南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

UDP 可靠传输协议——part4 性能测试

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：张建忠 徐敬东

2024 年 12 月 10 日

摘要

本次实验借助 UDP 协议，设计了报文类，参考 rdt3.0 的协议思想，设计了三次握手，四次挥手和可靠数据传输协议。在先前的实验中，分别实现了停等机制和滑动窗口机制，并实现了拥塞控制 Reno 算法。在本次实验中，我将利用 router 对协议进行性能测试，并进行分析。

关键词：UDP，可靠数据传输，rdt3.0，router，超时和校验和处理

目录

一、 实验要求

1. 作业要求：基于给定的实验测试环境，通过改变延时和丢包率，完成下面 3 组性能对比实验：（1）停等机制与滑动窗口机制性能对比；（2）滑动窗口机制中不同窗口大小对性能的影响；（3）有拥塞控制和无拥塞控制的性能比较。
2. 控制变量法：对比时要控制单一变量（算法、窗口大小、延时、丢包率）
3. Router：可能会有较大延时，传输速率不作为评分依据，也可自行设计
4. 延时、丢包率对比设置：要有梯度（例如 30ms,50ms, …；5%，10%，…）
5. 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）
6. 性能测试指标：时延、吞吐率，要给出图、表并进行分析

二、 实验环境

本次实验在 x86-64 架构物理机中进行。通过 VS Code 完成实验代码的编写，通过 G++ 编译器完成源代码的编译。本实验的程序对环境没有严苛的要求，理论上来说现代 x86 架构都可以成功运行。

文件结构如图1所示。

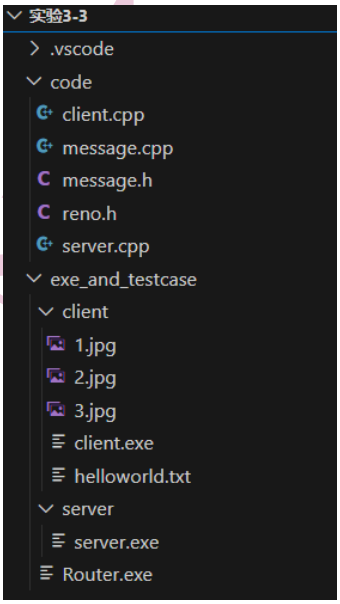

图 1: 文件结构

此处为测试准备好了样例，当测试通过时，收到的文件应出现在 server 目录下。

三、 实验设计回顾

总体上来说，无论是客户端还是服务器端，SEQ 的更新逻辑都是上一报文的 SEQ+ 上一报文的 LEN，ACK 的更新逻辑是上一次接收到的正确报文的 SEQ+LEN。

对于 SYN 或 FIN 置位的报文来说，情况略有不同。为体现出三次握手，四次挥手的过程，保证实验结果的可靠性（现实中也是这么实现的），这两类报文的 LEN 虽然是 0，但对应的 SEQ/ACK 需要自增 1。（也可以理解为，这两类报文的 LEN 可以视为 1）。

(一) 报文设计

报文设计需要包含以下元素：（由于本次实验为回环传输，没有设计 IP 地址元素）

- 源端口 & 目的端口
- SEQ & ACK
- 标志位 FLAGS
- 报文体长度
- 校验和
- 报文体内容

(二) 三次握手

三次握手的实现如下：

注：在握手过程中，客户端生成随机数 R1，服务器端生成随机数 R2。

1. 第一次握手：

   客户端-> 服务器端：SEQ=R1，ACK=0，FLAGS=SYN。

2. 第二次握手：

   服务器端-> 客户端：SEQ=R2，ACK=R1+1，FLAGS=SYN | ACK。

3. 第三次握手：

   客户端-> 服务器端：SEQ=R1+1，ACK=R2+1，FLAGS=ACK。

随后，客户端在发送第三次握手后建立连接，服务器端在接收第三次握手后建立连接。

在三次握手实现中，同样增加了超时重传机制，保证可靠性。

(三) 数据传输

在这一部分中，报文的 SEQ/ACK 设计遵循本节开头的原则。（由于单向传输，客户端的 ACK 和服务器端的 SEQ 并没有真正发生改变过，但客户端 SEQ 和服务器端 ACK 的对应关系已经足以进行报文的区分。）对于标记位置位，文件发送不进行 ACK 置位，文件接收进行 ACK 置位。

1. 客户端核心逻辑

对于客户端，需要将要发送的文件分包发送。

首先要发送一个包括文件名和文件长度的包，以便服务器端开辟合适的接收缓存。

随后每次尽可能将报文体填满，即除最后一个包外，每个包的大小都应该是报文体的最大大小，最后一个包发送剩余的所有内容。

此外，客户端还需要在发送前设计文件读取。即将文件全部内容读取到 buffer 中，以便后续发送。

2. 服务器端核心逻辑

对于服务器端，需要接收客户端发送的内容，并返回 ACK。

对于文件名和文件长度的包，进行合理的拆分，分别保存文件名和文件长度，并动态分配 buffer 大小。

在后续包的接收时，把客户端发送的内容保存到 buffer 中。

在客户端发送完毕后，将 buffer 中的内容写入文件。

3. 停等机制：超时重传和校验

在停等机制下，无论是超时重传，还是校验错误，都可以通过重传上一个包来解决。

因为该情况的原因，要么是发过去的包丢了或者错了，要么是对方收到后，返回的包丢了或者错了。

在第一种情况下，对上一个包的重传可以很显然的解决问题，在第二种情况下，通过重传上一个包可以提示对方传输失败，使得对方进行上一个包的重传，从而解决问题。

4. 滑动窗口机制：累计确认

在滑动窗口机制下，在发送端采用了滑动窗口累计确认的机制。具体细节为：

发送线程：

在当前窗口中若有未发送报文（在我的设计中，还要求此时不涉及重传触发，因为如果需要重传的话这些新发过去的包肯定不会被接收），则持续进行新报文的发送，直到窗口中所有报文均已经被发送。

在触发超时重传或三次错误重传时（即收到三次相同的验证通过的 ACK 报文），需要将窗口中已经发送的报文重新发送，以期待接收端更新 ACK。

接收线程：

当接收到新的 ACK 时，持续更新窗口，实现累计确认（即确认所有 SEQ+LEN 小于收到 ACK 的包）。

当收到三次相同的验证通过的 ACK 报文时，或距离上次发送报文已经超时时（判定方式为：若窗口得到更新，或置位了重传信号，都对计时器进行更新），置位重传信号，提示发送线程需要对已发送的包进行一次重传。

直到所有文件都已经被正确接收，提供接收结束信号，线程结束的同时提示发送线程退出循环。

5. Reno 拥塞控制

Reno 算法的状态机如图2所示。

Reno 算法共具有三个状态：慢启动，拥塞避免，快速恢复。

收发报文的过程中共有三个过程：收到正确的新 ACK，收到重复 ACK，以及超时。

引入这样两个概念：

- CWND：拥塞窗口大小
- MSS：最大报文段
- SSTHRESH：慢启动门限值，用于判定什么时候离开慢启动。

初始状态：CWND = 1MSS

1. 在当前状态为慢启动时：
   - 若收到新 ACK,则 CWND+=MSS,清空重复 ACK 计数,若 CWND 超过了 SSTHRESH 则切换到拥塞避免状态。
   - 若收到重复 ACK，则增加重复 ACK 计数。
   - 若超时，则置 SSTHRESH 为 CWND/2，CWND 为 1，重复 ACK 计数为 0，重新开始慢启动。
2. 在当前状态为拥塞避免时：
   - 若收到新的 ACK，每收到 CWND 个，将 CWND+1，并清空重复 ACK 计数。
   - 若收到重复 ACK, 则增加重复 ACK 计数。当重复 ACK 达到 3 个的时候, SSTHRESH 置为 CWND/2，CWND 置为 SSTHRESH + 3，切换到快速恢复阶段。
   - 若超时，同慢启动超时处理。
3. 在当前状态为快速恢复时：
   - 若收到新的 ACK，则置 CWND 为 SSTHRESH，清空重复 ACK 计数，回到拥塞避免状态。
   - 若收到重复 ACK，使 CWND +=1。
   - 若超时，同慢启动超时处理。

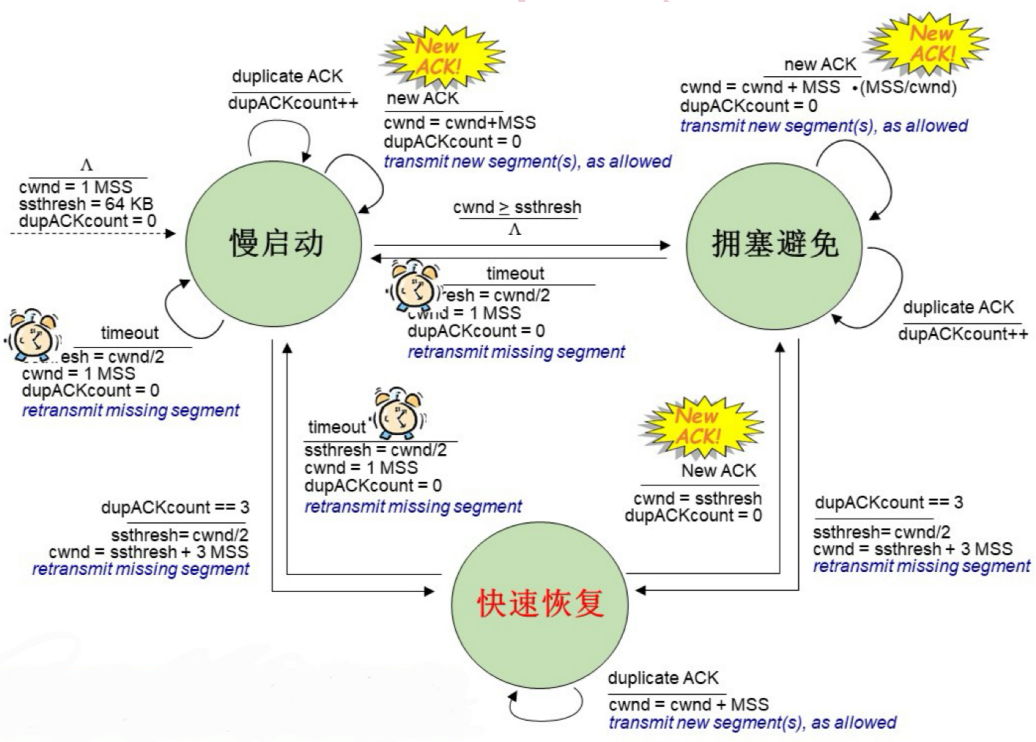

图 2: Reno 算法状态机

（四） 四次挥手

四次挥手的实现如下：

注：下文中的 L1 表示在数据传输阶段，客户端发送的最后一个报文的 SEQ+LEN；L2 表示在数据传输阶段，服务器端发送的最后一个报文的 SEQ+LEN。

1. 第一次挥手：

   客户端-> 服务器端：SEQ=L1，ACK=L2，FLAGS=FIN | ACK。

2. 第二次挥手：

   服务器端-> 客户端：SEQ=L2，ACK=L1+1，FLAGS=ACK。

3. 第三次挥手：

   服务器端-> 客户端：SEQ=L2，ACK=L1+1，FLAGS=FIN | ACK。

4. 第四次握手：

   客户端-> 服务器端：SEQ=L1+1，ACK=L2+1，FLAGS=ACK。

随后，客户端在第四次挥手发送后等待 2RTT 断开连接，服务器端在收到第四次挥手后断开连接。

在四次挥手实现中，同样增加了超时重传机制，保证可靠性。

四、 router 设计

由于先前的 router 在我的电脑上总出现奇奇怪怪的问题，我使用 python 重新设计了 router。

生成可执行文件

```python
import socket
import time
import random

# 地址设置
SERVER_ADDRESS = ('127.0.0.1', 2222)
ROUTER_ADDRESS = ('127.0.0.1', 3333)
CLIENT_ADDRESS = ('127.0.0.1', 4444) #直接给出了CLIENT地址，简化实现

DROP_RATE = 0.05  # 丢包率
DELAY = 0.005  # 时延(s)

def should_drop(): # 这里我使用random函数，比原有router更贴近现实场景
    return random.random() < DROP_RATE

def router():
    # 创建 UDP 套接字
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
        sock.bind(ROUTER_ADDRESS)
        print(f"Router启动成功，地址： {ROUTER_ADDRESS}\nRouter的丢包率为{
            DROP_RATE}，传输时延为{DELAY}。")

        while True:
            # 接收数据
```

```
            data, addr = sock.recvfrom(100000)

            # 判断数据包来源
            if addr == SERVER_ADDRESS:
                # 来自 server 的包
                print(f"收到了来自server的包，大小 {len(data)} 字节。")
                sock.sendto(data, CLIENT_ADDRESS)

            elif addr == CLIENT_ADDRESS:
                # 来自 client 的包
                print(f"收到了来自client的包，大小 {len(data)} 字节。")

                if should_drop():
                    print("丢包。")
                    continue  # 丢包

                print("延迟发送。")
                time.sleep(DELAY)
                sock.sendto(data, SERVER_ADDRESS)
                continue

if __name__ == "__main__":
    try:
        router()
    except KeyboardInterrupt:
        print("\nRouter stopped.")
```

五、 性能测试

(一) 测试说明

在本次实验中，所有程序都尽可能的做到了控制变量。先前 3-1 提交的报文输出格式与 3-2, 3-3 内容量相差较大，在本次测试中已经进行了统一。

对于每一个测试点，均进行多次测试（3 次或 5 次），以取得平均值，避免偶然误差。

在测试中，使用固定超时时间为 2s。

(二) 测试环境配置

1. IP 和端口号设置

1. 服务器端 port：2222
2. Router 端 port：3333
3. 客户端 port：4444

虽然服务器端设置了使用全部 IP 地址，但在后续测试中都使用 127.0.0.1，Router 和客户端则是确定了 IP 地址为 127.0.0.1。

2. 可执行文件生成

生成可执行文件的指令如下:

生成可执行文件

```
生成可执行文件的指令如下:
//服务器端
g++ code/server.cpp code/message.cpp -o exe_and_testcase/server/server -
    lws2_32
//客户端
g++ code/client.cpp code/message.cpp -o exe_and_testcase/client/client -
    lws2_32
//router
pyinstaller --onefile router.py
//python也可以不生成可执行文件直接运行
python router.py
```

(三) 停等机制与滑动窗口机制性能对比

在本部分中，测试组共分为:

- 停等机制：实验 3-1 实现的程序。
- 滑动窗口机制：实验 3-2 实现的程序，窗口大小设为 30。

1. 不同丢包率下的性能对比

延时为 0ms 下，测试不同丢包率下的性能，如表1所示:

丢包率		0%	2%	4%	6%	8%	10%
停等	传输时间 (s)	1.121	7.117	9.134	15.118	23.161	25.122
	吞吐量 (KB/s)	1657	261.0	203.3	122.9	80.19	73.93
滑动窗口	传输时间 (s)	0.641	0.729	0.752	0.856	0.948	1.265
	吞吐量 (KB/s)	2898	2547	2470	2170	1959	1468

表 1: 不同丢包率下的性能对比

将对比结果绘制成图片，如图3所示。

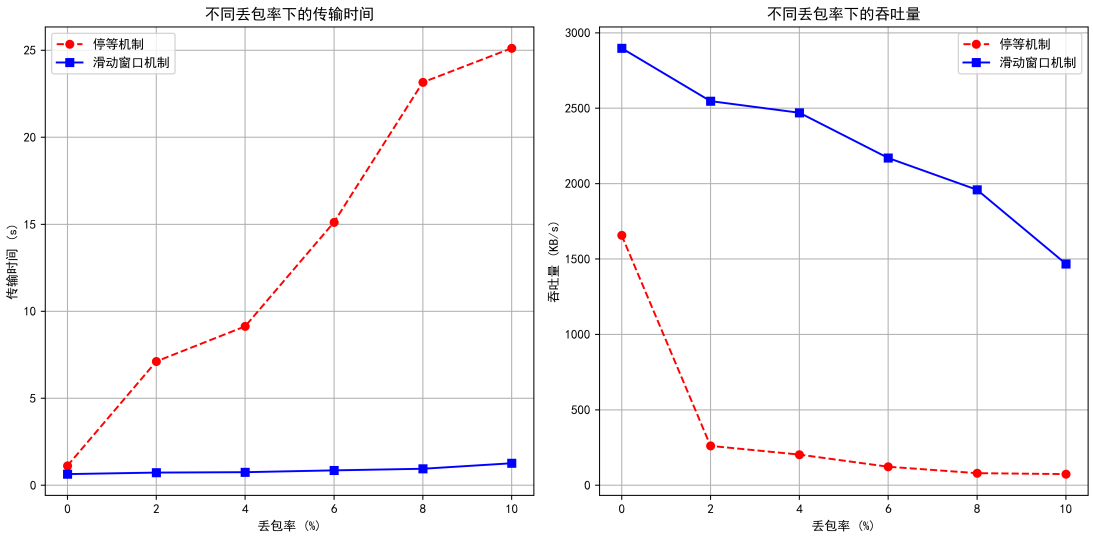

图 3: 不同丢包率下的性能对比

注：停等机制下，不同样本测试结果波动较大，即使已经采用了五次取平均值的方式，曲线仍然不够平滑。

分析：在不同丢包率下的性能对比可以看出，滑动窗口机制全面领先于停等机制。在丢包率为 0% 时，两方法差距并不明显。然而随着丢包率的增大，停等机制产生了非常大的延时。然而滑动窗口机制随丢包率增大延时并不明显。这主要是因为，在滑动窗口机制下，下一个包的发送不需要等待上一个包的接收，且收到三个重复 ACK 即可触发快速重传，对于一次丢失，需要多发送的包可能仅有五个左右。相比之下，停等机制需要等待超时重传，每一次包的丢失都会造成超过我设置的超时时间 2s 以上的延时，这样的开销是非常大的。

2. 不同延时下的性能对比

丢包率为 0% 下，测试不同延时下的性能，如表2所示：

传输延时 (ms)		0	20	40	60	80	100
停等	传输时间 (s)	1.121	4.286	8.430	10.611	11.474	14.014
	吞吐量 (KB/s)	1657	433.4	220.3	175.0	161.8	132.5
滑动窗口	传输时间 (s)	0.641	18.304	21.234	25.545	31.368	34.461
	吞吐量 (KB/s)	2898	101.5	87.47	72.70	59.21	53.89

表 2: 不同延时下的性能对比

将对比结果绘制成图片，如图4所示。

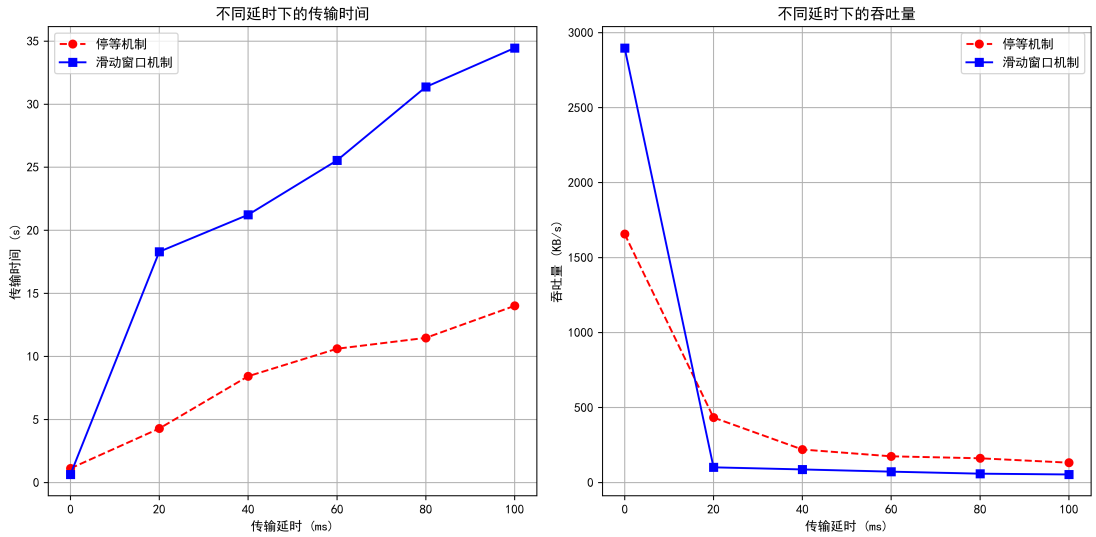

图 4: 不同延时下的性能对比

分析: 可以看到，随着传输延时增加，停等机制和滑动窗口机制的性能都有较大幅度的下降。然而，滑动窗口机制的性能下降更加显著，除传输延时为 0 时的测试结果之外，滑动窗口机制的测试结果明显劣于停等机制。分析报文可以发现，滑动窗口机制产生了大量的重传报文。例如：当窗口大小为 30 时，若第 2 个报文在发送时丢失，且 30 个报文已经发送完毕。（由于我们的设置，实际上延时是比传输速度慢很多的，尤其是在传输延时达到 100ms 下，该情况很容易发生。虽然理论上的丢包率为 0%，但由于网络拥塞，实际上发生的丢包情况可能远高于设置。）这种情况下后面 29 个报文都需要进行重传，这对性能造成了较大的影响。

(四) 滑动窗口机制中不同窗口大小对性能的影响

在本部分中，测试组共分 4 组，使用实验 3-2 实现的程序，在窗口大小为 5，10，20，30 的情况下进行测试。

1. 不同丢包率下的性能对比

延时为 0ms 下，测试不同丢包率下的性能，如表3所示：

丢包率		0%	2%	4%	6%	8%	10%
size=5	传输时间 (s)	0.620	0.684	0.740	0.800	0.850	1.010
	吞吐量 (KB/s)	2996	2715	2510	2322	2185	1839
size=10	传输时间 (s)	0.610	0.670	0.760	0.846	0.850	0.970
	吞吐量 (KB/s)	3045	2772	2444	2195	2185	1915
size=20	传输时间 (s)	0.620	0.680	0.720	0.760	0.970	1.001
	吞吐量 (KB/s)	2996	2731	2580	2444	1915	1856
size=30	传输时间 (s)	0.641	0.729	0.752	0.856	0.948	1.265
	吞吐量 (KB/s)	2898	2547	2470	2170	1959	1468

表 3: 不同丢包率下不同滑动窗口大小性能对比

将对比结果绘制成图片，如图5所示。

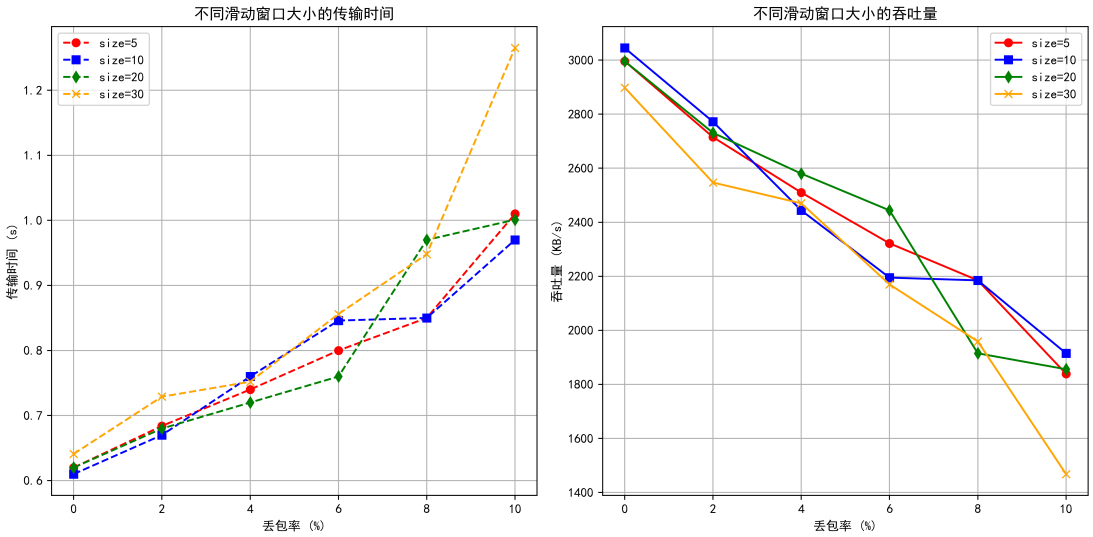

图 5: 不同丢包率下不同滑动窗口性能对比

分析：总体上，随着丢包率的升高，也产生了更多的重传，不过根据滑动窗口的实现机制，若单个包丢失，接收到三个重复 ACK 后，就可以对包发生重传，一次包的丢失理论上仅仅会产生一次对窗口内已发送报文的快速重传。因此，随着丢包率的提升，传输时间也相应的提升，吞吐量也相应下降，不过趋势并不显著，且窗口大小对性能的影响并不大。不过窗口大小为 30 的测试组整体上来说相比于其他组性能略低，原因是窗口过大导致了更多的重传。

2. 不同延时下的性能对比

丢包率为 0% 下，测试不同延时下的性能，如表4所示:

传输延时 (ms)		0	20	40	60	80	100
size=5	传输时间 (s)	0.620	3.431	6.072	8.723	11.157	14.559
	吞吐量 (KB/s)	2996	541.3	305.9	212.9	166.5	127.6
size=10	传输时间 (s)	0.610	9.183	12.263	18.673	23.806	28.786
	吞吐量 (KB/s)	3045	202.2	151.4	99.47	78.02	64.52
size=20	传输时间 (s)	0.620	9.652	13.383	22.417	31.014	33.567
	吞吐量 (KB/s)	2996	192.4	138.8	82.85	59.89	55.33
size=30	传输时间 (s)	0.641	18.304	21.234	25.545	31.368	34.461
	吞吐量 (KB/s)	2898	101.5	87.47	72.70	59.21	53.89

表 4: 不同延时下不同滑动窗口大小性能对比

将对比结果绘制成图片，如图6所示。（增加了与停等机制的对比）

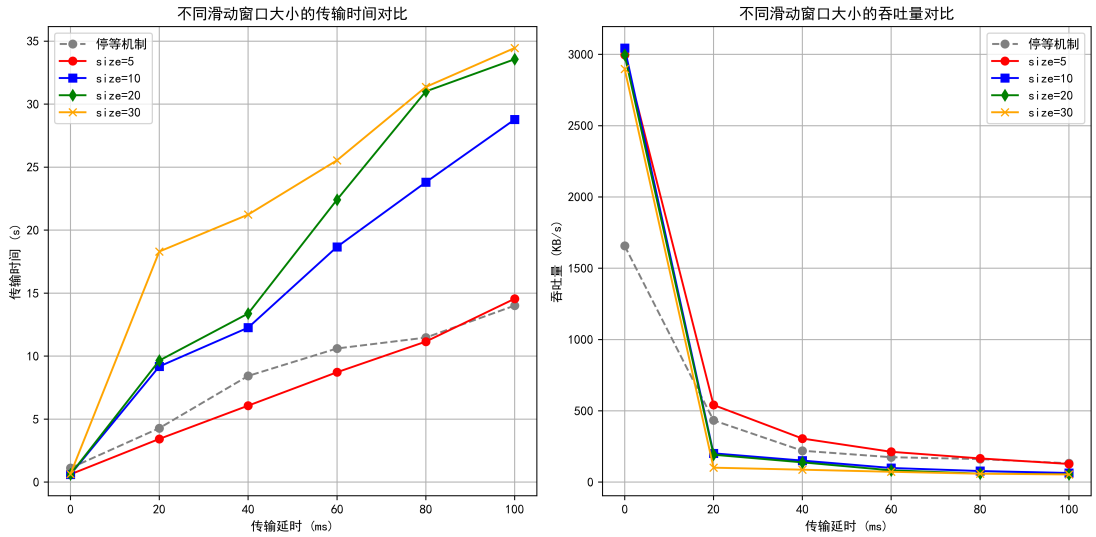

图 6: 不同延时下不同滑动窗口大小性能对比

分析: 随传输延时增加，各测试组的性能都有了较大幅度的下降。可能是由于本机性能较低，或我的 router 设计还有待改进，容易受到网络堵塞的影响，仅仅窗口大小为 5 的测试组取得了比停等机制更好的测试结果，其他测试组都产生了过高的传输时延，且随着滑动窗口大小增大，传输时间也随之提升。因此，根据不同的网络环境与带宽，选择适宜的窗口大小，不盲目追求过大的窗口，是很有必要的。

(五) 有拥塞控制和无拥塞控制的性能比较

在本部分中，测试组共分为:

- 无拥塞控制：实验 3-2 实现的程序。
- 有拥塞控制：Reno 拥塞控制算法，实验 3-3 实现的程序。

窗口大小均设为 30。

1. 不同丢包率下的性能对比

延时为 0ms 下，测试不同丢包率下的性能，如表5所示:

丢包率		0%	2%	4%	6%	8%	10%
有拥塞控制	传输时间 (s)	0.708	0.745	0.817	0.889	2.930	9.034
	吞吐量 (KB/s)	2623	2493	2273	2089	633.9	205.6
无拥塞控制	传输时间 (s)	0.641	0.729	0.752	0.856	0.948	1.265
	吞吐量 (KB/s)	2898	2547	2470	2170	1959	1468

表 5: 不同丢包率下的性能对比

将对比结果绘制成图片，如图7所示。

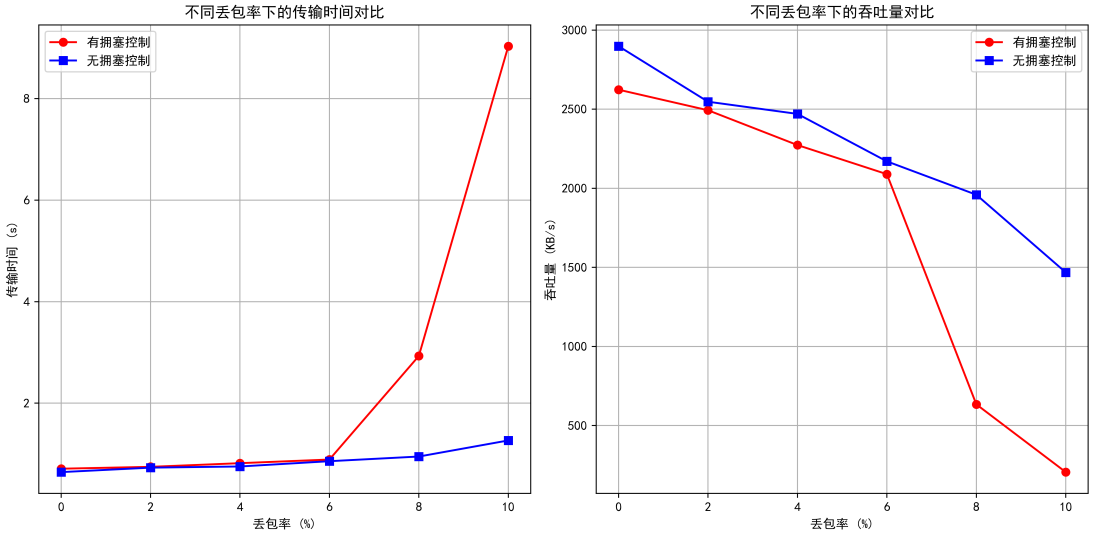

图 7: 不同丢包率下的性能对比

分析：随丢包率增大，各方法均出现了传输时间增加，吞吐量下降的情况。然而，在丢包率为 6% 以下，有拥塞控制的方法与无拥塞控制的方法差距并不显著。但当丢包率提升后，由于频繁发生的丢包情况，拥塞窗口大小一直位于较低水平，若在窗口大小小于 3 的时候发生了丢包，则会造成超时，返回慢启动阶段，因此产生了比较大的开销。

总体来说，由于无拥塞控制的性能已经很高了，有拥塞控制不但没有提升性能，反而产生了性能的降低。

2. 不同延时下的性能对比

丢包率为 0% 下，测试不同延时下的性能，如表6所示：

传输延时 (ms)		0	20	40	60	80	100
有拥塞控制	传输时间 (s)	0.708	10.407	17.931	19.955	22.971	25.996
	吞吐量 (KB/s)	2623	178.4	103.5	93.08	80.86	71.45
无拥塞控制	传输时间 (s)	0.641	18.304	21.234	25.545	31.368	34.461
	吞吐量 (KB/s)	2898	101.5	87.47	72.70	59.21	53.89

表 6: 不同延时下的性能对比

将对比结果绘制成图片，如图8所示。

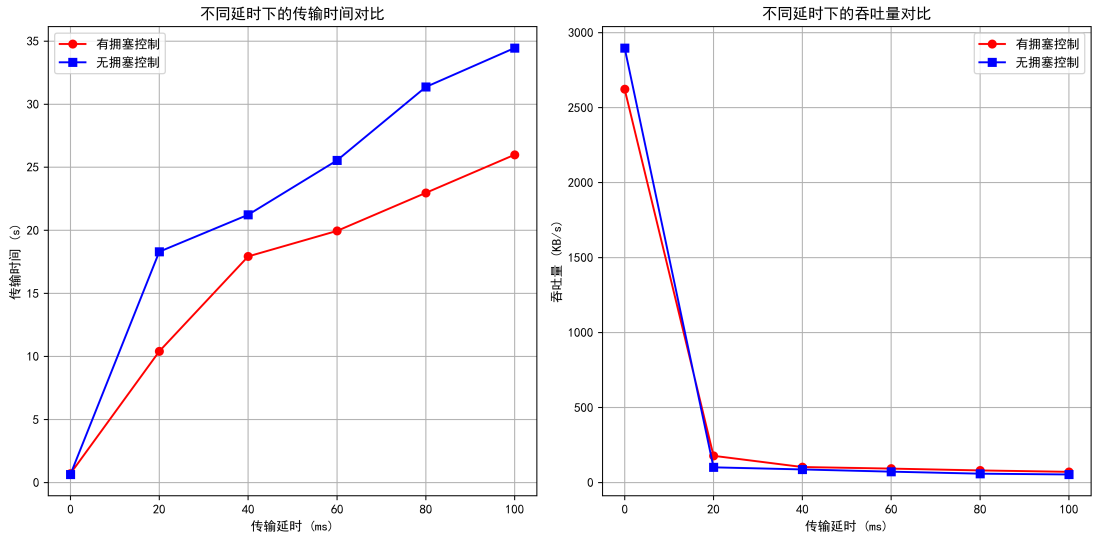

图 8: 不同延时下的性能对比

分析：随传输延时的增加，有无拥塞控制的传输时间都发生了较为显著的增加。然而，相比于有拥塞控制的测试结果，无拥塞控制的测试组的优化随传输延时的增加而越发显著。这是因为，在发生拥塞时（重复 ACK 或超时），Reno 算法会变换状态，改变窗口大小，从而减少不必要的包的发送，进而成功起到提升性能的效果。

六、 总结

在本次实验中，我综合前三次子实验中设计的程序，进行性能测试与对比。

主要针对：停等 (3-1) 与滑动窗口 (3-2) 机制的性能对比，滑动窗口机制 (3-2) 中不同窗口大小的性能对比，Reno 拥塞控制 (3-3) 下与无拥塞控制 (3-2) 的性能对比进行研究。通过分析性能测试结果，更进一步的理解了各协议的优劣势。

在本次测试中通过本人物理机进行性能测试。通过测试结果可以看出：在不同的环境背景下，由于网络传输性能的不同（在本实验中主要考虑了丢包率和传输延时，但在实际环境中或许要考虑的问题会更多），最优的算法也会有差异。

在整个 lab3 的实现过程中，我对 UDP 协议，停等机制与滑动窗口机制，以及 RENO 拥塞控制算法有了更进一步的理解与感悟，与理论课内容结合得到了更进一步的理解。