lab 5: Test TCP Performance

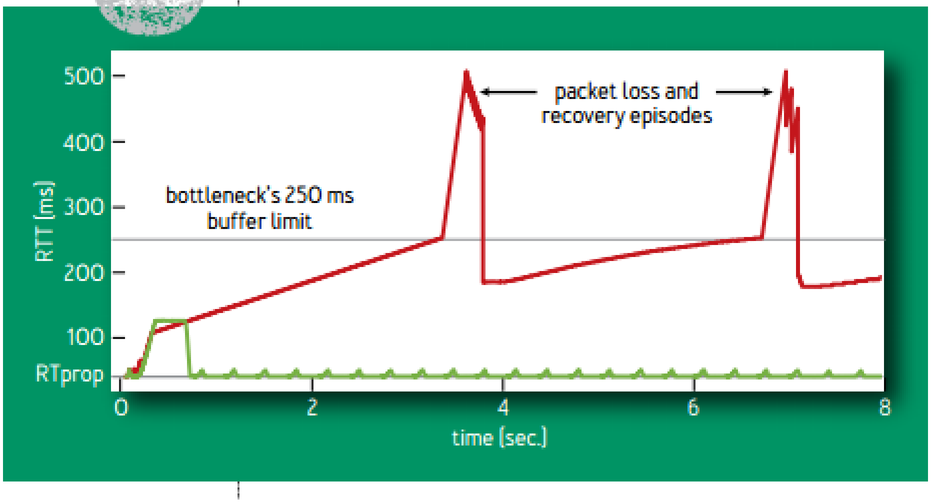
Q1:The BBR Congestion Control Algorithm

The BBR algorithm aims to solve 2 problems:

1. making full use of the bandwidth in the network
2. decrease the buffer occupancy rate in the link.

   The traditional TCP congestion control increase the sender windowsize until loss packet, caused the latency and the "bufferbloat" which caused the packet loss.

FIGURE 5: FIRST 8 SECONDS OF 10-MBPS, 40-MS CUBIC AND BBR FLOWS



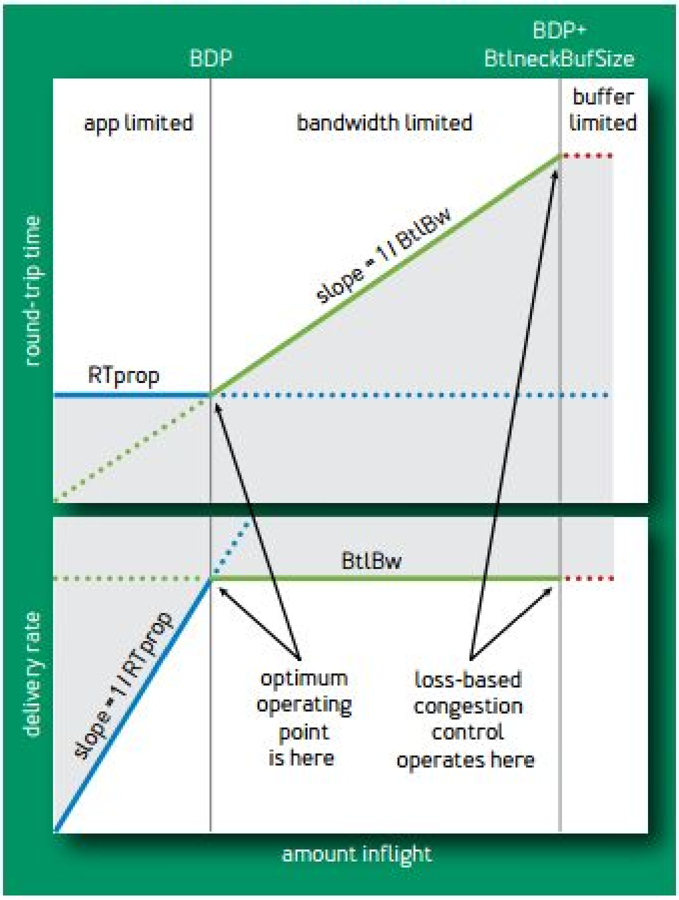
The TCP BBR doesn't estimate the latency×bandwidth either the packet loss. It estimates them separately.

The latency×bandwidth=the windowsize in the traditional algorithm,

TCP BBR measures the bandwidth and the latency alternately, using the maximum bandwidth and the minimum latency within a period of time as the estimated value.

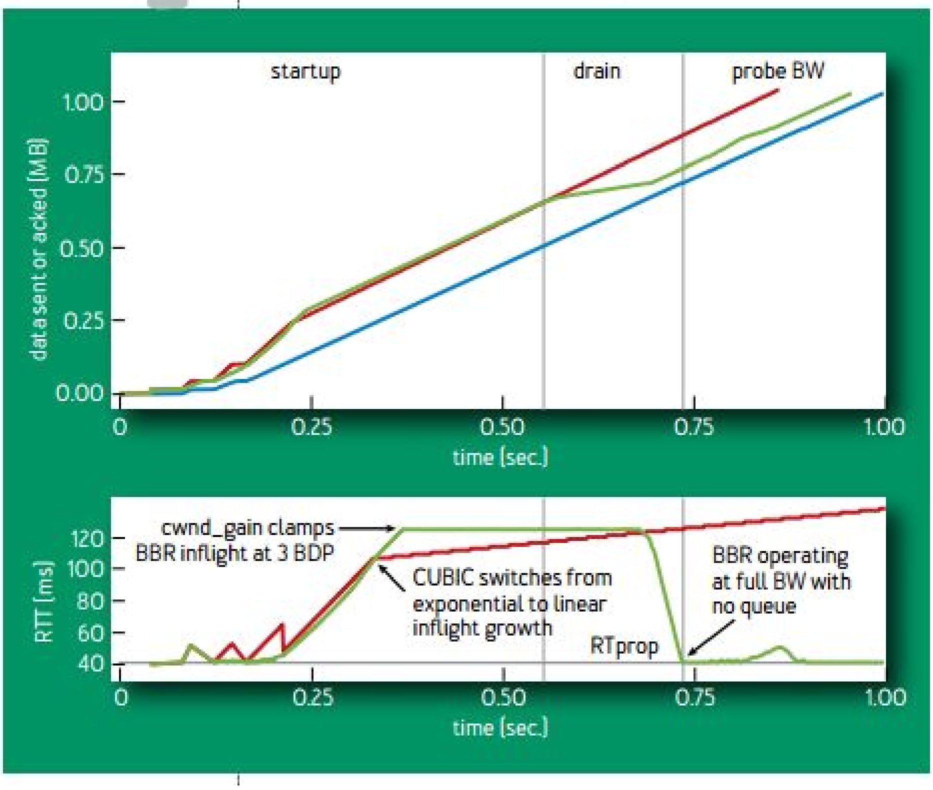
At the begging of the connection establishment, using "Slow Start", it enters the congestion avoidance stage when it finds that the effective bandwidth no longer increases.

FIGURE 1: DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT

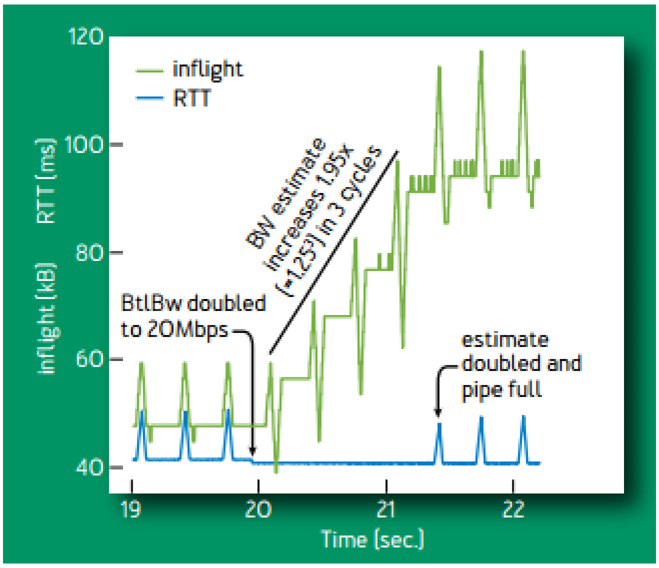


After the slow start is over, in order to consume 2 times the bandwidth delay, the BBR will enter the drain phase, exponentially reduce the sending rate, at this time the packets in the buffer are slowly drained until the round-trip delay is not Lower again. As shown by the green line in the figure below.

FIGURE 4: FIRST SECOND OF A 10-MBPS, 40-MS BBR FLOW



After the emptying phase is over, the BBR enters a stable operating state, alternately detecting bandwidth and delay. Because the network bandwidth changes more frequently than the delay changes Fan, the BBR is in a stable state most of the time in the bandwidth detection stage. The bandwidth detection stage is a positive feedback system: periodically try to increase The packet rate, if the rate of receiving acknowledgments has also increased, further increase the packet sending rate. Specifically, with every 8 round-trip delay as the cycle, at the time of the first round-trip, BBR tries to increase the packet transmission rate by 1/4 (that is, the estimated bandwidth Wide 5/4 speed transmission). At the time of the second round-trip, in order to empty the packets that were sent in the previous round-trip, BBR estimates the bandwidth On the basis of this, reduce 1/4 as the packet sending rate. With 6 round trips left, BBR uses the estimated bandwidth to send packets. When the network bandwidth is doubled, the estimated bandwidth will increase by 1/4 per cycle, and each cycle is 8 round-trip delays. Where the upward spike is Try to increase the packet sending rate by 1/4, the downward spike is to reduce the packet sending rate by 1/4 (empty phase), after 6 round-trip delays, use the updated Estimate the bandwidth. After 3 cycles, that is, 24 round-trip delays, the estimated bandwidth reaches the increased network bandwidth.



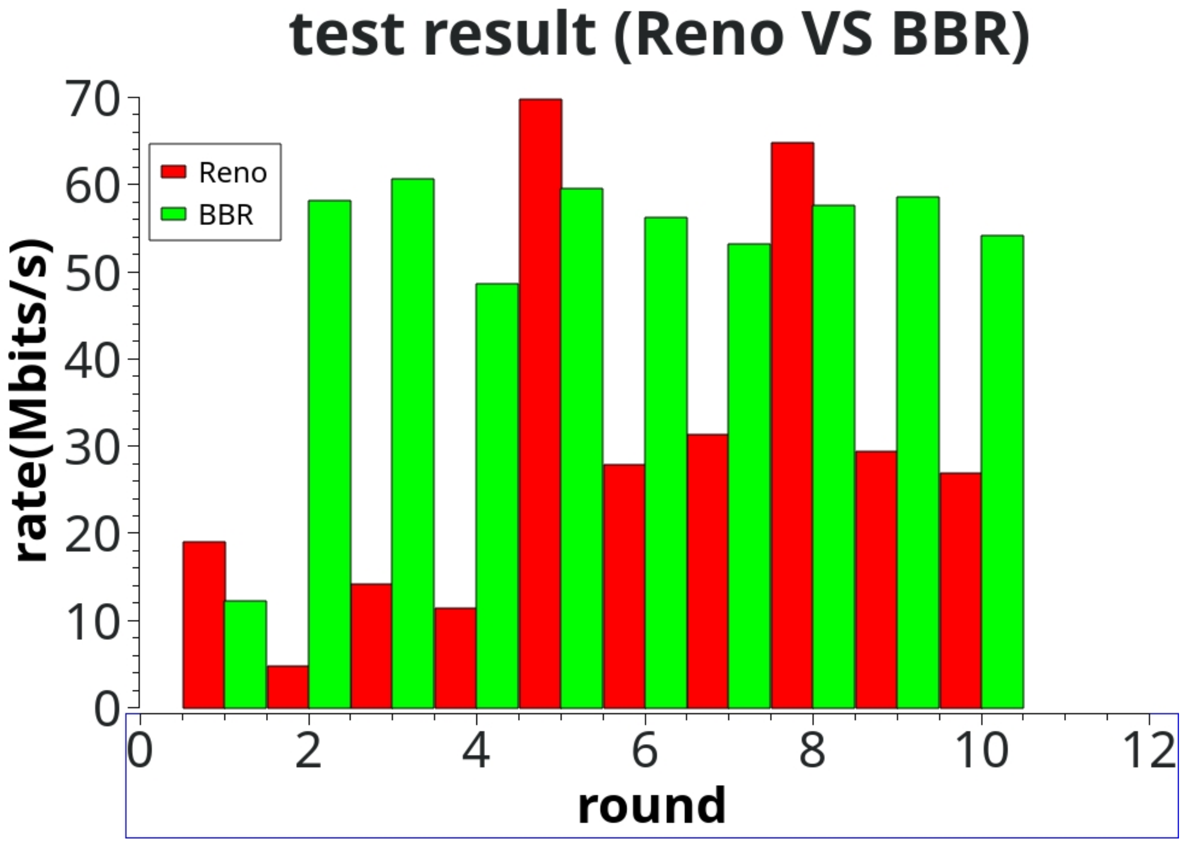
Q2:Test the congestion algorithm

the Reno algorithm

round	1	2	3	4	5	6	7	8	9	10
bandwidth between h1 and h2(Mbits/s)	18.9	4.73	14.1	11.3	69.7	27.8	31.3	64.7	29.3	26.8

the BBR algorithm

round	1	2	3	4	5	6	7	8	9	10
bandwidth between h1 and h2(Mbits/s)	12.2	58.1	60.6	48.5	59.5	56.2	53.1	57.5	58.5	54.1



I tested 10 rounds and the BBR is more stable than the Reno results.

Q3:Construct a network with only one pair of sender and receiver. Study how TCP throughput varies with respect to link bandwidth/link delay/loss rate for the above two TCP versions.

Reno

1. the linkbandwidth(linkdelay=5ms, lossrate=0)

bandwidth(Mbit)	100	200	300
test result(Mbits/s)	98.6	197	289

2. the linkdelay (bandwidth=100Mbit, lossrate=0)

latency(ms)	5	50	100	250	500
test result(Mbits/s)	103	88.6	71.4	13.3	0.943

3. the lossrate(linkdelay=5ms, bandwidth=100Mbit)

lossrate(%)	0	0.1	0.2	0.4	0.6	0.8
test result(Mbits/s)	103	75.3	31.8	18.2	9.32	13.1

BBR

1. the linkbandwidth(linkdelay=5ms, lossrate=0)

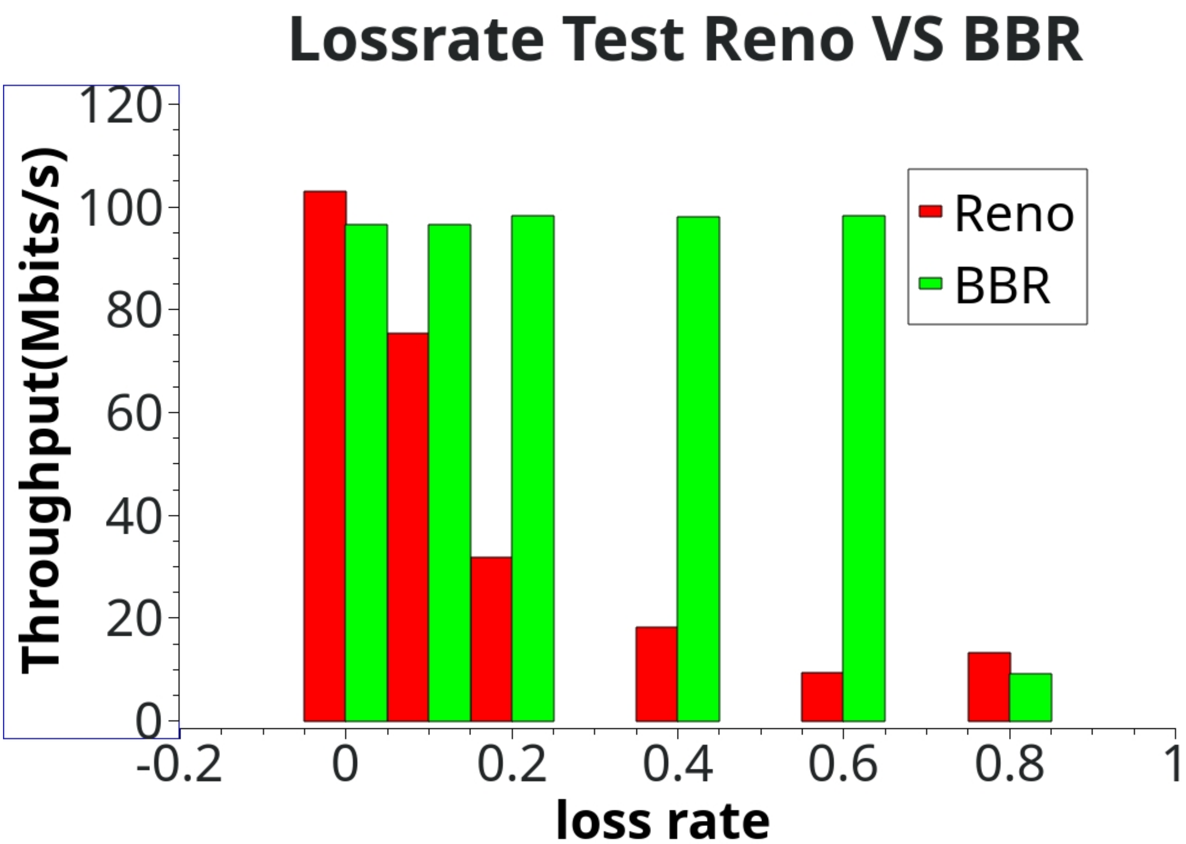
bandwidth(Mbit)	100	200	300
test result(Mbits/s)	96.6	196	290

2. the linkdelay (bandwidth=100Mbit, lossrate=0)

latency(ms)	5	50	100	250	500
test result(Mbits/s)	97	89.2	70.4	14.2	1

3. the lossrate(linkdelay=5ms, bandwidth=100Mbit)

lossrate(%)	0	0.1	0.2	0.4	0.6	0.8
test result(Mbits/s)	96.4	96.5	98.3	97.9	98.1	8.96



The advantage of BBR more concentrates on the loss rate, with the loss rate increasing, the BBR get a more stable results nearby 100Mbits/s.

Q4: Bandwidth Share

BBR

In BBR, I set multi-pairs client and the server to test the Bandwidth Share. The bottleneck bandwidth is 100Mbit.

The results shows as follows:

Pairs	2	3	4	5
Bandwidth share per pair	1:1	32.6%:43.2%:23.3%	20.8%:19.5%:31.5%:27.6%	16.5%:20.4%:30.7%:15.9%:17.3%

Reno

Pairs	2	3	4	5
Bandwidth share per pair	54.8%:55.8%	45.8%:19.7%:35.1%	6.53%:22.9%:70.8%:13.1%	16.2%:31.9%:27.9%:22.8%:27.8%

from the results we can see the BBR implement more fair bandwidth share.

Codes

```
 #!/usr/bin/python
"""
Simple example of setting network and CPU parameters
"""
import threading
import time
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import OVSBridge
from mininet.node import CPULimitedHost
from mininet.link import TCLink
from mininet.util import quietRun, dumpNodeConnections
from mininet.log import setLogLevel, info
from mininet.cli import CLI
from sys import argv
import time
from threading import Thread

"""重新定义带返回值的线程类"""


class MyThread(Thread):
    def __init__(self, func, kargs):
        super(MyThread, self).__init__()
        self.func = func
        self.kargs = kargs

    def run(self):
        self.result = self.func(**self.kargs)

    def get_result(self):
        try:
            return self.result
        except Exception:
            return None
```

```python
# It would be nice if we didn't have to do this:
# pylint: disable=arguments-differ
class SingleSwitchTopo( Topo ):
    def build( self ):
        switch1 = self.addSwitch('s1')
        switch2 = self.addSwitch('s2')
        host1 = self.addHost('h1', cpu=.25)
        host2 = self.addHost('h2', cpu=.25)
        host3 = self.addHost('h3', cpu=.25)
        host4 = self.addHost('h4', cpu=.25)
        host5 = self.addHost('h5', cpu=.25)
        host6 = self.addHost('h6', cpu=.25)
        host7 = self.addHost('h7', cpu=.25)
        host8 = self.addHost('h8', cpu=.25)
        host9 = self.addHost('h9', cpu=.25)
        host10 = self.addHost('h10', cpu=.25)
        self.addLink(host1, switch1, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host2, switch1, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(switch1, switch2, bw=100, delay='5ms', loss=0.1, use_htb=True)
        self.addLink(host3, switch1, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host4, switch1, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host5, switch1, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host6, switch2, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host7, switch2, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host8, switch2, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host9, switch2, bw=100, delay='5ms', loss=0, use_htb=True)
        self.addLink(host10, switch2, bw=100, delay='5ms', loss=0, use_htb=True)
def Test(tcp):
    "Create network and run simple performance test"
    topo = SingleSwitchTopo()
    net = Mininet( topo=topo,
                   host=CPULimitedHost, link=TCLink,
                   autoStaticArp=False )
    net.start()
    info( "Dumping host connections\n" )
    dumpNodeConnections(net.hosts)
    # set up tcp congestion control algorithm
    output = quietRun( 'sysctl -w net.ipv4.tcp_congestion_control=' + tcp )
    assert tcp in output
    info( "Testing bandwidth between h1 and h4 under TCP " + tcp + "\n" )
    h1,h2,h3,h4,h5,h6,h7,h8,h9,h10 = net.getNodeByName('h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8', 'h9', 'h10')
    t1 = MyThread(net.iperf, kargs={"hosts":[h1,h6],"seconds":10})
    t2 = MyThread(net.iperf, kargs={"hosts":[h2,h7],"seconds":10})
    t3 = MyThread(net.iperf, kargs={"hosts":[h3,h8],"seconds":10})
    t4 = MyThread(net.iperf, kargs={"hosts":[h4,h9],"seconds":10})
    t5 = MyThread(net.iperf, kargs={"hosts":[h5,h10],"seconds":10})
```

```
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t5.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
    t5.join()
    res1 = t1.get_result()
    res2 = t2.get_result()
    res3 = t3.get_result()
    res4 = t4.get_result()
    res5 = t5.get_result()
    #_serverbw, clientbw = net.iperf( [ h1, h2 ], seconds=10 )
    info( res1, '\n' )
    info( res2, '\n' )
    info( res3, '\n' )
    info( res4, '\n' )
    info( res5, '\n' )
    CLI(net)
    net.stop()
if __name__ == '__main__':
    setLogLevel('info')
    # pick a congestion control algorithm, for example, 'reno', 'cubic', 'bbr', 'vegas', 'hybla', etc.
    tcp = 'reno'
    Test(tcp)

```