

深圳大学实验报告

课程名称	计算机网络		
实验名称	实验 4：拥塞控制算法		
学 院	计算机与软件学院		
专 业	软件工程（腾班）		
指导教师	张磊		
报 告 人	黄亮铭	学号	2022155028
实验时间	2024 年 5 月 15 日		
提交时间	2024 年 5 月 21 日		

教务处制

一、实验目的与要求

- 1. 理解数据传输过程中可能会发生拥塞
- 2. 了解数据传输过程中发生拥塞时的现象和表现
- 3. 理解拥塞控制算法的基本思想
- 4. 理解不同拥塞控制算法的基本工作原理和区别
- 5. 理解和掌握拥塞控制算法；
- 6. 基于提供的资料和代码，完成任务要求；
- 7. 按照任务要求进行实验，并按要求绘制实验结果图；
- 8. 对实现代码和实验结果进行截图展示；
- 9. 撰写实验报告。

二、实验过程

任务 1：拥塞现象观察

任务目标及要求

- 1. 理解拥塞产生的原因，并观察拥塞所带来的现象；
- 2. 学会使用 OMNet++ 模拟器，对比不同网络状态下数据传输的表现，并通过作图进行分析；
- 3. 利用 OMNet++ 模拟器模拟数据传输，记录不使用拥塞算法时，数据传输在不同网络环境下的性能表现（丢包率（LossRate）以及有效吞吐量（GoodPut））；
- 4. 在多种网络配置下进行对比实验，并将实验数据作图展示，进行分析；

任务步骤

Step1：安装 OMNet++ 模拟器与 INET 框架，并新建项目

- 1. 下载 OMNet++ 与 INET。



 omnetpp.zip	2024/5/13 21:01	ZIP 文件	867,460
 inet4.5.zip	2024/5/13 21:00	ZIP 文件	24,288

图 1：压缩包下载成功

- 2. 具体安装步骤请参考 安装文档。


 OMNeT++ 6.0.3 IDE	2024/5/13 22:05	快捷方式
 OMNeT++ 6.0.3 Shell	2024/5/13 22:05	快捷方式

图 2：软件安装成功

Step2：运行项目并观察拥塞现象

观察拥塞现象并分析原因：在不同网络配置和传输配置下，观察不适用拥塞控制算法时，数据传输过程中 Lossrate 与 Goodput 的变化，并回答下述问题：

- 保持上述的链路参数配置与传输参数配置不变（即默认参数），此时的 Lossrate 与 Goodput 分别是多少？

① 将实验参数配置修改成与实验文档一致。

```
39 **.router*.ppp[*].queue.typename = "DropTailQueue"
40 **.router*.ppp[*].queue.packetCapacity = 2000

39 **.router*.ppp[*].queue.typename = "DropTailQueue"
40 **.router*.ppp[*].queue.packetCapacity = 2000
```

图 3: omnet.ini 文件

```
--
channel Sender1toRouter extends DatarateChannel
{
    delay = 20ms;
    datarate = 2Mbps;
}
channel Sender2toRouter extends DatarateChannel
{
    delay = 20ms;
    datarate = 2Mbps;
}
channel RoutertoReceiver extends DatarateChannel
{
    delay = 20ms;
    datarate = 2.0Mbps; //
}
```

图 4: package.ned 文件

② 运行模拟器。

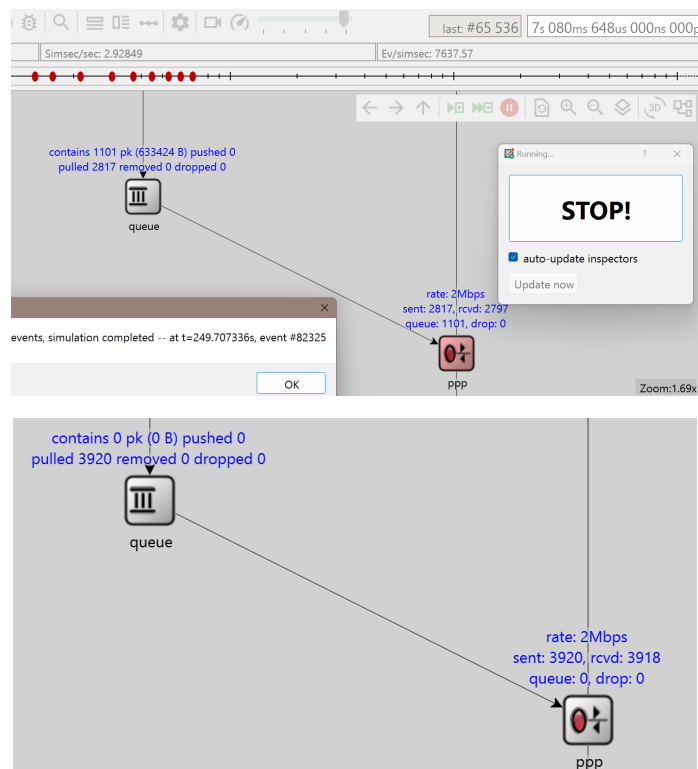


图 5: 模拟器运行结果

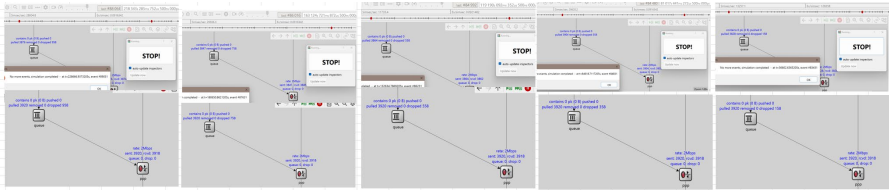
③ 计算得到结果。

$Lossrate: 0.00\%$

$Goodput: 3920 \times 536 \div 7.080 \approx 296768.36(B/s)$

• 其他配置不变，将路由器的队列长度依次更改为[1800, 1600, 1400, 1200, 1000]。此时 Lossrate 与 Goodput 随着队列长度的降低发生了什么变化？为什么会这样？

- ① 更改实验参数配置：将路由器的队列长度依次更改为[1800, 1600, 1400, 1200, 1000]。
- ② 运行模拟器。



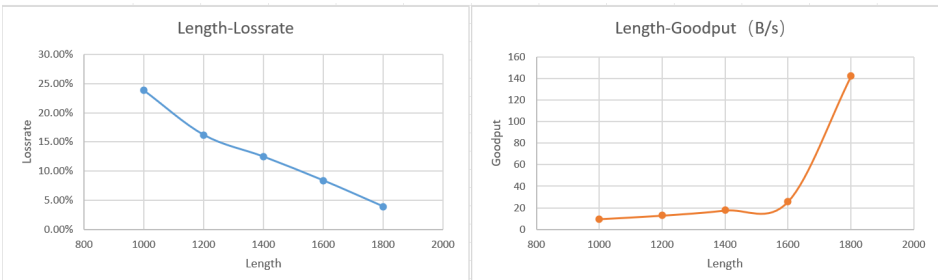
6： 模拟器运行结果

③ 实验结果列表。

单个数据包大小为536B，两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	23.84%	16.20%	12.46%	8.36%	3.88%	0.00%
Goodput (B/s)	9.62	12.88	17.62	25.93	142.4	221057.7
the datarate of RoutertoReceiver: 2.0Mbps						

7： 实验结果

④ 实验结果可视化。



8： 可视化结果

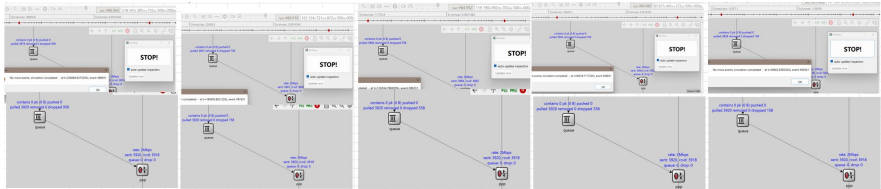
⑤ 分析。

由④中的可视化结果可知：**Lossrate 随着路由器队列长度的降低而增高，呈线性上升趋势；Goodput 随着路由器队列长度的降低而降低，呈指数下降趋势。**

可能的原因：**对于 Lossrate**，路由器队列用于存储数据包，当路由器处理流量时，队列中的数据包等待被转发。如果队列长度较大，能够存储更多的数据包，从而在高流量情况下减少丢包现象。当队列长度减少时，存储数据包的容量变小。在高流量情况下，队列更容易被填满，一旦队列满了，新来的数据包就会被丢弃。因为每减少一个队列单元，就减少了一个数据包存储位置，所以这包率随队列长度减少而增加的关系基本符合是线性的。**对于 Goodput**，当队列长度减少，丢包率增加，网络需要更多的重传来补偿丢失的数据包。重传占用了额外的带宽，导致有效数据传输速率下降。此外，TCP 协议具有拥塞控制机制，当丢包率增加时，TCP 会降低发送速率来适应网络条件。这导致有效吞吐量急剧下降。

• 将路由器的队列长度更改回默认值，并依次更改 RoutertoReceiver (router 与 receiver 之间的链路) 的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。Lossrate 与 Goodput 是如何变化的？为什么？

- ① 更改实验参数配置：更改 RoutertoReceiver (router 与 receiver 之间的链路) 的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。
- ② 运行模拟器。



9： 模拟器运行结果

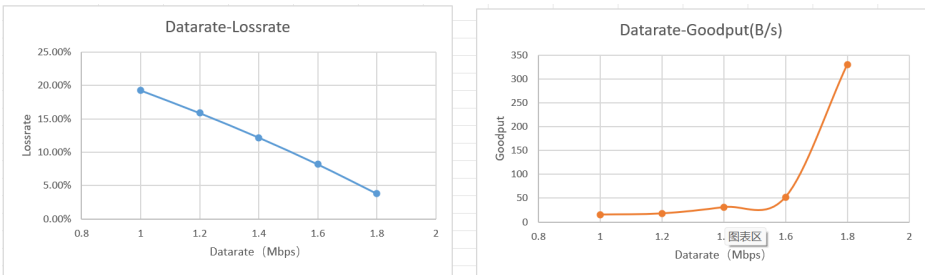
- ③ 实验结果列表。

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	19.28%	15.88%	12.20%	8.18%	3.78%	0.00%
Goodput (B/s)	15.26	17.66	30.87	51.94	330.72	221057.7

the length of Router: 2000

10： 实验结果

- ④ 实验结果可视化。



11： 可视化结果

- ⑤ 分析。

由④中的可视化结果可知：**Lossrate 随着 datarate 的下降而上升，呈线性上升趋势；Goodput 随着 datarate 的下降而下降，呈指数下降趋势。**
可能的原因：**对于 Lossrate**，链路速度决定了数据包传输的速率。较低的链路速度在单位时间内传输的数据包更少，增加了队列中的数据包的积压，提高了丢包的可能性。当链路速度下降时，数据包传输变慢，队列中的数据包的等待时间变长，容易导致队列溢出（队列满）。这种情况下，新的数据包将会被丢弃。由于每单位时间内传输的数据包数量减少，导致丢包率增加。因为链路速度每减少一定比例，会相应地增加丢包的概率，所以这种趋势通常是线性的。**对于 Goodput**，Goodput 是成功传输的有用数据的速率，不包括重传的数据包。它反映了网络的实际有效传输性能。当链路速度下降时，传输速率降低，Goodput 也会相应下降。链路速度下降导致丢包率上升，网络需要更多的重传来补偿丢失的数据包。重传占用了额外的带宽，进一步降低有效数据传输速率。TCP 协议的拥塞控制机制在丢包率增加时，会显著降低发送速率，以适应链路的传输能力。由于 TCP 的拥塞控制调整速率通常是指数级的，这导致 Goodput 下降呈现指数趋势。

总结

产生拥塞的原因：① **链路速度不足**。链路速度决定了数据传输的最大速率。当流量数据超过链路的处理能力时，数据包会在路由器的队列中等待，造成拥塞。如果链路速度太低而数据太多会导致队列溢出，数据包被会丢弃，网络中重传请求增加，占用带宽，进一步加剧拥塞。② **高流量负载**。如果网络中数据量突然增加或者数据量持续高于链路传输能力，容易导致路由器队列迅速积满，导致拥塞。③ **不合理的队列长度**。如果队列设置过短，容易导致队列溢出，增加丢包率，进而引起重传，加剧网络拥塞。如果队列设置过长，可能会增加延迟。

任务二：实现 TCP-reno 拥塞控制算法

任务目标及要求

1. 实现经典的 TCP-reno 拥塞控制算法；
2. 利用 OMNet++ 模拟器模拟数据传输，观察不同场景下传输数据时，TCP-reno 的性能表现，并作图进行分析；
3. 理解并实现 TCP-reno，解释 reno 中每个阶段的开始与退出条件；
4. 在多种网络配置下进行对比（与任务 1 的结果进行对比）实验，并将实验数据作图展示，进行分析；

任务步骤

Step1: 阅读并理解 TCP-reno 的工作流程。

理解 TCP-reno 的设计思想，以及为什么在原有的慢启动(Slow Start)和拥塞避免(Congestion Avoidance)中加入了快速重传(Fast Retransmit)和快速恢复(Fast Recovery)算法？

在传统的 TCP 拥塞控制中，当一个数据包丢失时，发送方需要等待一个超时事件后才会重新发送丢失的数据包。这段等待时间可能会显著降低传输效率。**因此引入快速重传机制**。当发送方收到三个重复的 ACK 时，立即重传认为丢失的数据包，而不必等待超时。这大大缩短了丢包检测和重传的时间，提高了数据传输效率。

传统的 TCP 拥塞控制在检测到丢包后会将拥塞窗口重置为 1 个 MSS，并重新进入慢启动阶段。① 这种剧烈的拥塞窗口减小和慢启动过程会导致传输速率大幅波动，从而影响网络性能。② 此外，发送方仍然能接收来自接收方的确认信息，说明网络拥塞不是十分严重，因此不必将拥塞窗口置为 1 个 MSS，重新启动慢启动过程。**因此引入快速回复机制**。快速恢复机制在检测到丢包后，不是直接进入慢启动，而是将慢启动阈值设置为当前拥塞窗口的一半，并将拥塞窗口设为慢启动阈值的大小，然后进入拥塞避免阶段。这种方法避免了剧烈的速率波动，更快地恢复到合适的传输速率。

Step2: 在 OMNet++ 中实现 TCP-reno.

请根据文件中的代码注释, 补全关键部分的代码 (TODO parts), 实现 OMNet++ 中实现 TCP-reno.

- I 打开 \$YOURWORKSHOP\$/TcpMultipleSender/src/TcpCustomReno.cc 文件。
- II 实现 TcpCustomReno::recalculateSlowStartThreshold(): 用于重新计算慢启动阈值, 决定何时从慢启动阶段切换到拥塞避免阶段。

```
/*
TODO:
当出现网络拥塞时, 发送方可以将慢开始阈值 (ssthresh) 设置为当前拥塞窗口 (cwnd) 的一半,
同时, 根据 RFC 2581, 慢开始阈值 ssthresh 应设置为不超过以下值:
ssthresh = max(FlightSize / 2, 2 * SMSS)
其中, FlightSize 是网络中未确认的数据量, 在此处可表示为目前的拥塞窗口 (cwnd) 大小,
@note: max函数可以直接使用 std::max(parameter1, parameter2), 它会返回两个参
*/
state->ssthresh = std::max(state->snd_cwnd / 2, 2 * state->snd_mss);
```

12: 补充代码

- III 实现 TcpCustomReno::processRexmitTimer(TcpEventCode& event): 处理超时事件的重传定时器, TCP 传输过程中发生数据包丢失会触发该函数, 启动慢开始算法。

```
/*
TODO:
在遇到超时事件时, 也即发生了丢包, TCP Reno 应当启动慢开始算法, 将拥塞窗口设置为1个MSS大小。
*/
TcpCustomReno::recalculateSlowStartThreshold();
state->snd_cwnd = state->snd_mss;
```

13: 补充代码

- IV 实现 TcpCustomReno::receivedDuplicateAck(): 收到重复“重复 ACK 数量阈值”个 ACK 时, 启动快重传算法和快恢复算法。

```
// TODO: 快重传首先需要重新计算慢开始阈值, 然后更改新的拥塞窗口为新的慢开始阈值
TcpCustomReno::recalculateSlowStartThreshold();
state->snd_cwnd = state->ssthresh; // 启动快速恢复

conn->emit(cwndSignal, state->snd_cwnd);

// TODO: 在快重传逻辑中, 并不等待重传定时器到期, 立即重传丢失的数据段
conn->retransmitOneSegment(false);
```

14a: 补充代码

```
// TODO: Reno 规定: 对于每个额外接收到的重复确认
state->snd_cwnd += state->snd_mss;
```

14b: 补充代码

- V 实现 TcpCustomReno::receivedDataAck(uint32_t firstSeqAcked): 在接收方收到位确认的数据包的 ACK 时会被调用。

```
/*
TODO:
如果目前正处于快恢复阶段 (也即收到的重复ACK数量大于或等于重复ACK数量阈值), 并且收到了一个正确的数据
那么 Reno 会认为这是网络恢复的一个信号, 将拥塞窗口重置为慢开始阈值, 重新进入一个稳定的拥塞避免状态。
*/
state->snd_cwnd = state->ssthresh;
```

15a: 补充代码

```
// TODO: 执行慢开始算法, 每增加一个往返就将拥塞窗口值加倍
// note: 注意, 本方法是收到一个数据的ACK时调用的, 而不是每轮往返结束时调用的, 所以逻辑应是每收到一个数据
state->snd_cwnd += state->snd_mss;
```

15b: 补充代码


```
// TODO: 执行拥塞避免算法, 每增加一个往返就将拥塞窗口值加 1 个MSS大小
// note: 注意, 本方法是收到一个数据的ACK时调用的, 而不是每轮往返结束时调用的, 所以这里的窗口增加值应是 1 个 MSS 大小
// 这样才能实现在一轮RTT传输中, 将拥塞窗口的值增加 1 个 MSS 大小
// 并且还要注意, 由于 snd_mss * snd_mss / snd_cwnd 可能为0, 需要处理0值, 使其至少增加1个MSS大小
uint32_t inc = (state->snd_mss * state->snd_mss) / state->snd_cwnd;
if (inc == 0)
    inc = state->snd_mss;
state->snd_cwnd += inc;
```

15c: 补充代码

Step3: 在不同网络环境中运行 TCP-reno。

- 使用任务 1 中默认的链路参数配置与传输参数配置, 此时的 Lossrate 与 Goodput 分别是多少?

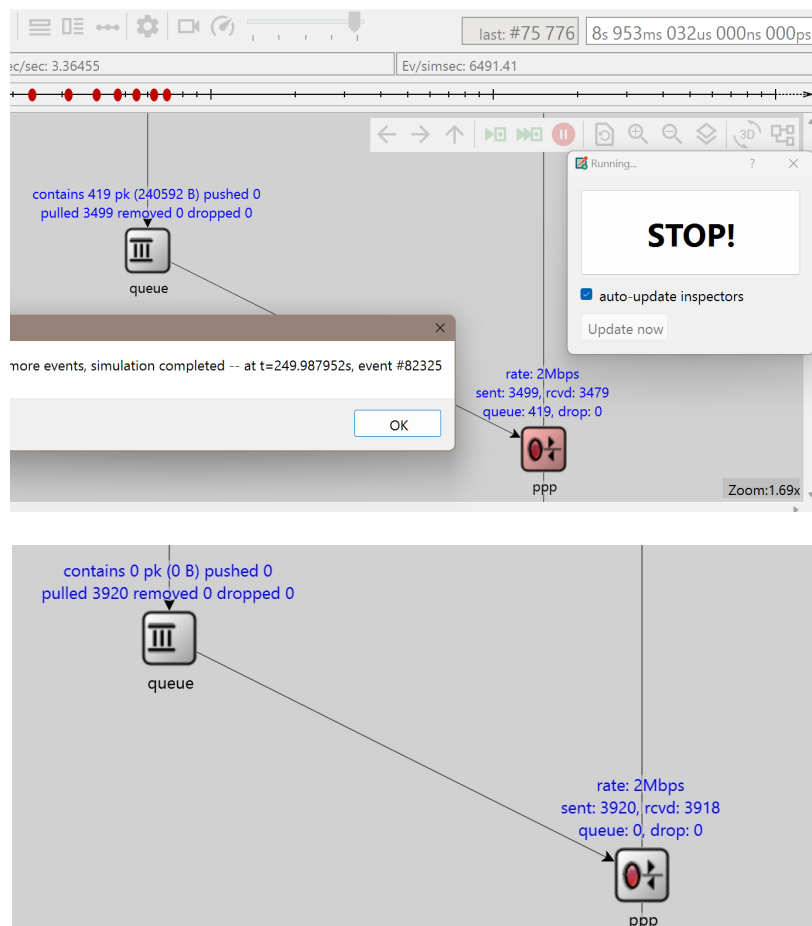
① 将链路参数配置与传输参数配置修改至与任务一默认参数一致。

```
***.tcp.tcpAlgorithmClass = "TcpNoCongestionControl"
**.tcp.tcpAlgorithmClass = "TcpCustomReno"
***.tcp.tcpAlgorithmClass = "TcpBBR"

} **.router*.ppp[*].queue.typename = "DropTailQueue"
l **.router*.ppp[*].queue.packetCapacity = 2000
}
```

16: 实验参数

② 运行模拟器。



17: 模拟器运行结果

③ 计算得到结果。

Lossrate: 0.00%

Goodput: $3920 \times 536 \div 8.953 \approx 234683.34(B/s)$

- 其他配置不变，将路由器的队列长度依次更改为[1800, 1600, 1400, 1200, 1000]。记录 Lossrate 与 Goodput 的变化。
 - ① 更改实验参数配置，将路由器的队列长度依次更改为[1800, 1600, 1400, 1200, 1000]。
 - ② 运行模拟器。



图 18：运行结果

运行结果列表。

单个数据包大小为536B，两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	18.84%	16.20%	12.46%	8.36%	3.88%	0.00%
Goodput (B/s)	157644.7	291018	156766.9	300992.9	299283.2	234683.3
the datarate of Router to Receiver: 2.0Mbps						

图 19：结果列表

运行结果可视化。

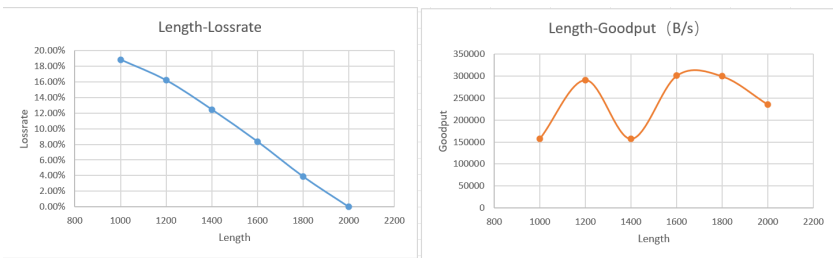


图 20：可视化结果

- 将路由器的队列长度更改回默认值，并依次更改 RoutertoReceiver（router 与 receiver 之间的链路）的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。记录 Lossrate 与 Goodput 的变化。
 - 更改实验参数配置，更改 RoutertoReceiver（router 与 receiver 之间的链路）的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。
 - ② 运行模拟器。



图 20：运行结果

运行结果列表。

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Goodput (B/s)	114096.4	184436.7	217382.7	248158.7	264725.5	221057.7
the length of Router: 2000						

图 21：结果列表

④ 运行结果可视化。

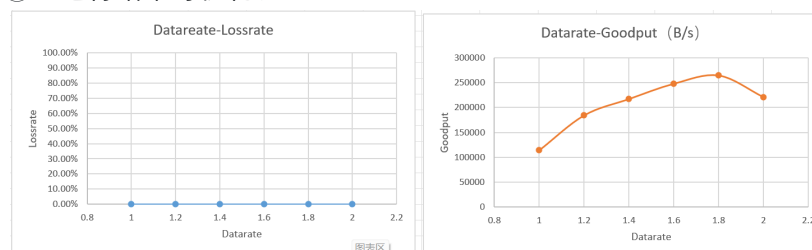


图 22：可视化结果

将使用 TCP-reno 拥塞控制算法下的传输表现进行作图并对比。请解释在 TCP-reno 是如何控制数据包的发送以对进行拥塞控制的？

解释在 TCP-reno 是如何控制数据包的发送以对进行拥塞控制的。

TCP-reno 主要通过慢启动、拥塞避免、快速重传和快速恢复这几个机制协同工作，实现拥塞控制。

- ① 慢启动：初始时，拥塞窗口设为 1 个 MSS。每次成功接收到一个 ACK，cwnd 增加 1 个 MSS。这使得 cwnd 呈指数增长。慢启动一直持续到 cwnd 达到慢启动阈值，然后转入拥塞避免阶段。
- ② 拥塞避免：当 cwnd 达到 ssthresh 时，进入拥塞避免阶段。在拥塞避免阶段，每个 RTT (Round-Trip Time) 内 cwnd 增加 1 个 MSS。这使得 cwnd 线性增长，而非指数增长。
- ③ 快速重传：当发送方收到三个重复的 ACK 时，表明接收方下一个期望的数据包未收到，立即重传被认为丢失的数据包。
- ④ 快速恢复：进入快速恢复阶段时，将 ssthresh 设为当前 cwnd 的一半。将 cwnd 设为 ssthresh。在快速恢复期间，每收到一个重复的 ACK，cwnd 增加一个 MSS，直到未确认的数据包被确认。一旦重传的数据包被确认，cwnd 设为 ssthresh，并进入拥塞避免阶段。

工作步骤：1.连接开始：使用慢启动快速增长 cwnd，直到检测到网络的容量极限 (ssthresh)。2.达到 ssthresh：进入拥塞避免阶段，缓慢增加 cwnd，控制传输速率。3.丢包检测：通过快速重传立即处理丢包，避免等待超时。4.恢复阶段：通过快速恢复机制，迅速恢复传输速度，减小因丢包引起的性能损失。

将使用 TCP-reno 拥塞控制算法下的传输表现进行作图并对比。

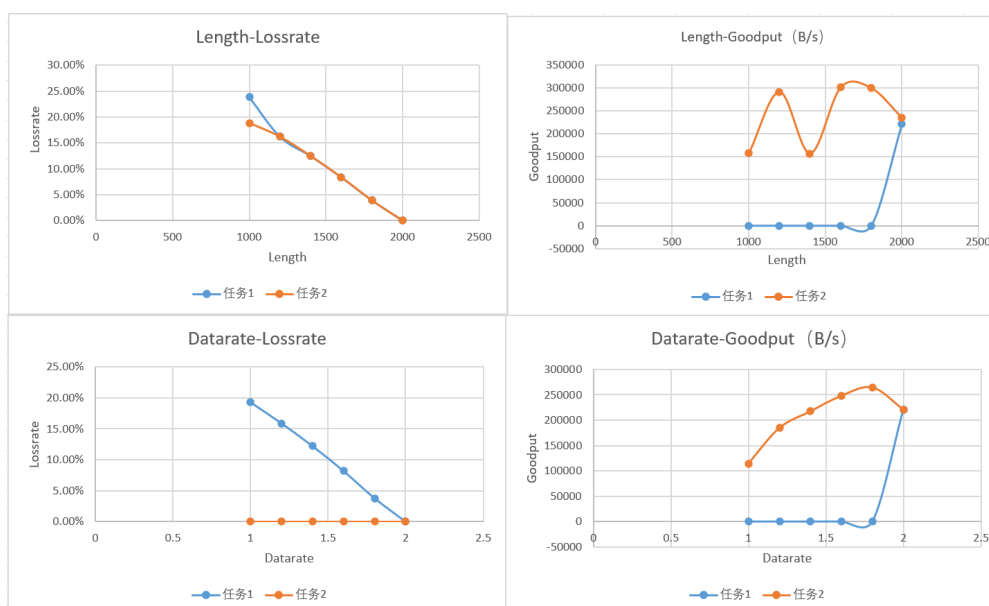


图 23: TCP-reno 和无拥塞控制对比

1. **TCP-reno 和无拥塞控制对于路由器队列长度与丢包率的关系影响相对较小**，只有在路由器长度为 1000 方有差距显现。TCP-reno 控制下和无拥塞控制下路由器队列长度与丢包率的关系均为线性关系。
2. **TCP-reno 和无拥塞控制对于路由器队列长度与有效吞吐量的关系影响较大**。由图 23 可以发现，TCP-reno 控制的有效吞吐量在测量路由器队列长度范围内均比无拥塞控制的有效吞吐量高，证明 TCP-reno 的有效性。
3. **TCP-reno 和无拥塞控制对于链路速度与丢包率的关系影响较大**。由图 23 可以发现，在 TCP-reno 控制下，链路丢包率为 0。而无拥塞控制链路丢包率随链路速度降低而升高。再次证明了 TCP-reno 算法的有效性。
4. **TCP-reno 和无拥塞控制对于链路速度与有效吞吐量的关系影响较大**。由图 23 可以发现，TCP-reno 控制的有效吞吐量在测量路由器队列长度范围内均比无拥塞控制的有效吞吐量高，证明 TCP-reno 的有效性。

任务 3：实现 TCP-bbr 拥塞控制算法

任务目标及要求

1. 实现 TCP-bbr 拥塞控制算法；
2. 利用 OMNet++ 模拟器模拟数据传输，观察不同场景下传输数据时，TCP-bbr 的性能表现，并作图进行分析；
3. 理解并实现 TCP-bbr，解释 bbr 与 reno 这两个拥塞控制算法最核心的区别以及 bbr 的优势；
4. 在多种网络配置下进行对比（与任务 1 的结果、任务 2 的结果进行对比）实验，并将实验数据作图展示，进行分析；

任务步骤

Step1: 阅读并理解 TCP-bbr 的工作流程。

请阅读 TCP-bbr 相关文章和文档：说明 TCP-bbr 与 TCP-reno 最本质的区别是什么？分析 TCP-bbr 相比于 TCP-reno 进行拥塞控制的优势是什么？

最本质的区别：TCP-reno 基于丢包和延迟（超时）进行网络拥塞控制；TCP-bbr 基于带宽和 RTT 进行网络拥塞控制。

优势：① TCP-reno 在检测到丢包后会显著降低发送速率，导致带宽利用率降低。而 TCP-bbr 可以确保发送速率接近实际的瓶颈带宽，高效利用带宽。② TCP-reno 在路由器队列满，而丢包的时候开始进行拥塞控制。一方面数据包排队导致高延迟，另一方面重发数据包导致网络拥塞进一步加剧。TCP-bbr 通过控制发送速率避免队列积压，使数据包以较低延迟发送到接收方。

Step2: 在 OMNet++ 中实现 TCP-bbr。

1. 请打开 \$YOURWORKSHOP\$/TcpMultipleSender/src/XXXXX.cc 文件。
2. 基于给出的代码框架，补充代码内缺失内容并解释实现的内容（标记为 TODO 的部分，部分需要编程实现，部分只需要解释说明）。

TODO 之编程实现部分。

- 计算 RTT，实现 app_limited_until 的更新机制。

```
/*  
    TODO:  
    请计算 rtt，参考论文 p28 中的伪代码  
*/  
rtt = cur_t - packet->sendTime;
```

图 24a: 计算 RTT

```
/*  
    TODO:  
    请根据论文 p28 中伪代码实现 app_limited_until 的更新机制  
*/  
if (app_limited_until > 0) {  
    app_limited_until -= packet->size;  
}  
  
set_app_limited_until(app_limited_until);
```

图 24b: 实现 app_limited_until 的更新机制
根据 sendSize 和 spendT 计算 bw。

```
/*  
    TODO:  
    请根据 sendSize 和 spendT 计算 bw，单位是字节/秒  
*/  
double bw;  
bw = sendSize / spendT;
```

图 25: 计算 bw

- BBR_STARTUP 状态下，对 pacing_gain 和 cwnd_gain 赋值；BBR_DRAIN 状态下，对 pacing_gain 赋值。

```
/*
TODO:
参考论文p51, 对pacing_gain和cwnd_gain赋值为对应的参数, 并解释设置为该参数的原因
*/
pacing_gain = bbr_high_gain;
cwnd_gain = bbr_high_gain;
```

图 26a: BBR_STARTUP 状态下赋值

```
/*
TODO:
参考论文p52, 对pacing_gain赋值为对应的参数, 并解释设置为该参数的原因
*/
pacing_gain = bbr_drain_gain;
```

图 26b: BBR_DRAIN 状态下赋值

实现 BDP 的赋值。

```
/*
TODO:
根据论文p25中对时延带宽积bdp的定义, 实现bdp的赋值
其中时间需要做格式转换 double new_time=(double)old_time.inUnit(SIMTIME_US)/1000000.0;
*/
bdp = ((double)min_rtt.inUnit(SIMTIME_US) / 1000000.0) * btlbwfilter.max_bw;
```

图 27: BDP 赋值

之解释说明部分。

请阅读 `bbr_update_model`，理解 `bbr` 是如何更新 `rtt` 和 `bw` 估计以及状态机的。

更新 bw: TCP-bbr 维护一个滑动窗口，记录一段时间内在链路中存在的最大数据量（正确发送 ACK 的包的序号到当前已发送但未确认的第一个包的序号），同时记录正确发送 ACK 的包的序号的时间到当前时间的时间间隔，最后将两者相除即可得到 bw。

更新 RTT: 每当接收端 ACK 一个数据包时，发送端可以计算该数据包的 RTT。TCP-bbr 维护一个滑动的时间窗口（RTT 值）。如果新的 RTT 值小于当前最小 RTT 值，则更新最小 RTT 值。如果超出一个时间窗口，则直接认为当前 RTT 值为最小 RTT 值。

更新状态机: ① Startup: TCP-bbr 在连接开始时进入 Startup 状态，通过指数增加发送速率，迅速探测并达到瓶颈带宽。当检测到带宽增长停止时，转换到 Drain 状态。② Drain: 在 Drain 状态，BBR 通过降低发送速率，排空网络中的队列，使得数据包的数量接近 BDP（带宽时延积）。Drain 状态结束后，进入 ProbeBW 状态。③ ProbeBW: 在 ProbeBW 状态，BBR 周期性地轻微调整发送速率，探测带宽的变化。大部分时间保持在稳定发送速率，以维持高带宽利用率和低延迟。④ ProbeRTT: 在 ProbeRTT 状态，BBR 通过短时间内显著降低发送速率，探测最小 RTT。ProbeRTT 状态周期性触发，以确保 RTT 估计的准确性。

请问 `bbr` 算法是如何控制发送速率的？拥塞窗口的作用是什么？

通过估算当前瓶颈带宽 bw 和最小 RTT 动态调整发送速率。TCP-bbr 计算带宽时延积确定发送窗口大小，然后根据不同状态选择不同的增益系数。

请问 **bbp** 算法是根据什么计算下一次发送时间间隔的？

通过估算当前瓶颈带宽 bw 和最小 RTT 获得瓶颈带宽估计值，然后根据当前状态的增益系数计算发送速率，最后根据发送速率和数据包大小即可计算出发送间隔。

请根据论文 p26 中的 **BtlBw** 公式解释 **btlbwfilter** 是如何工作的？**rtt_cnt** 的作用是什么？

btlbwfilter 是一个用于过滤和保持最近一段时间内最大带宽值的结构。该结构使用一个循环数组来存储在多个 RTT 期间测量的带宽样本，并在每次更新时计算出这些样本中的最大带宽值并存储在 max_bw 中。

Rtt_cnt 是一个计数器，记录了经过的 RTT (Round-Trip Time) 次数。可以通过对 **rtt_cnt** 的取模实现在循环数组中记录当前样本带宽并删除过时的样本带宽。

请根据论文 p51 解释 **bbr_check_full_bw_reached()** 是如何判断是否达到最大带宽的？

在 **StartUP** 状态下，TCP-bbr 通过 bw 估计管道是否已满。如果它注意到有三轮尝试将交付率翻倍但是实际上几乎没有增加（增长幅度小于 25%），那么它估计 bw 已经是最大带宽，然后退出 **Startup** 并进入 **Drain**。

请从 **StartUP** 阶段的主要任务的角度解释 **btlbwfilter.max_bw >= bw_thresh** 时要将 **count_full_bw_cnt** 的计数重置为 0？

如果当前测量的最大带宽 **btlbwfilter.max_bw** 已经达到了阈值 **bw_thresh**，说明当前的发送速率已经足够高，进一步的增长可能并不会显著提高带宽。这时重置 **count_full_bw_cnt** 的目的是：① 允许算法在后续的 RTT 中继续观察带宽变化，确保不会过早地认为已经达到了瓶颈带宽；② 通过重置计数器，可以避免因短期波动或误测导致的错误判断，从而确保只有在多次 RTT 测量中都确认带宽没有显著增长时，才认为达到了瓶颈带宽。

参考论文 p31 和 p49 解释 **bbp** 算法中 **cycle gain** 的作用。

在 TCP-bbr 中，**cycle_gain** 的作用是通过动态调整发送速率来优化带宽利用率和控制网络中的排队延迟。TCP-bbr 的发送速率调控基于对链路带宽 bw 和最小 RTT 的估计，并且在其控制机制中引入了周期性增益变化，以不同的增益值（**pacing gain**）来调整发送速率。这种方式可以确保既能充分利用带宽，也能避免网络拥塞。

针对 **bbr_is_next_cycle_phase()** 解释三个 **return** 的含义是什么？为什么要区分三种 **return**？

- ① 当 **pacing_gain** 为 1 时，TCP-bbr 处于常规带宽探测阶段。此时，只需判断是否已经过了一个完整的 RTT 周期。如果是，则进入下一个周期阶段。
- ② 当 **pacing_gain** 大于 1 时，TCP-bbr 处于增加发送速率以探测更高带宽的阶段。此时，除了需要经过一个完整的 RTT 周期外，还需要检查发送中的数据量是否达到了按当前增益计算的 BDP。
- ③ 当 **pacing_gain** 小于 1 时，TCP-bbr 处于减速阶段。此时，只需满足以下任一条件即可进入下一个周期阶段：经过了一个完整的 RTT 周期或发送中的数据量低于按增益为 1 计算的 BDP。

参考论文 p51，对 pacing_gain 和 cwnd_gain 赋值为对应的参数，并解释设置为该参数的原因。

StartUP 状态下发送速率应呈指数级增长（每轮翻倍），为了以最平稳的方式实现翻倍，所以将 pacing_gain 和 cwnd_gain 赋值为 $2/\ln 2$ ，这是允许每轮发送速率翻倍的最小值。

参考论文 p52，对 pacing_gain 赋值为对应的参数，并解释设置为该参数的原因。

Drain 状态的目的是快速排空在 StartUP 状态下积压的队列，因此使用 StartUP 状态下使用的 pacing_gain 的倒数来排空队列。综上，pacing_gain 赋值为 $\ln 2/2$ 。

Step3: 在不同网络环境下运行 TCP-bbr。

将 ini 文件中的 `**tcp.tcpAlgorithmClass` 赋值为 "TcpBBR"，使用 TCP-bbr 拥塞控制算法，观察不同网络状态下的数据传输表现。

- 使用任务 1 中默认的链路参数配置与传输参数配置，此时的 Lossrate 与 Goodput 分别是多少？
 - ④ 将实验参数设置为任务 1 中默认的链路参数配置与传输参数配置。
 - ⑤ 运行模拟器。

8s 222ms 261us 000ns 000ps

图 28a：运行时间

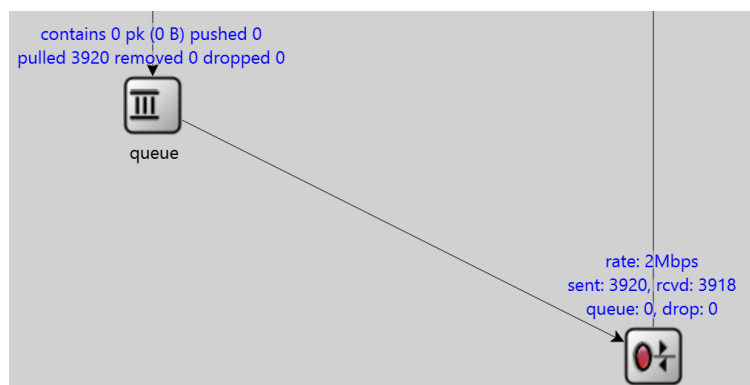


图 28b：发送包数

计算得到结果。

Lossrate: 0.00%

Goodput: $3920 \times 536 \div 8.222 \approx 255548.52(B/s)$

- 其他配置不变，将路由器的队列长度依次更改为[1800, 1600, 1400, 1200, 1000]。记录 Lossrate 与 Goodput 的变化。

- ① 更改实验参数，将路由器的队列长度依次更改为[18 00, 1600, 1400, 1200, 1000]。
- ② 运行模拟器。



图 29：运行结果

③ 实验结果列表。

单个数据包大小为536B，两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Goodput (B/s)	255025.2	180168.1	211806.5	255672.9	215278.7	255548.5

the datarate of RoutertoReceiver: 2.0Mbps

图 30：结果列表

④ 实验结果可视化。

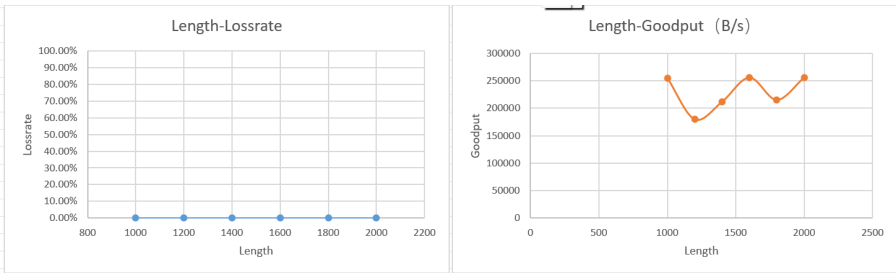


图 31：可视化结果

- 将路由器的队列长度更改回默认值，并依次更改 RoutertoReceiver (router 与 receiver 之间的链路) 的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。记录 Lossrate 与 Goodput 的变化。

更改实验参数，依次更改 RoutertoReceiver (router 与 receiver 之间的链路) 的 datarate 为[1.8Mbps, 1.6Mbps, 1.4Mbps, 1.2Mbps, 1.0Mbps]。

② 运行模拟器。

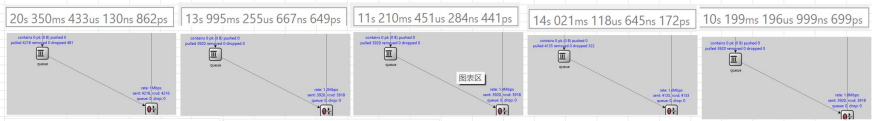


图 32：运行结果

③ 实验结果列表。

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	11.28%	0.00%	0.00%	7.22%	0.00%	0.00%
Goodput (B/s)	111098.2	150133.6	187432.6	158074.3	206012.4	255548.5

the length of Router: 2000

图 33：结果列表

④ 实验结果可视化。

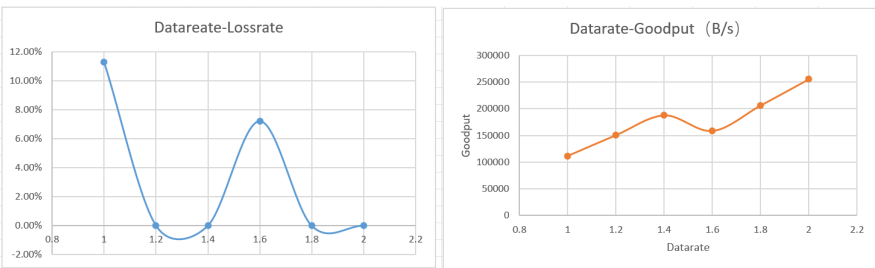


图 34：可视化结果

将使用 TCP-bbr 拥塞控制算法下的传输表现进行作图并对比。请解释在 TCP-bbr 是如何控制数据包的发送以对其进行拥塞控制的？

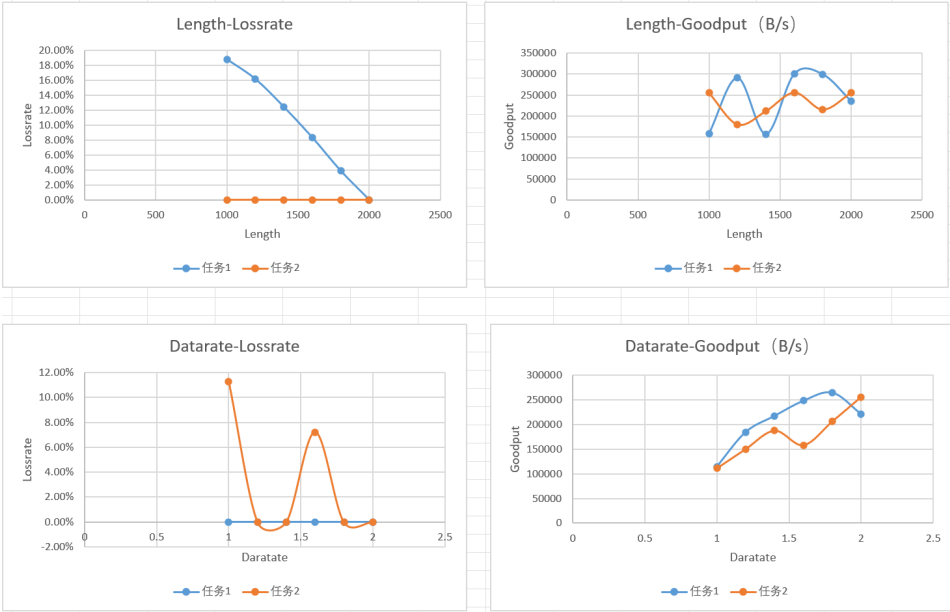


图 35: TCP-bbr 和 TCP-reno 对比

解释在 TCP-bbr 是如何控制数据包的发送以对其进行拥塞控制。

- ① Startup: TCP-bbr 在连接开始时进入 Startup 状态，通过指数增加发送速率，迅速探测并达到瓶颈带宽。当检测到带宽增长停止时，切换到 Drain 状态。② Drain: 在 Drain 状态，BBR 通过降低发送速率，排空网络中的队列，使得数据包的数量接近 BDP（带宽时延积）。Drain 状态结束后，进入 ProbeBW 状态。③ ProbeBW: 在 ProbeBW 状态，BBR 周期性地轻微调整发送速率，探测带宽的变化。大部分时间保持在稳定发送速率，以维持高带宽利用率和低延迟。④ ProbeRTT: 在 ProbeRTT 状态，BBR 通过短时间内显著降低发送速率，探测最小 RTT。ProbeRTT 状态周期性触发，以确保 RTT 估计的准确性。

三、实验结果

任务 1：拥塞现象观察

单个数据包大小为536B，两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	23.84%	16.20%	12.46%	8.36%	3.88%	0.00%
Goodput (B/s)	9.62	12.88	17.62	25.93	142.4	221057.7
the datarate of Router to Receiver: 2.0Mbps						

图 36: 结果列表

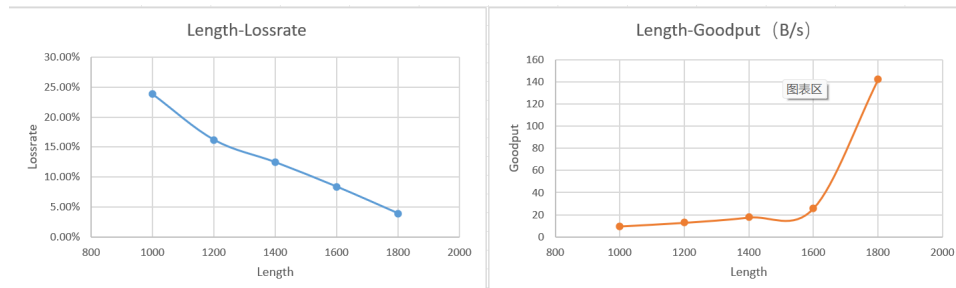
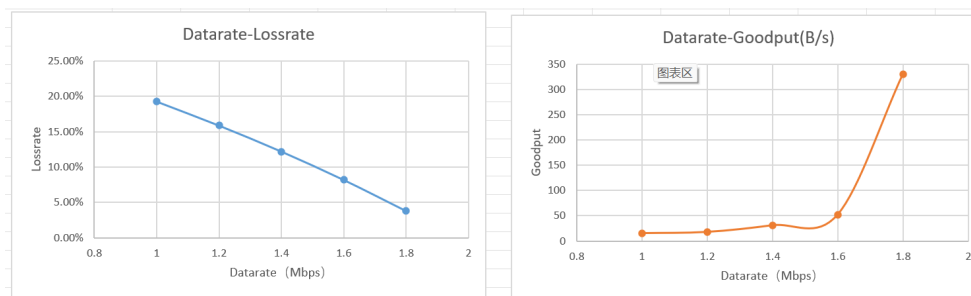


图 37: 可视化结果

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	19.28%	15.88%	12.20%	8.18%	3.78%	0.00%
Goodput (B/s)	15.26	17.66	30.87	51.94	330.72	221057.7

the length of Router: 2000

38: 结果列表



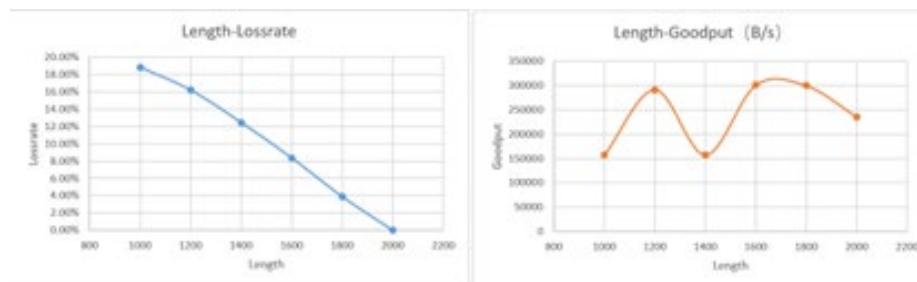
39: 可视化结果

任务 2: 实现 TCP-reno 拥塞控制算法

单个数据包大小为536B, 两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	18.84%	16.20%	12.46%	8.36%	3.88%	0.00%
Goodput (B/s)	157644.7	291018	156766.9	300992.9	299283.2	234683.3

the datarate of Router to Receiver: 20Mbps

图 40: 结果列表



41: 可视化结果

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Goodput (B/s)	114096.4	184436.7	217382.7	248158.7	264725.5	221057.7

the length of Router: 2000

42: 结果列表

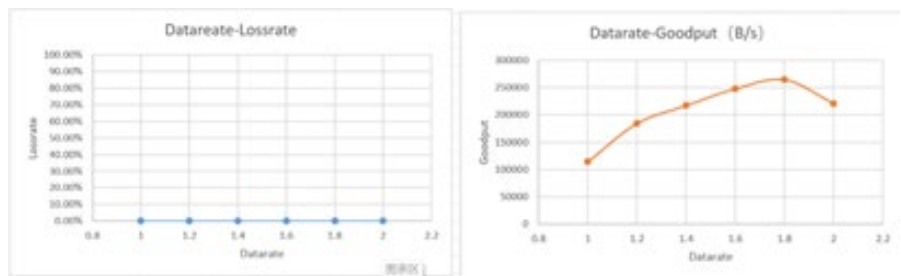
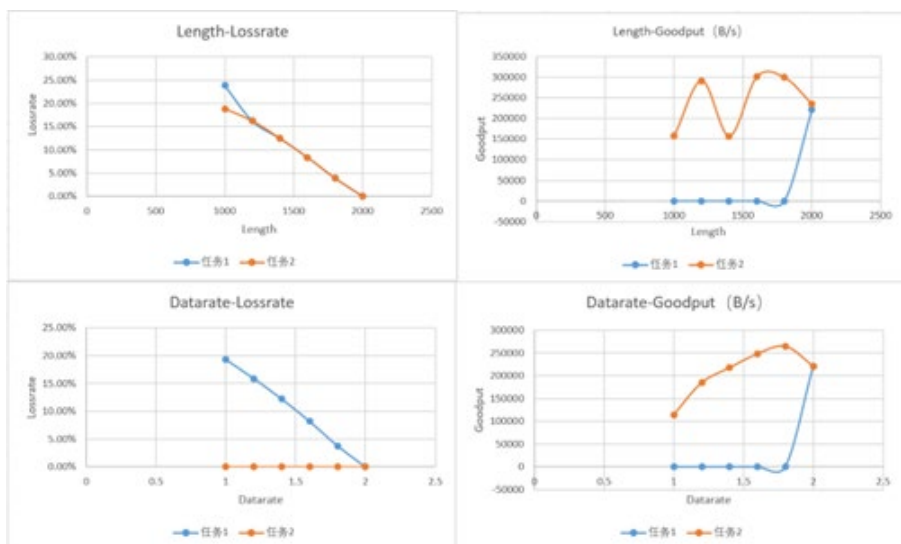


图 43: 可视化结果



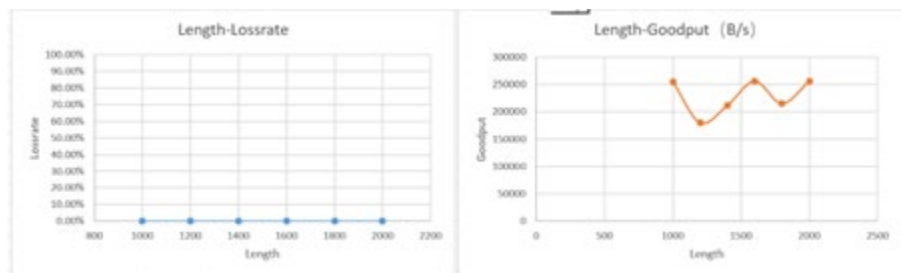
44: TCP-reno 和无拥塞控制对比

任务 3: 实现 TCP-bbr 拥塞控制算法

单个数据包大小为536B, 两个发送方一共需要发送 (1956+1) *2=3914个数据包						
length	1000	1200	1400	1600	1800	2000
Lossrate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Goodput (B/s)	255025.2	180168.1	211806.5	255672.9	215278.7	255548.5

the datarate of RoutertoReceiver: 2.0Mbps

图 45: 结果列表



46: 可视化结果

datarate (Mbps)	1	1.2	1.4	1.6	1.8	2
Lossrate	11.28%	0.00%	0.00%	7.22%	0.00%	0.00%
Goodput (B/s)	111098.2	150133.6	187432.6	158074.3	206012.4	255548.5

the length of Router: 2000

47: 结果列表

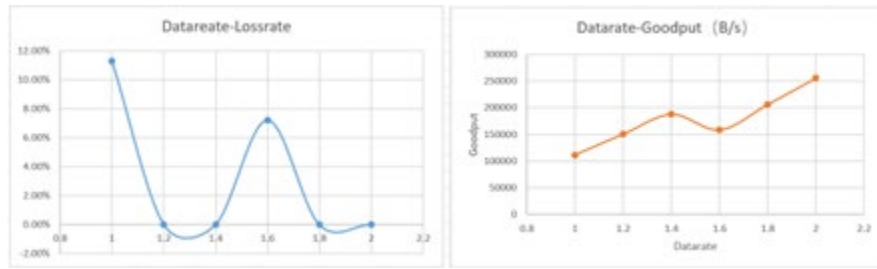
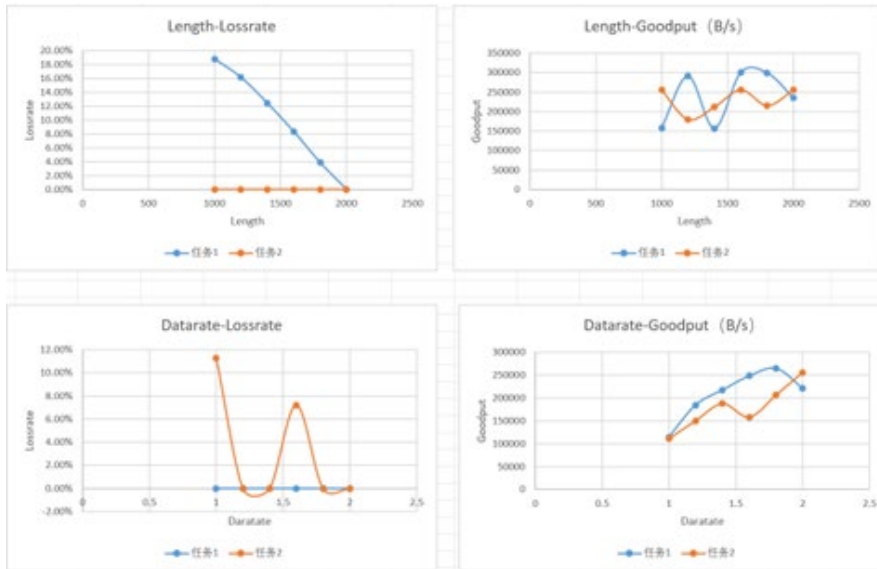


图 48：可视化结果



49：TCP-reno 和 TCP-bbr 对比

四、实验分析

1

Lossrate 随着 **datarate** 的下降而上升，呈线性上升趋势；**Goodput** 随着 **datarate** 的下降而下降，呈指数下降趋势。可能的原因：对于 **Lossrate**，链路速度决定了数据包传输的速率。较低的链路速度在单位时间内传输的数据包更少，增加了队列中的数据包的积压，提高了丢包的可能性。当链路速度下降时，数据包传输变慢，队列中的数据包等待时间变长，容易导致队列溢出（队列满）。这种情况下，新的数据包将会被丢弃。由于每单位时间内传输的数据包数量减少，导致丢包率增加。因为链路速度每减少一定比例，会相应地增加丢包的概率，所以这种趋势通常是线性的。对于 **Goodput**，**Goodput** 是成功传输的有用数据的速率，不包括重传的数据包。它反映了网络的实际有效传输性能。当链路速度下降时，传输速率降低，**Goodput** 也会相应下降。链路速度下降导致丢包率上升，网络需要更多的重传来补偿丢失的数据包。重传占用了额外的带宽，进一步降低有效数据传输速率。TCP 协议的拥塞控制机制在丢包率增加时，会显著降低发送速率，以适应链路的传输能力。由于 TCP 的拥塞控制调整速率通常是指数级的，这导致 **Goodput** 下降呈现指数趋势。

产生拥塞的原因：① **链路速度不足**。链路速度决定了数据传输的最大速率。

当流量数据超过链路的处理能力时，数据包会在路由器的队列中等待，造成拥塞。如果链路速度太低而数据太多会导致队列溢出，数据包被会丢弃，网络中重传请求增加，占用带宽，进一步加剧拥塞。② **高流量负载**。如果网络中数据量突然增加或者数据量持续高于链路传输能力，容易导致路由器队列迅速积满，导致拥塞。③ **不合理的队列长度**。如果队列设置过短，容易导致队列溢出，增加丢包率，进而引起重传，加剧网络拥塞。如果队列设置过长，可能会增加延迟。

任务 2

1. **TCP-reno 和无拥塞控制对于路由器队列长度与丢包率的关系影响相对较小**，只有在路由器长度为 1000 方有差距显现。TCP-reno 控制下和无拥塞控制下路由器队列长度与丢包率的关系均为线性关系。
2. **TCP-reno 和无拥塞控制对于路由器队列长度与有效吞吐量的关系影响较大**。由图 23 可以发现，TCP-reno 控制的有效吞吐量在测量路由器队列长度范围内均比无拥塞控制的有效吞吐量大，证明 TCP-reno 的有效性。
3. **TCP-reno 和无拥塞控制对于链路速度与丢包率的关系影响较大**。由图 23 可以发现，在 TCP-reno 控制下，链路丢包率为 0。而无拥塞控制链路丢包率随链路速度降低而升高。再次证明了 TCP-reno 算法的有效性。
4. **TCP-reno 和无拥塞控制对于链路速度与有效吞吐量的关系影响较大**。由图 23 可以发现，TCP-reno 控制的有效吞吐量在测量路由器队列长度范围内均比无拥塞控制的有效吞吐量大，证明 TCP-reno 的有效性。

任务 3

1. TCP-bbr 在丢包率方面的表现明显优于 TCP-reno，原因在于它们**最本质的区别：TCP-reno 基于丢包和延迟（超时）进行网络拥塞控制；TCP-bbr 基于带宽和 RTT 进行网络拥塞控制**。TCP-bbr 较于 TCP-reno 有如下优势：
① TCP-reno 在检测到丢包后会显著降低发送速率，导致带宽利用率降低。而 TCP-bbr 可以确保发送速率接近实际的瓶颈带宽，高效利用带宽。
② TCP-reno 在路由器队列满，而丢包的时候开始进行拥塞控制。一方面数据包排队导致高延迟，另一方面重发数据包导致网络拥塞进一步加剧。TCP-bbr 通过控制发送速率避免队列积压，使数据包以较低延迟发送到接收方。
2. 在有效吞吐量方面，两种拥塞控制算法的表现相差无几。原因在于两种算法都能比较好地以接近链路瓶颈的发送速率发送数据包。

五、实验总结

- 1 通过本次实验，我了解了数据传输过程中出现拥塞的原因。
- 2 通过本次实验，我了解了拥塞现象在网络中的表现。
- 3 通过本次实验，我了解了 TCP-reno 和 TCP-bbr 两种拥塞控制算法各自的工作原理。
- 4 通过本次实验，我了解了 TCP-reno 和 TCP-bbr 两种拥塞控制算法的区别以及它们各自的优势。

六、 思考题

在什么情况下 TCP-Reno 比 TCP-BBR 表现更好，反之亦然？请设计实验场景来验证你的假设，并解释其原因。要求：设计两种网络拓扑结构，分别运行 TCP-Reno 和 TCP-BBR，并记录丢包率（LossRate）和有效吞吐量（GoodPut）。分析并解释实验结果，讨论两种算法在不同网络条件下的优势和劣势。

TCP-reno 表现更好的情况：① **低延迟、高带宽的网络**。在这种环境下，TCP-Reno 能够快速恢复从丢包中，并且其拥塞窗口的增长方式可以迅速利用高带宽。由于网络延迟低，TCP-Reno 的基于丢包的拥塞控制机制不会带来明显的传输性能问题。② **稳定的网络环境**。在稳定的网络环境下，TCP-Reno 的丢包检测和窗口调整机制能够有效维持高吞吐量，并且由于网络的稳定性，TCP-Reno 的拥塞控制算法能保持较好的性能。

TCP-bbr 表现更好的情况：① **高延迟、低带宽的网络**。TCP-bbr 通过主动测量带宽和 RTT 来调节发送速率，不依赖丢包作为拥塞信号，能够更好地适应高延迟和低带宽环境。TCP-bbr 可以避免由于高 RTT 导致的慢启动和拥塞窗口收缩。② **动态变化的网络环境**。TCP-bbr 在动态网络环境下能够通过实时测量带宽和 RTT 来调整发送速率，提供稳定的传输性能。它的带宽探测机制使其在网络条件变化时依然能维持较高的吞吐量。

为了验证假设，我们设计两种网络拓补结构，两种均由两个客户端、一个路由器（队列长度为 1800）以及一个服务端构成。这两种网络分别为低延迟、高带宽网络（20ms、1.6Mbps）以及高延迟、低带宽网络（200ms、1.0Mbps）。

1. 运行模拟器，实验结果见下图。

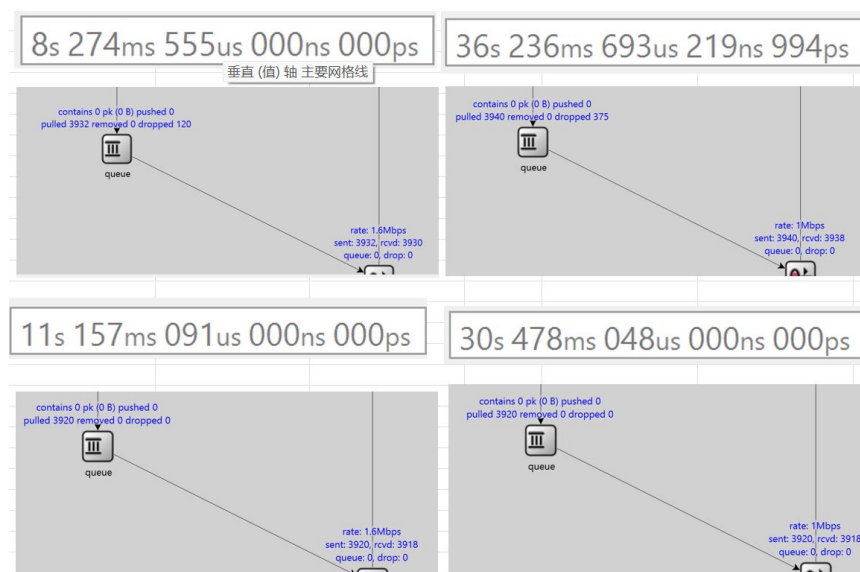


图 50：实验结果（左低延迟、高带宽网络，右高延迟、低带宽）
实验结果列表。

Goodput (B/s)	2ms、1.8Mbps	200ms、1.0Mbps
TCP-reno	254719.84	58280.16
TCP-bbr	188323.02	68938.9

图 51a：Gooput

Lossrate	2ms、1.8Mbps	200ms、1.0Mbps
TCP-reno	2.96%	8.69%
TCP-bbr	0.00%	0.00%

图 52b: Lossrate

实验结果可视化。

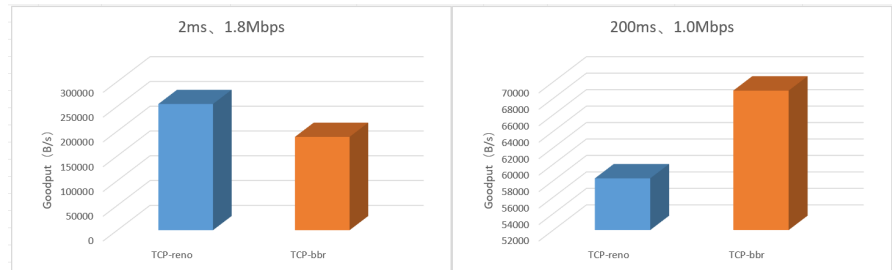


图 51a: Gooput

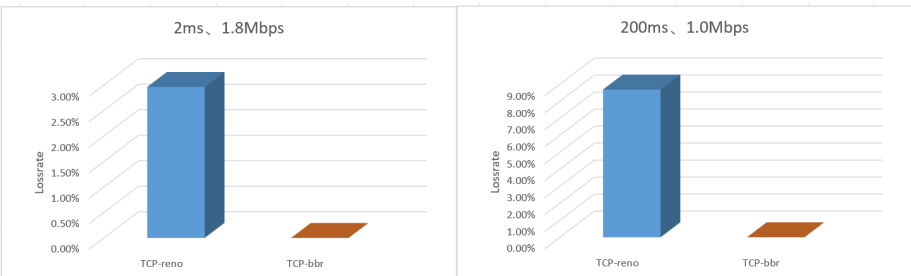


图 52b: Lossrate

实验结果分析。

由上展示的实验结果可知，实验假设正确。TCP-reno 在低延迟、高带宽的网络下表现更好，TCP-bbr 在高延迟、低带宽的网络下表现更好。

TCP-reno 是基于丢包和时延的算法，在低延迟、高带宽的网络中始终能以接近带宽的发送速率进行发送活动；TCP-bbr 在低延迟、高带宽的网络中，StartUP 和 Drain 两种状态可能需要消耗较多的时间，在这一段时间里无法有效利用带宽。

TCP-reno 在高延迟、低带宽的网络中会发生多次重传现象，导致网络拥塞进一步加重，丢包数量增加；TCP-bbr 在高延迟、低带宽的网络中则能比较好地以网络最大带宽的发送速率发送数据包。

指导教师批阅意见

成绩评定

指导教师签字：

年 月 日

- 注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。