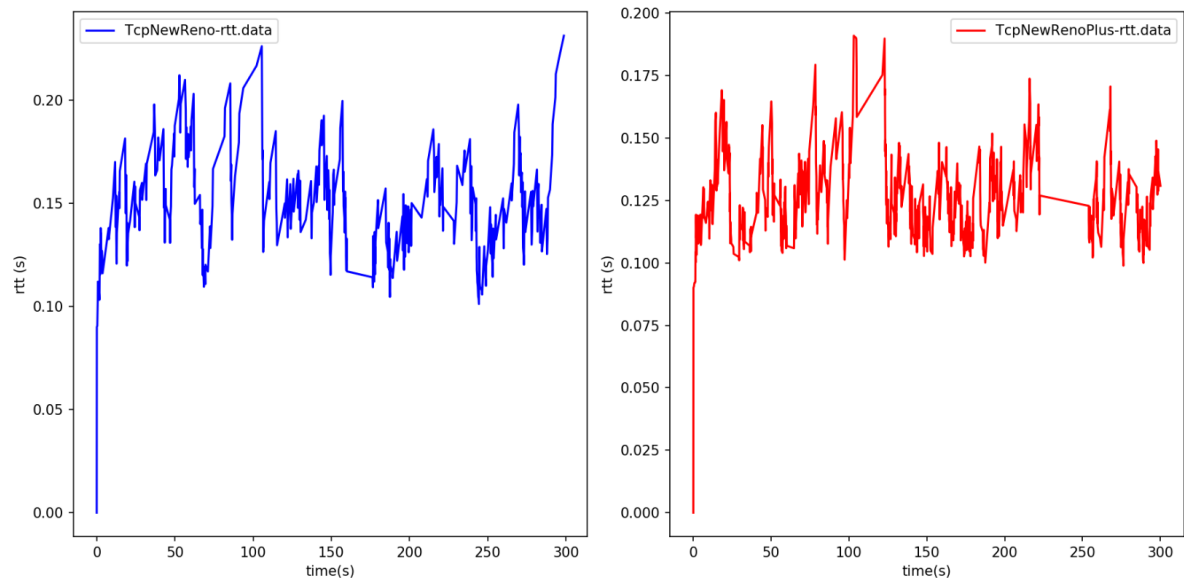
rtt时间对比:



两个算法的CWND随时间的变化

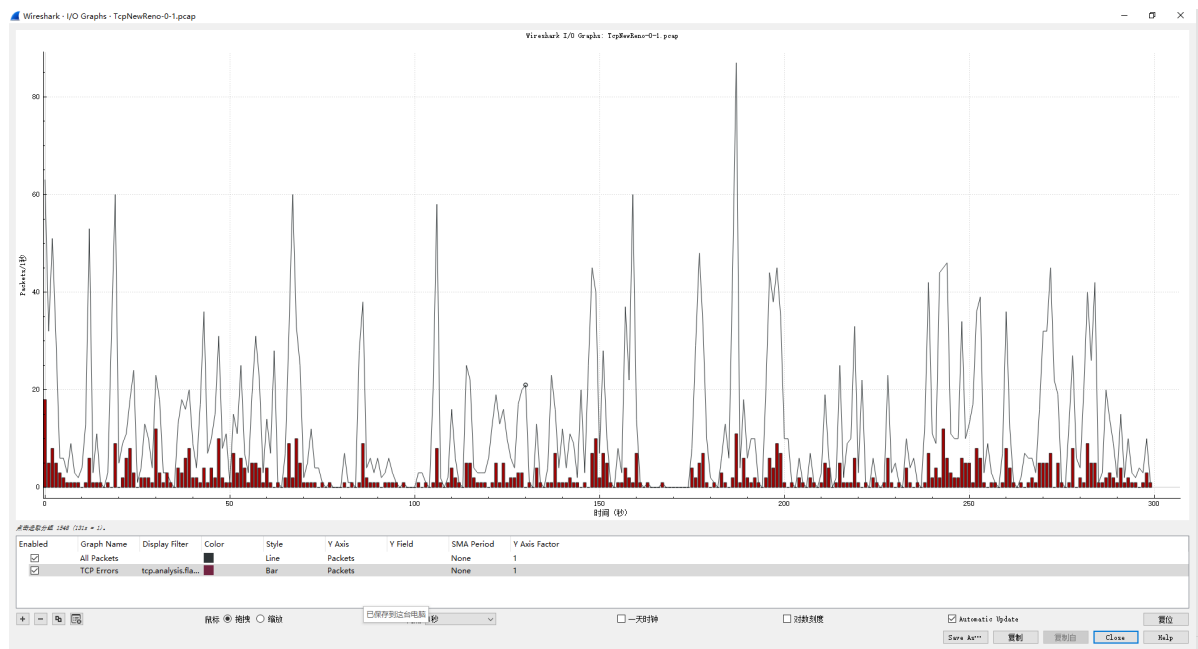


两个算法的rtt随时间的变化

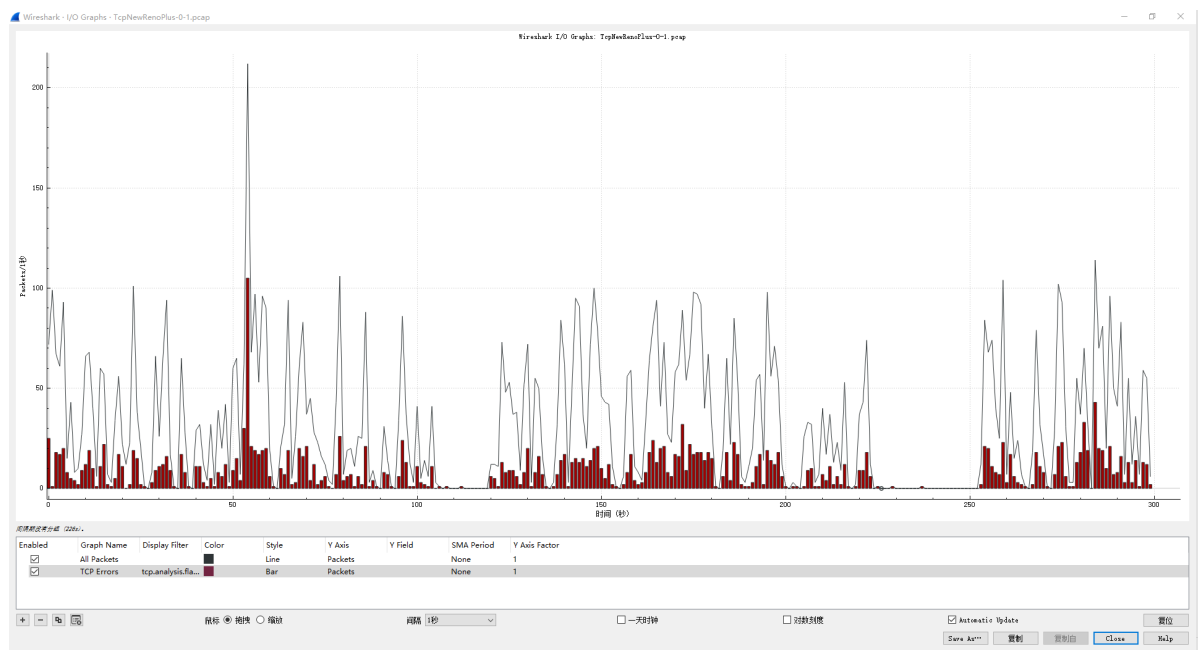


TCPNEWRENO:

线条是那个时刻的吞吐量，条状图是丢包量。



TCPNEWRENOPLUS:



如程序提前结束，可以增大TCP缓冲区，执行如下指令

```
echo "4096 32768 65536" > /proc/sys/net/ipv4/tcp_rmem #tcp收
缓冲区的默认值
```

```
echo "4096 65536 256960" > /proc/sys/net/ipv4/tcp_wmem #tcp发
缓冲区默认值
```

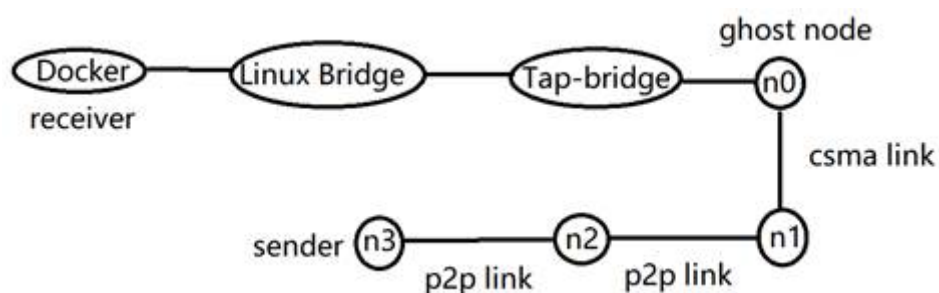
2.2.7 NS3 TO DOCKER实验

NS3作为发送方， Docker容器作为接收方， 设置NS3 to Docker模拟测试。对比 TCP NewReno和TCP NewRenoPlus在RTT、吞吐量和丢包速率的表现。

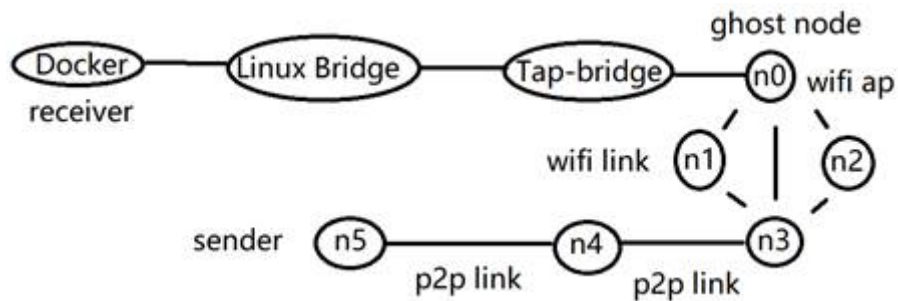
模拟时间 300秒

丢包率 0.5%

Docker容器中的SACK选项关闭。



NS3 TO DOCKER测试CSMA网络拓扑图



NS3 TO DOCKER测试WIFI网络拓扑图

这部分的实验会手动操作多一些，有能力的同学可以根据下面的步骤写出自己的自动化脚本。

首先配置docker和ns3之间的连接。

ns-allinone-3.33 /ns-3.33/ 目录下终端，执行 `sudo ./setup_tap.sh`

```

ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ sudo ./setup_tap.sh
Set 'tap-test1' persistent and owned by uid 0
/proc/sys/net/bridge /home/ydy/Documents/ns-allinone-3.33/ns-3.33
/home/ydy/Documents/ns-allinone-3.33/ns-3.33
2deb0424dc90f62e0a310d53cec8e592b23d66f7035c856dd3e188d143845e54
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$

```

方便起见，同目录下新开一个终端，执行 `pwd`

```

ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ pwd
/home/ydy/Documents/ns-allinone-3.33/ns-3.33
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$

```

得到当前目录的地址，如 `/home/xxx/workspace/ns-allinone-3.33/ns-3.33/`

然后执行下面这条指令，将接收方出现拷贝到docker容器 `testvm123` 中。

`docker cp /home/xxx/workspace/ns-allinone-3.33/ns-3.33/scratch/testdocker.cpp testvm123:/app/`

```

ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ sudo docker cp /home/ydy/Documents/ns-allinone-3.33/ns-3.33/scratch/testdocker.cpp testvm123:/app/
[sudo] password for ydy:
Successfully copied 3.58kB to testvm123:/app/
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$

```

然后执行 `docker exec -it testvm123 /bin/bash`，进入docker容器内，执行如下代码，关闭TCP接收方的SACK选项

`echo 0 > /proc/sys/net/ipv4/tcp_sack`

`sysctl -w net.ipv4.tcp_sack=0`

然后在docker容器执行 `g++ -o testdocker testdocker.cpp`，编译接收方程式。

```
Successfully copied 3.58kB to testvm123:/app/
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ sudo docker exec -it testvm123 /bin/bash
root@2deb0424dc90:/app# 020520Ydy
bash: 020520Ydy: command not found
root@2deb0424dc90:/app# echo 0 > /proc/sys/net/ipv4/tcp_sack
root@2deb0424dc90:/app# sysctl -w net.ipv4.tcp_sack=0
net.ipv4.tcp_sack = 0
root@2deb0424dc90:/app# g++ -o testdocker testdocker.cpp
root@2deb0424dc90:/app#
```

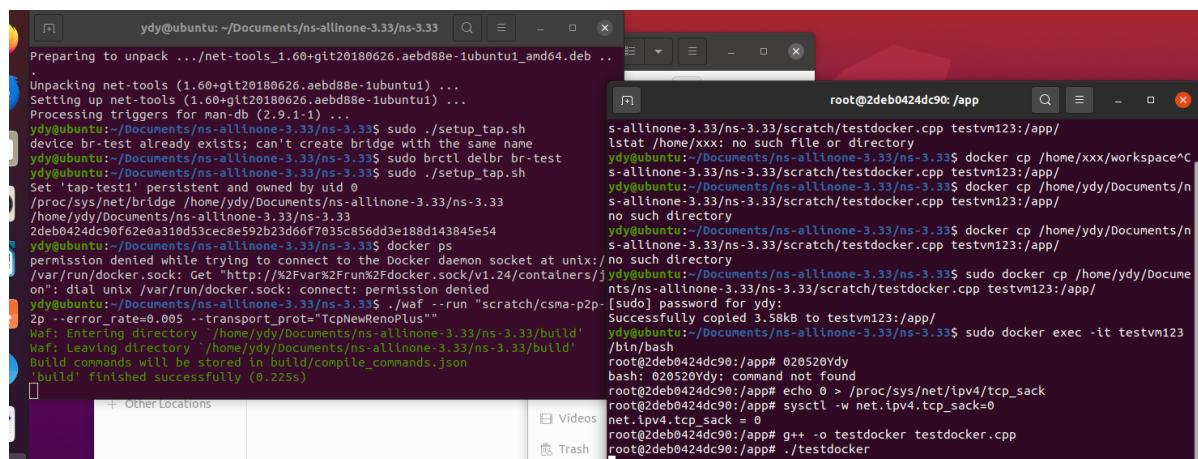
在启动NS3程序之后，执行 `./testdocker`，接收数据包，请先看下面的步骤。

在第一个终端或者 `ns-allinone-3.33 /ns-3.33/` 目录下新建一个终端，执行

```
./waf --run "scratch/csma-p2p-p2p --error_rate=0.005 --
transport_prot="TcpNewRenoPlus""
```

```
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ ./waf --run "scratch/csma-p2p-p2p --error_rate=0.005 --transport_prot="TcpNewRenoPlus""
Waf: Entering directory '/home/ydy/Documents/ns-allinone-3.33/ns-3.33/build'
Waf: Leaving directory '/home/ydy/Documents/ns-allinone-3.33/ns-3.33/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.225s)
```

然后执行 `./testdocker`，结果啥都不出现，终端就像卡住了，过了300秒(5分钟)在ns-3.33文件夹下生成了pcap文件



```
ydy@ubuntu:~/Documents/ns-allinone-3.33/ns-3.33$ ./waf --run "scratch/csma-p2p-p2p --error_rate=0.005 --transport_prot="TcpNewRenoPlus""
Waf: Entering directory '/home/ydy/Documents/ns-allinone-3.33/ns-3.33/build'
Waf: Leaving directory '/home/ydy/Documents/ns-allinone-3.33/ns-3.33/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.225s)

root@2deb0424dc90:/app# 020520Ydy
bash: 020520Ydy: command not found
root@2deb0424dc90:/app# echo 0 > /proc/sys/net/ipv4/tcp_sack
root@2deb0424dc90:/app# sysctl -w net.ipv4.tcp_sack=0
net.ipv4.tcp_sack = 0
root@2deb0424dc90:/app# g++ -o testdocker testdocker.cpp
root@2deb0424dc90:/app# ./testdocker
```

这条指令是测试csma链路的TcpNewRenoPlus，改为TcpNewReno即可测试原版NewReno。

同理wifi链路

```
./waf --run "scratch/wifi-p2p-p2p --error_rate=0.005 --
transport_prot="TcpNewRenoPlus""
```

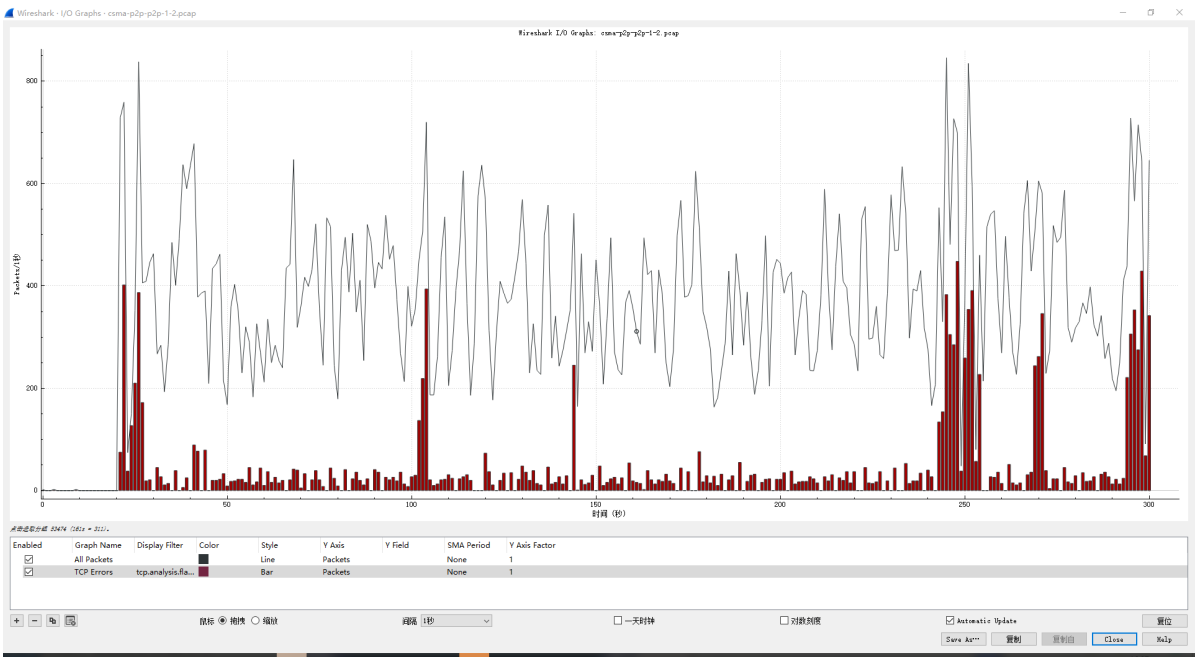

启动NS3程序出现 'build' finished successfully之后，即可运行docker内的c++程序。

吞吐量和丢包速率如NS3内部实验一样在I/O图表获得，RTT需在统计->TCP流图形->往返时间中获得。

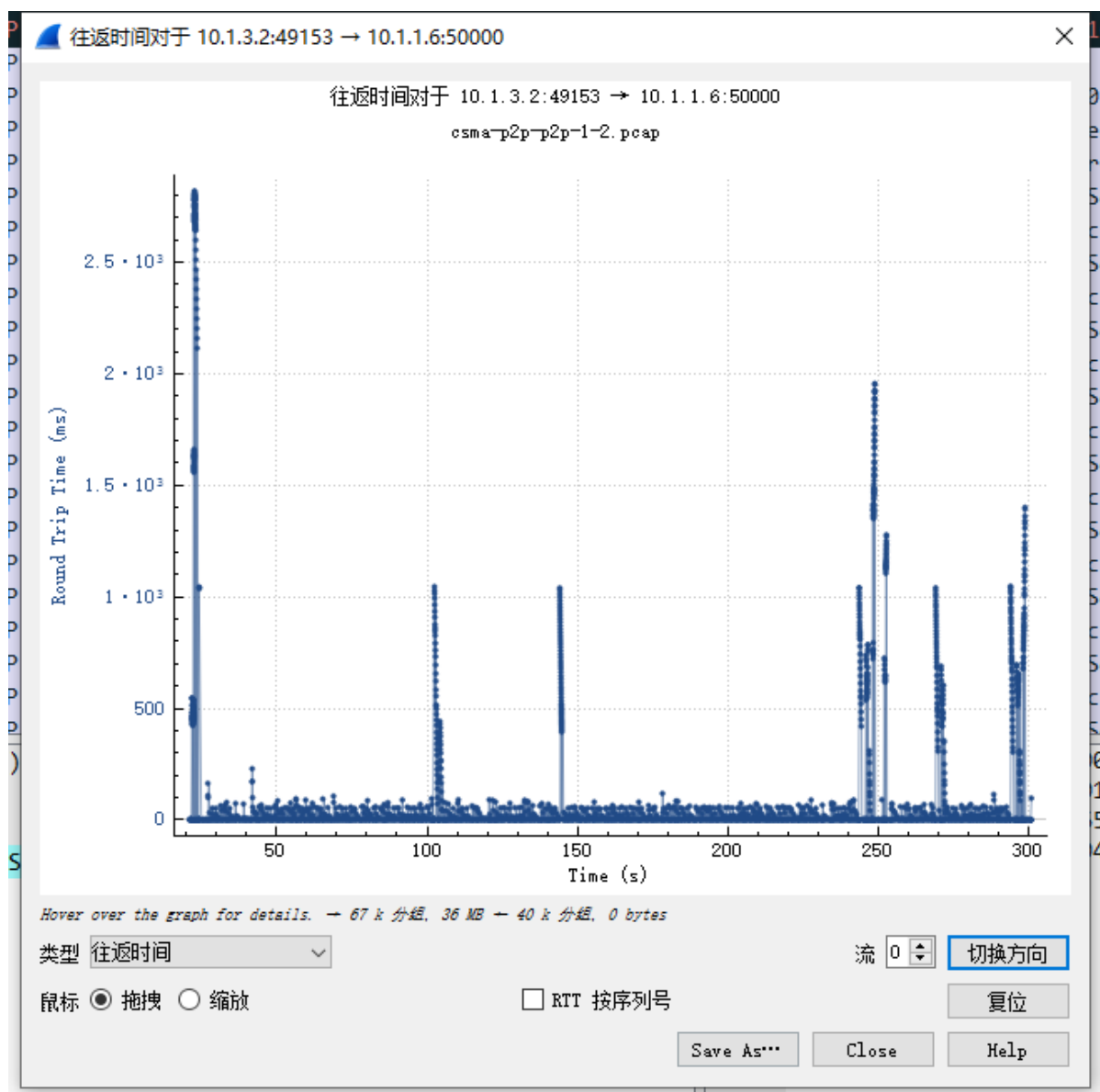
测试csma链路的结果如下

- TCPNEWRENO

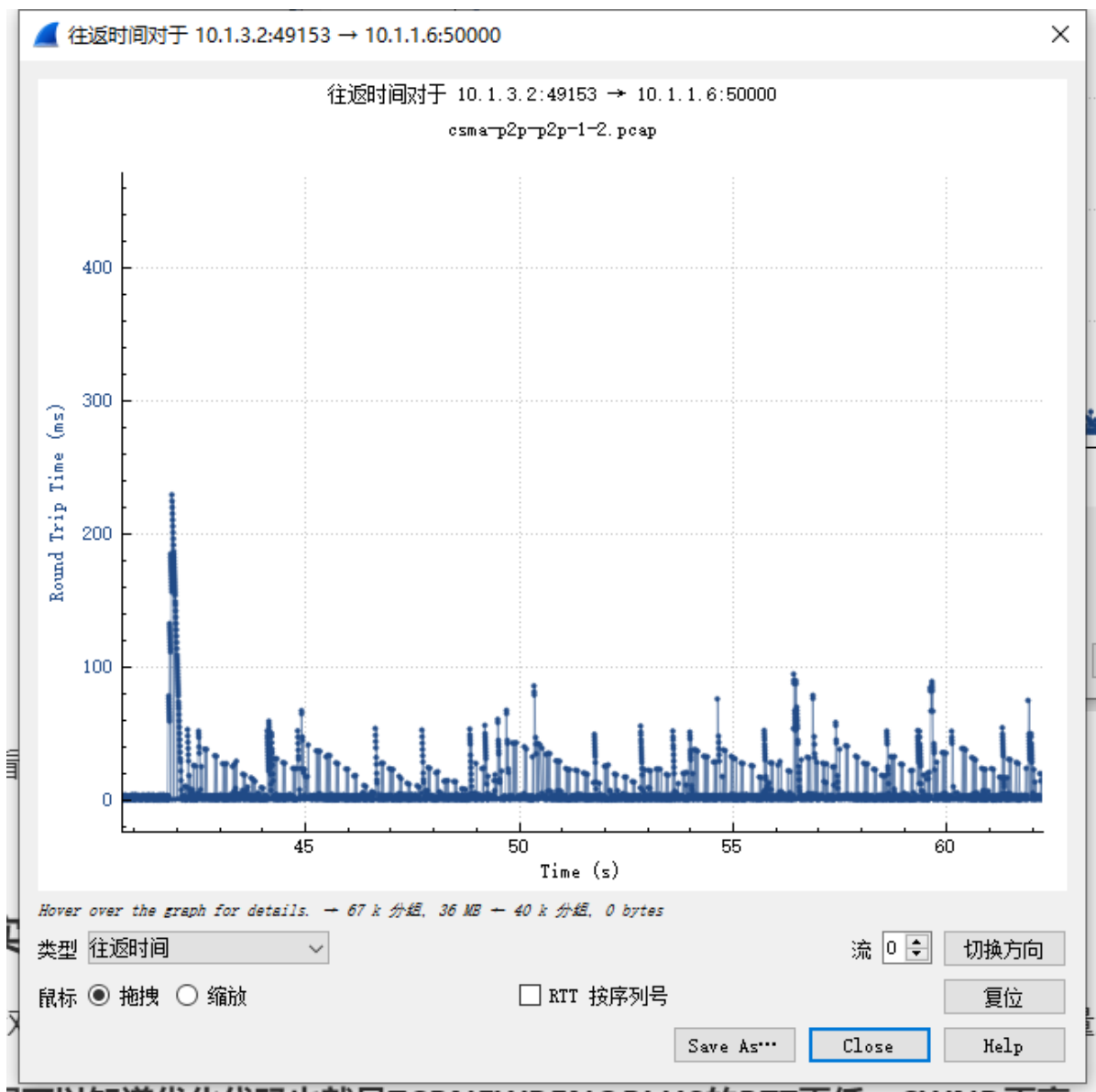
IO图



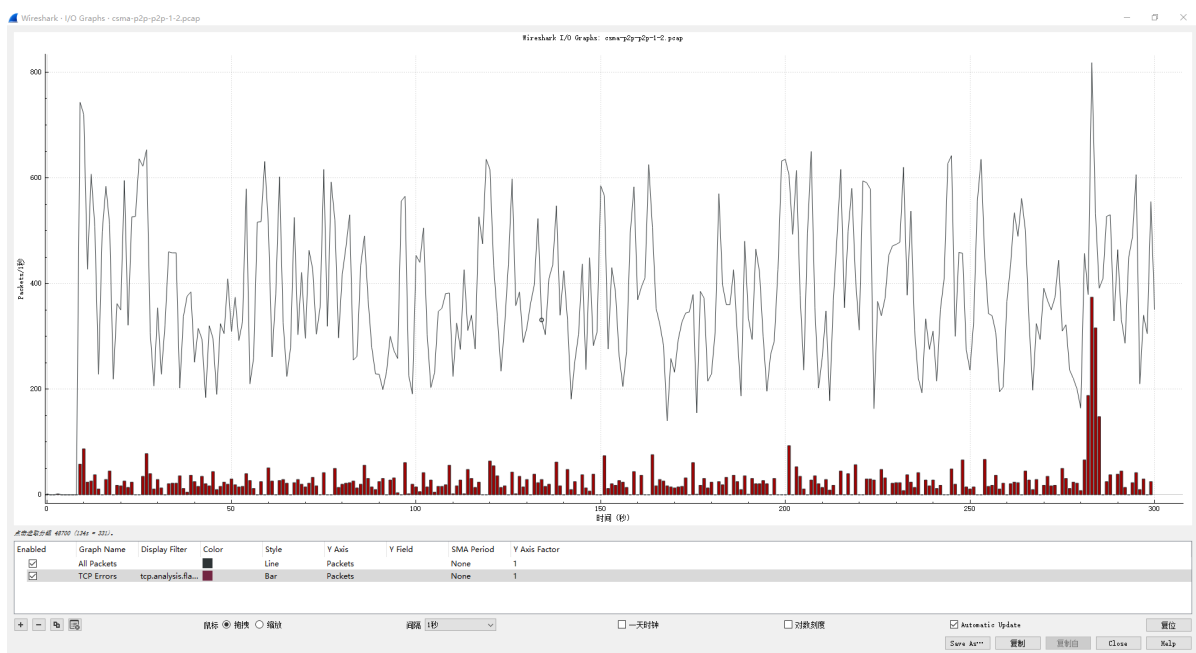
RTT图

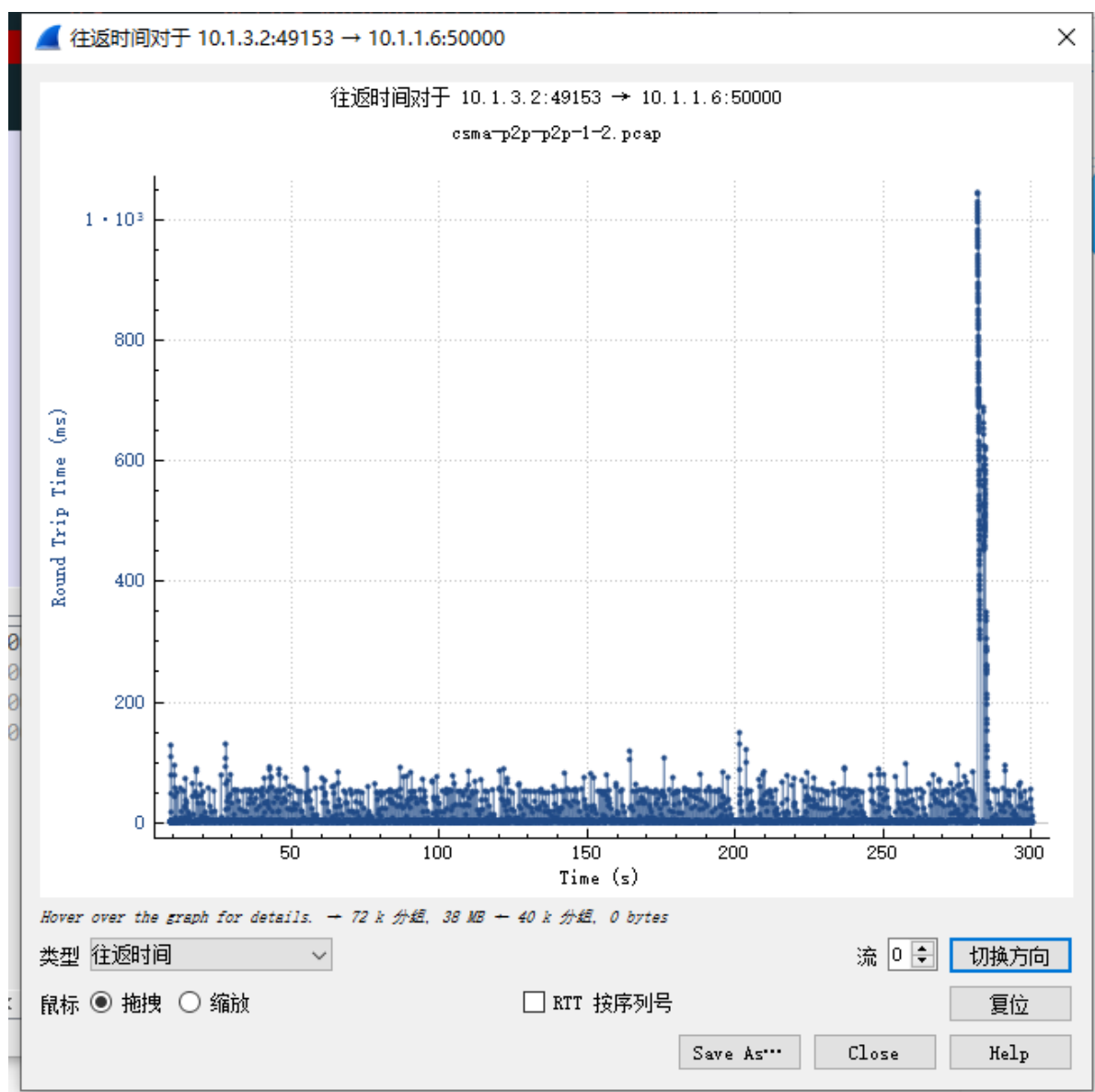


放大了看，RTT大部分都在50ms左右



- TCPNEWRENOPLUS





2.2.8实验分析

(1) 请对比TCP NewReno 和 TCP NewRenoPlus 在NS3内部实验里 RTT、吞吐量、拥塞窗口和丢包速率的表现，并分析原因。

从对比图可以知道优化代码也就是TCPNEWRENOPLUS的RTT更低，CWND更高。丢包率也更高，吞吐量更大

- NewReno在快速恢复阶段中，无论网络状态如何，都会将其拥塞窗口减半。NewReno的另一个问题是，当没有发生丢包，但是数据包不守序地到达接收方，发送方可能会收到超过三个重复确认的数据包，TCP NewReno错误地进入快速恢复状态，并将拥塞窗口减半。由于 TCP 的拥塞窗口控制着 TCP 发送方在任何 时间内可以通过网络发送的数据包数量，因此将拥塞窗口设置为其值的一半会使 TCP NewReno 的链路利用率低下。

- NewReno Plus就是尝试解决这个问题。计算新的拥塞窗口 (cwndn) 的时候, 将sssthresh设置为两个报文段值和cwndn之间的最大值, 并将cwnd设置为sssthresh值加上接收到的重复确认的数量, 并继续留着快速恢复阶段。因此CWND平均会比NEWRENO更高。
- RTT小, 窗口增长就快, 算法又导致CWND比NEW RENO更高, 抢占资源, rtt就不会过高。
- CWND高, 一次性发的数据报就多, 吞吐就大, 吞吐大不可避免的丢包多。

(2) 请对比TCP NewReno 和 TCP NewRenoPlus 在NS3 TO DOCKER实验里RTT、吞吐量和丢包速率的表现, 并分析原因。

- NR(NEW RENO)和NRP(NEW RENO PLUS)两者的吞吐量差不多, 在两者网络状况差不多的情况下, NRP响应更快, 算法更优; 两者的丢包率NRP要更加优秀
- NRP的算法中TCP发送方对每个接收到的重复确认都将cwnd增加一个报文段值, 并在允许的情况下发送新的数据段。利用可用的网络资源保持CWND不过于缩小, 允许更多数据包传输到目的地(CWND更大), 吞吐也更大(第一次实验就是NRP的吞吐量明显更大, 这次两者差不多。)是尽量减小拥塞的可能性, 根据网络拥塞状况来减小发送速率, 所以在两者网络环境差不多的情况下, NRP的吞吐和丢包都更加优秀

三、实验总结

一些问题

1. 使用 `sudo apt-get install` 命令安装uml-utils,bridge-utils,nettools
2. 如果看到以下提示: `device br-test already exists; can't create bridge with the same name`,使用命令删除这个网桥: `sudo ip link del br-test`
3. 以及不知道 `setup_tap.sh` 到底放在哪, 之前说是在scratch文件夹里面, 现在又说在ns-3.33目录下执行, 不知道到底咋搞。
4. 前面提到过生成pcap文件就在ns-3.33的目录下, 找了半天, 以为还在data目录下面, 以及运行的时候没什么提示, 奇奇怪怪都不知道做对没有