南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

Lab3-4：基于 UDP 服务设计可靠传输协议并编程实现

冯思程 2112213

年级：2021 级

专业：计算机科学与技术

指导教师：吴英、文静静

2023 年 12 月 23 日

摘要

基于给定的实验测试环境，通过改变延时和丢包率，完成了下面 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 (累计确认和选择认两种情形) (3) 滑动窗口机制中相同窗口大小情况下，累计确认和选择确认的性能比较。

关键字：停等机制、滑动窗口、累计确认、选择确认

目录

一、 预备工作及实验环境

(一) 实验要求与功能

实验要求的基础功能与要求:

基于给定的实验测试环境，通过改变延时和丢包率，完成了下面 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 (累计确认和选择确认两种情形)
3. 滑动窗口机制中相同窗口大小情况下，累计确认和选择确认的性能比较。

合理的自行扩展功能:

1. 由于给定的 MFC 程序 router 延时不稳定，会使误差较大，这里我自行编写了一个基于 python 的 router 用于本次实验的性能测试。

(二) 实验环境与说明

具体的实验环境配置如下:

Windows 版本	vs code 版本	docker 版本	g++ 版本
windows11	1.82.0	24.0.6	8.1.0

表 1: 实验环境说明表

前三次实验的代码文件均使用 GBK 编码，以支持中文字符。前三次 makefile 文件使用正常的 UTF-8 编码。

感谢老师与助教的审查批阅与指正，辛苦!

二、 实验过程

(一) 协议设计

在之前的实验中，已经完成了三种协议的实现。下面简单地进行一下回顾与总结:

1. 报文格式

这部分的设计一直贯穿 3-1 到 3-3 实验，如下：报文分为两个部分：报文头和报文段。

报文头 报文头一共包括源 IP（4 字节)、目的 IP（4 字节)、源 PORT（2 字节)、目的 PORT (2 字节)、序列号（4 字节)、确认号（4 字节)、发送文件大小（4 字节)、标志（2 字节)、校验和（2 字节)。一共 28 字节。

报文段 这里我设计的报文段就是装载数据的数据段。这里最大字节数我设置为 10000 字节，因为 router 转发的数据包最大是 15000 字节。

源 IP	
目的 IP	
源 PORT	目的 PORT
序列号	
确认号	
发送文件大小	
标志	校验和
报文段	

表 2: 报文格式

标志位 报文头的标志字段可以从下面设计的四种选择若干种进行加到 flag 上从而实现标志设置，其中包括 TCP 协议中用到的 SYN、ACK、FIN，还有一个我自己设计的 SFileName 标志，这个是用来表示传输文件的具体数据前发送文件名和文件大小的那条报文的，代码如下：

标志位

```
//设置不同的标志位，用到的包括SYN、ACK、FIN，SFileName表示传输了文件名字
const unsigned short SYN = 0x1;//0001
const unsigned short ACK = 0x2;//0010
const unsigned short FIN = 0x4;//0100
const unsigned short SFileName = 0x8;//1000
```

2. 建立连接：三次握手（包含超时重传）

三次握手的设计也是贯穿了 3-1 到 3-3 实验，如下：三次握手基于 TCP 协议进行设计的，实现的示意图如下：

图 1: 三次握手示意图

具体的流程为了节省篇幅我不在本次实验中赘述了。

三次握手的超时重传机制 这里我设计了基于 rdt3.0 协议的超时重传机制，下面分别是 client 和 server 的重传机制实现流程：

client：

1. 发送第一次握手，在发送后接着就开始计时
2. 接收 server 发来的第二次握手，在这个不断等待接收的过程中，如果第一次握手超时，则会重传第一次握手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。
3. 发送第三次握手，这是一个 ACK 报文，不需要进行重传。发送后 client 成功建立连接。

server：

1. 接收第一次握手
2. 接收到第一次握手后，发送第二次握手消息并在发送后接着进行计时。
3. 接收 client 的第三次握手，在这个不断等待接收的过程中，如果第二次握手超时，则会重传第二次握手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。接收第三次握手成功后 server 成功建立连接，现在双方可以开始进行数据传输了。

3. 不可靠信道上的可靠数据传输：三次实验的不同协议

本次实验中从 client 向 server 发送文件，文件较大，需要拆分成多个数据报文进行传输，一次完整的文件传输的数据报文一共分为三类，如下：（该设计贯穿前三次实验）

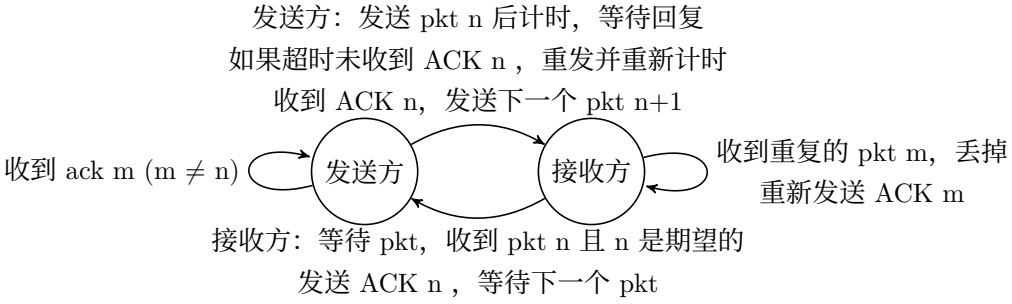
1. 第一类数据报文是装载文件名字和文件大小的数据报文。
2. 第二类数据报文是满载数据报文，这类报文传输的是具体的文件数据，满载就是这个报文的报文段可以装满。
3. 第三类数据报文是未满载数据报文，这类报文只能是 0 个或者 1 个，装载的数据大小是文件大小除以报文段最大装载大小的余数。

停等机制 由于之前的实验中已经对协议设计进行了详尽的说明和分析，所以这里只是简短的回顾一下，如下：

数据传输协议设计我基于 rdt3.0 协议进行改进，沿用上文中三次握手连接中由 client 和 server 双端各自维护序列号进行递增，在正常状态下，client 发送一个序列号为 k 的数据报文，接收端会回复一个确认号为 k 的 ACK 报文（client 会对来自 server 的 ACK 报文进行校验和检查确保数据报文传输无误），同时这里依旧沿用三次握手中 ACK 报文也会占用序列号的设计，server 发送的 ACK 报文的序列号是在 server 维护的序列号上每次自动递增 1 进行计算的。

同时引入了超时重传机制，这里之前设计中已经详细分析过，这里不进行赘述。

展示数据传输过程中的 client 端和 server 端的状态机，如下：



GBN 协议 由于之前的实验中已经对协议设计进行了详尽的说明和分析，所以这里只是简短的回顾一下，如下：

基于 GBN 协议的滑动窗口机制，这里的滑动窗口其实可以理解为“流水线”，在窗口范围内，可以在没有收到前面发送的数据报文的确认报文继续发送，直到窗口被塞满。

这里我实现了**超时重传**和**三次快速重传**两种重传机制，超时重传机制是类似 3-1 实验中的设计（有所不同的是，这次不会有最大重传次数了，而是可以无限重传），具体讲解已经在 3-2 实验中详细说明过，这里不进行赘述。

累计确认机制用于确认接收到的数据报文，并告知发送端哪些数据报文已被成功接收。累计确认机制其实主要是应对发送的 ACK 报文丢失的问题，这里我设计接收方窗口大小为 1，确保接收数据报文是**按序接收**，累计确认保证只要发送端收到了一个 ACK 报文，就说明前面的数据报文都被正确接收到了。如果发生了 ACK 报文丢失，我仍可以通过后面发送的 ACK 报文去一起确认之前收到过所有的报文。

最后我展示一下数据传输过程中的 client（发送端）和 server（接收端）的状态机，具体如下：

client(发送端)：

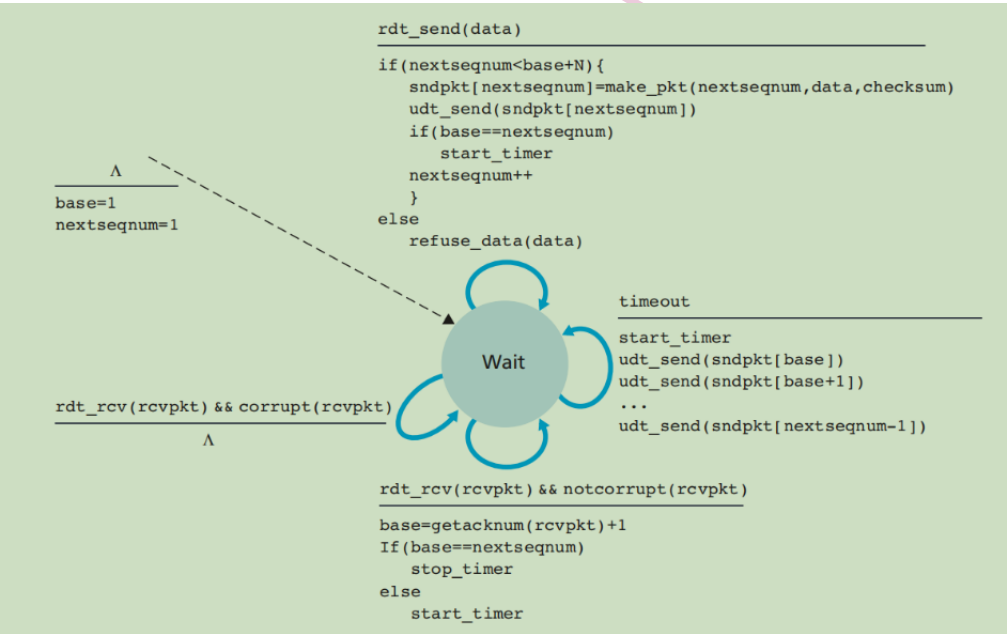


图 2: GBN 发送端状态机

server(接收端)：

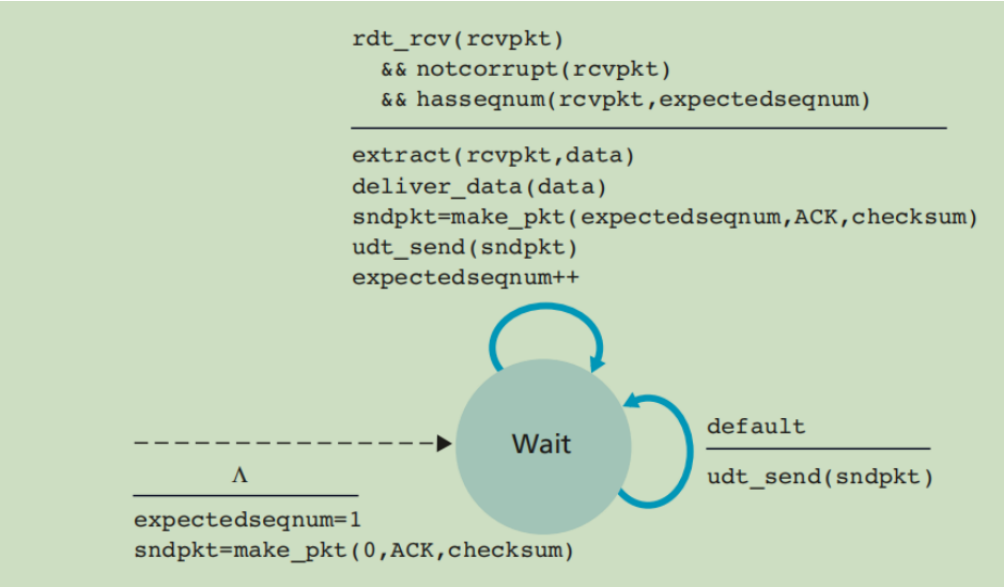


图 3: GBN 接收端状态机

SR 协议 由于之前的实验中已经对协议设计进行了详尽的说明和分析，所以这里只是简短的回顾一下，如下：

client 端和 server 端的工作逻辑（**关键**）：

1. **client 端**：如果发送窗口没被塞满，则按序发送分组报文；如果收到了 ACK 报文，如果确认号在 [start, start+windowssize-1] 区间内，则标记对应的分组报文为 **acked**，如果 start 所在的分组报文被标记成 acked，则可以更新窗口向前滑动，直到遇到没有确认的分组报文。更新窗口后，可以继续发送报文。（窗口的 start 为最小的未确认报文的序列号。）

2. **server 端**：接收到序列号为 n 的报文：n 在 [start, start+windowssize-1] 区间，发送对应 ACK 报文，缓存失序分组，按序到达的分组交付给上层处理（写进 filebuffer 中用于后续文件写入），窗口向前滑动；n 在 [start-windowssize, start-1] 区间，仅发送对应 ACK 报文；对于剩余情况，接收到的报文直接丢掉，不做任何处理。

超时重传 超时重传机制这里的设计我是参考了类似 3-1 实验中的设计（有所不同的是，这次不会有最大重传次数了，而是可以无限重传，同时这次并不是一共设置一个计时器来进行超时逻辑的判断，而是借用结构体中的 Message 和 sendTime 成员来实现对每个报文都设置对应的计时器并缓存），在开始文件传输之前会把窗口内的这个结构体实例中的 sendTime 成员都初始化为：**std::numeric_limits<clock_t>::max()**。在报文发送的时候会为报文设置其发送时间，如果某个报文超时（超时时间内没有被确认），则只会重传这个报文，并不会像 GBN 协议中重传所有缓冲区中的报文，这样可以节省了时间。每次重传的时候会重启对应的定时器。

4. 断开连接：四次挥手（包含超时重传）

四次挥手的设计也贯穿了从 3-1 到 3-3 实验，实现的示意图如下：

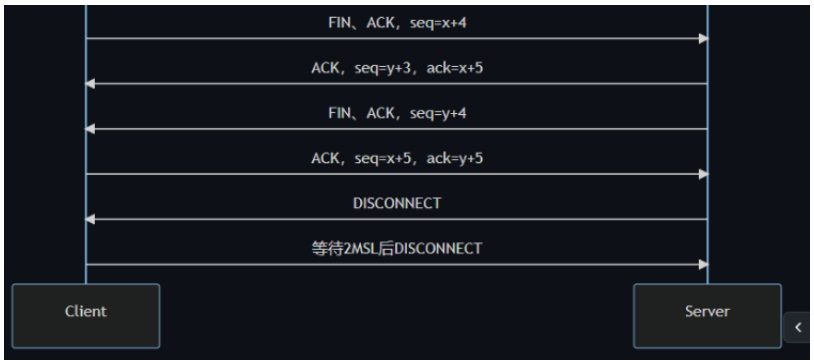


图 4: 四次挥手

具体的流程为了节省篇幅我不在本次实验中赘述了。

四次挥手的超时重传机制 这里我设计了基于 rdt3.0 协议的超时重传机制，下面分别是 client 和 server 的重传机制实现流程：

client：

1. 发送第一次挥手，在发送后接着就开始计时
2. 接收 server 发来的第二次挥手，在这个不断等待接收的过程中，如果第一次挥手超时，则会重传第一次挥手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。
3. 接收第三次挥手
4. 发送第四次挥手，这是一个 ACK 报文
5. 等待 2MSL，由于 client 端并不知道 server 端是否正确收到了最后一次 ACK 报文，为了防止最后一个 ACK 报文丢失，要等待 2MSL，如果再次收到了消息，则说明最后一个 ACK 丢失，会重传第四次挥手的 ACK 包，注意这里是只重传了一次，没有设置最大重传次数。等待结束后则会退出。

server：

1. 接收第一次挥手
2. 接收到第一次挥手后，发送第二次挥手，这是一个 ACK 报文。
3. 发送第三次挥手，并开始计时。
4. 接收 client 的第四次握手，在这个不断等待接收的过程中，如果第三次挥手超时，则会重传第三次挥手并重新计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。接收到第四次挥手就会退出。

5. 日志输出

在前三次实验中，我已经在每次实验中完成了完整的输出设计，这里不再进行赘述。

(二) 核心代码实现（router 程序）

由于原来的 MFC 程序的 router 会发生莫名的失序问题，会导致测试结果产生较大误差，于是这里我维持原 router 逻辑用 python 重写了 router 程序，而且避免了发生意料外的失序问题。

这里仅展示用 python 编写的 router 程序的代码，如下：

router 程序

```python
import socket
import time


def router_main():
    # 从用户那里获取延时和丢包率
    delay = int(input("Enter delay in ms: ")) / 1000  # 将毫秒转换为秒
    loss_rate = float(input("Enter packet loss rate (%): ")) / 100

    # router和server的IP和port
    router_ip, router_port = "127.0.0.1", 30000
    server_ip, server_port = "127.0.0.1", 40460

    # 创建套接字并bind，用的是UDP协议
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((router_ip, router_port))
    print(f"Router is running on {router_ip}:{router_port}")

    client_addr = None
    packet_count = 0  # 初始化包计数器
    loss_every_n_packets = (
        int(1 / loss_rate) if loss_rate > 0 else float("inf")
    )  # 每隔多少个包丢一个包

    while True:
        data, addr = sock.recvfrom(60000)

        # 来自server的包，直接转发给client
        if addr == (server_ip, server_port):
            if client_addr is not None:
                sock.sendto(data, client_addr)
            continue

        # 来自client的包，需要进行延时和丢包处理后转发给server
        client_addr = addr
        packet_count += 1  # 收到一个包，计数器加1

        # 模拟丢包处理
        if packet_count >= loss_every_n_packets:
            print("Simulating packet loss")
            packet_count = 0  # 重置计数器
            continue  # 丢弃当前包，不转发
```

```
43
44          # 模拟延时处理
45          if delay > 0:
46              print(f"Simulating␣delay␣of␣{delay}␣seconds")
47              time.sleep(delay)
48
49          sock.sendto(data, (server_ip, server_port))
50
51
52  if __name__ == "__main__":
53      router_main()
```

(三) 测试

1. makefile 编写

这里简单的编写了一个 makefile 来方便编译和运行，具体代码为了节省篇幅就不复制到报告中了。这里 makefile 的使用也是贯穿前三次实验的，之前已经提交过，这次不会再重复提交了。

2. 性能测试分析（基础与扩展）

地址使用的是本地回环地址 127.0.0.1，router 端口号是 30000，服务器端口号是 40460。超时时间统一设为 3s。

然后同时开启两个命令行，前后分别输入 **make server** 和 **make client** 命令，进行 server 端和 client 端的运行。

注意：下面实验均采用控制单一变量方法进行测试，报文格式、超时时间等，确保每次仅有一个变量变化，保证测试结果的科学准确性。同时为了确保数据的稳定性减轻由于网络情况波动对测试结果造成的影响，以及传输文件的大小的影响。我下面展示的每个数据都是同样条件下对 **1.jpg** 这个测试文件进行**重复测试 5 次**取的平均数。(时延保留整数，吞吐率保留 6 位有效数字) 这样会大幅度的降低由于网络波动产生的误差，让结果稳定并准确。

实验一 停等机制与滑动窗口机制性能对比

这部分的测试滑动窗口（累计确认）的窗口大小为：**client 端为 10；server 端为 1**。

滑动窗口（选择确认）的窗口大小为：**client 端和 server 端均为 10**。

(1) 先进行控制丢包率为 0%，查看不同延时下的停等机制与滑动窗口（累计确认和选择确认）的性能，结果在下表中：

丢包率 0%	延时 0ms	延时 10ms	延时 20ms	延时 30ms	延时 40ms
停等机制					
时延 ms	372	2995	5849	6748	8890
吞吐率 bytes/ms	4992.88	620.151	317.551	275.245	208.926
滑动窗口（累计确认）					
时延 ms	307	2897	5109	6452	8301
吞吐率 bytes/ms	6050.01	641.130	363.545	287.872	223.751
滑动窗口（选择确认）					
时延 ms	308	2953	5377	6015	8237
吞吐率 bytes/ms	6030.37	628.972	345.426	308.787	225.489

表 3: 0% 丢包率下不同延时对停等机制和滑动窗口（累计确认和选择确认）的影响

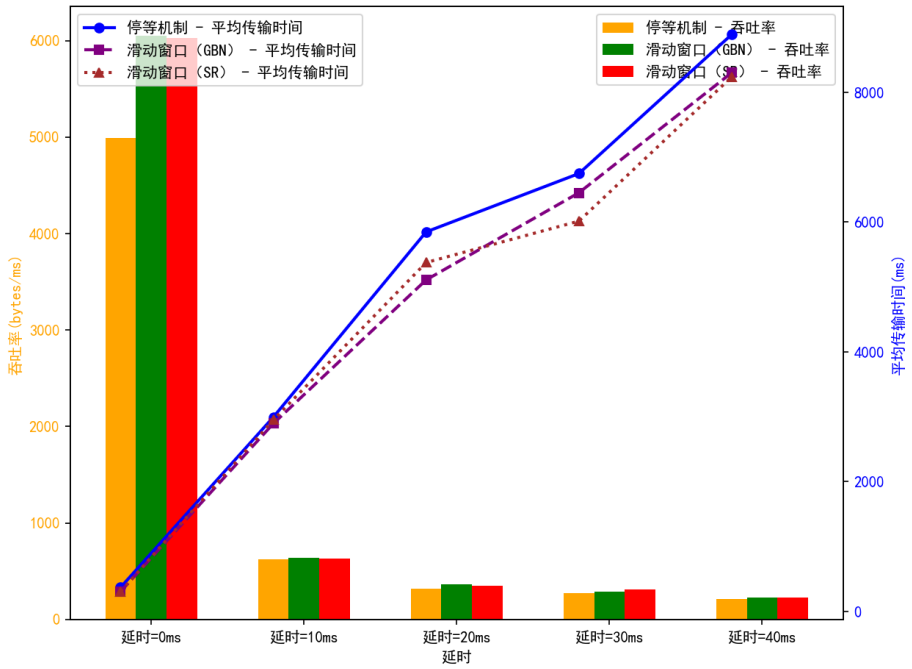


图 5: 0% 丢包率下不同延时对停等机制和滑动窗口（累计确认和选择确认）的影响

分析：丢包率为 0%，改变时延，滑动窗口（累计确认和选择确认）存在加速效果，其 GBN 协议和 SR 协议加速效果相近无法区分优劣。（这是由于这里并没有发生丢包，无法体现这两个协议的本质区别之处）但是 GBN 协议和 SR 协议的滑动窗口加速均不明显。分析打印的日志可知，这主要是因为，当发送一段时间后且没有丢包，滑动窗口会达到动态平衡，每当窗口收到一个确认报文，就会有一个空闲位置，也会立即发送新报文段，所以性能提升不明显。

(2) 进行控制延时为 0ms，查看不同丢包率下的停等机制与滑动窗口（累计确认和选择确认）的性能，结果在下表中：

延时 0ms	丢包率 0%	丢包率 1%	丢包率 3%	丢包率 6%	丢包率 10%
停等机制					
时延 ms	372	6419	18451	36392	58433
吞吐率 bytes/ms	4992.88	289.352	100.664	51.0374	31.7860
滑动窗口（累计确认）					
时延 ms	308	4486	5942	6210	6691
吞吐率 bytes/ms	6030.37	414.033	312.580	299.091	277.598
滑动窗口（选择确认）					
时延 ms	307	6272	13256	23454	32190
吞吐率 bytes/ms	6050.01	296.134	140.114	79.1912	57.6996

表 4: 0ms 延时下不同丢包率对停等机制和滑动窗口（累计确认和选择确认）的影响

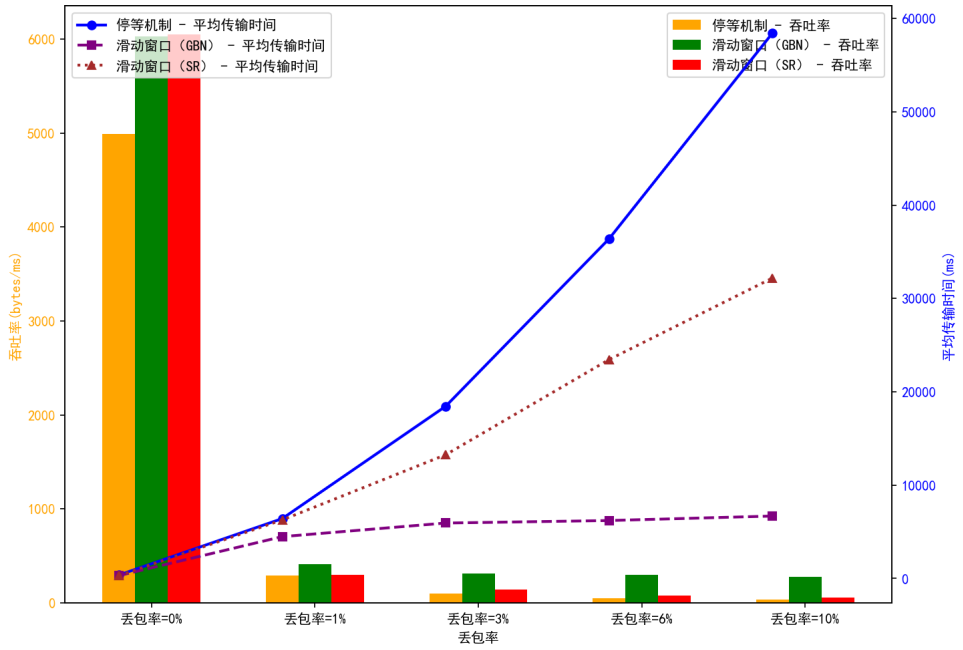


图 6: 0ms 延时下不同丢包率对停等机制和滑动窗口（累计确认和选择确认）的影响

分析：时延为 0ms，改变丢包率，滑动窗口加速效果十分明显。其中 GBN 协议的加速效果远超于 SR 协议。这是因为：

1. 采用流水线发送机制。
2. 有丢包时，基于 GBN 协议的滑动窗口机制实现了三次快速重传机制，可以避免超时等待的时间，当收到三次冗余的 ACK 即可重新发包，大大加速了因为丢包而需要等待的时间。
3. 有丢包时，基于 SR 协议的滑动窗口机制仅实现了超时重传机制，所以虽然流水线明显地加速了重传所需时间，但是每次仍需卡住一个重传时间，由于这里我设置的重传时间是 3s，这个数值较大，所以效果远不如基于 GBN 协议的滑动窗口机制的加速效果。
4. 三次快速重传机制使得滑动窗口加速效果更明显，且在丢包率高时更明显。

实验二 滑动窗口机制中窗口大小对性能的影响（累计确认和选择确认两种情形）

（1）针对滑动窗口（累计确认）进行实验，这里由于 server 端的窗口大小被固定为 1，所以这里我们在改变窗口大小的时候仅改变 client 端的窗口大小。

这部分的实验，我多加了一个**窗口大小等于 3** 的测试（**仅针对累计确认机制的测试**），具体分析我会在后文说明。在控制变量条件下，结果如下：

延时 0ms	丢包率 0%	丢包率 1%	丢包率 3%	丢包率 6%	丢包率 10%
		窗口 =1			
时延 ms	316	6289	18297	38451	62392
吞吐率 bytes/ms	5877.70	295.334	101.511	48.3044	29.7691
		窗口 =3			
时延 ms	307	6273	18256	37454	61190
吞吐率 bytes/ms	6050.01	296.087	101.739	49.5902	30.3539
		窗口 =4			
时延 ms	307	315	324	355	367
吞吐率 bytes/ms	6050.01	5896.36	5732.57	5231.98	5060.91
		窗口 =8			
时延 ms	294	325	347	348	389
吞吐率 bytes/ms	6317.53	5714.93	5352.60	5337.22	4774.69
		窗口 =16			
时延 ms	297	306	322	348	375
吞吐率 bytes/ms	6253.71	6069.78	5768.18	5337.22	4952.94
		窗口 =32			
时延 ms	285	297	305	306	346
吞吐率 bytes/ms	6517.03	6253.71	6089.68	6069.78	5368.07

表 5: 改变丢包率：不同窗口大小的对比（滑动窗口：累计确认机制）

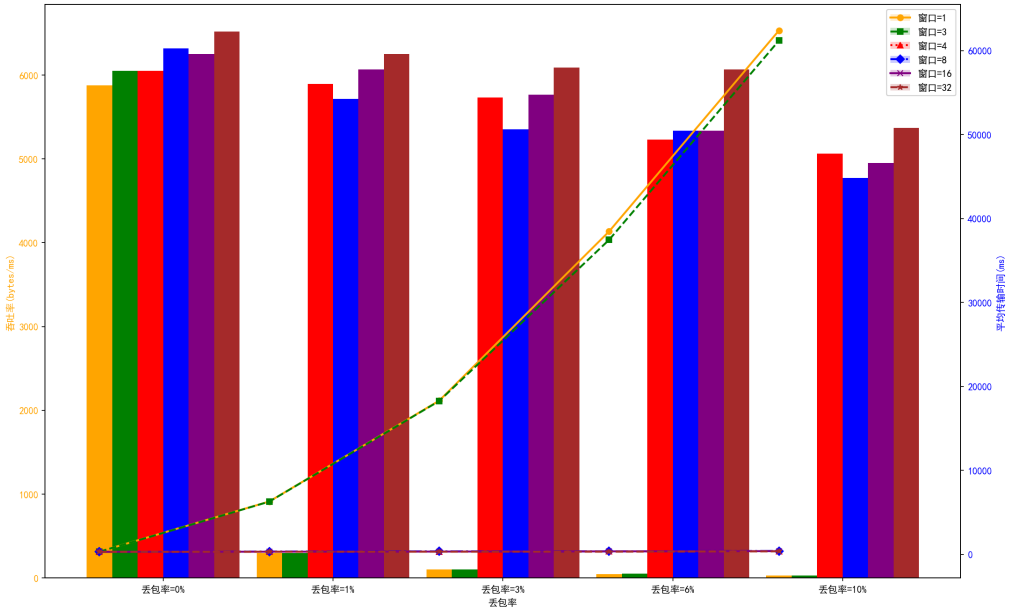


图 7: 改变丢包率：不同窗口大小的对比（滑动窗口：累计确认机制）

丢包率 0%	延时 0ms	延时 10ms	延时 20ms	延时 30ms	延时 40ms
		窗口 =1			
时延 ms	372	3019	5878	6723	8862
吞吐率 bytes/ms	4992.88	615.221	315.984	276.268	209.586
		窗口 =3			
时延 ms	307	2987	5722	6528	8924
吞吐率 bytes/ms	6050.01	621.812	324.599	284.521	208.13
		窗口 =4			
时延 ms	308	2899	5801	6627	8879
吞吐率 bytes/ms	6030.37	640.688	320.178	280.271	209.185
		窗口 =8			
时延 ms	306	2934	5720	6654	8888
吞吐率 bytes/ms	6069.78	633.045	324.712	279.133	208.973
		窗口 =16			
时延 ms	297	2766	5729	6335	8761
吞吐率 bytes/ms	6253.71	671.494	324.202	293.189	212.002
		窗口 =32			
时延 ms	296	2545	5439	6333	8344
吞吐率 bytes/ms	6274.84	729.805	341.488	293.281	222.597

表 6: 改变延时：不同窗口大小的对比（滑动窗口：累计确认机制）

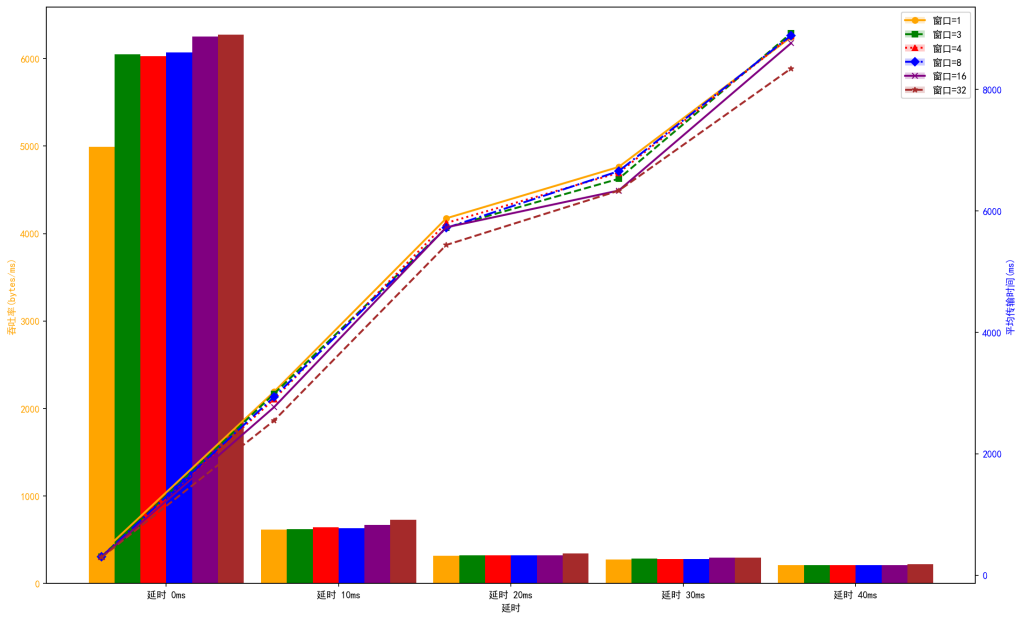


图 8: 改变延时：不同窗口大小的对比（滑动窗口：累计确认机制）

对上述结果一并分析：对于基于 GBN 协议的滑动窗口机制。从总体上看，在不同的丢包率和延时下，窗口越大，时延越低、吞吐率越大。因为窗口越大，可以同时发送的报文就越多，有利于更加充分的利用信道，降低时延，提高吞吐率。

丢包率为 0% 时，只改变延时，滑动窗口的加速比未达到窗口大小比。这是因为当滑动窗口达到平衡时，窗口内一收到一个确认报文，就会有一个空闲位置，也会立即发送新报文段，所以达到一个动态平衡的状态。(这主要是因为在本程序中，没有上层应用缓存报文的时延，导致一有空位就可以发送，而在实际中，存在上层应用产生和给派报文的过程，以及窗口内的缓存时间)。

窗口大小为 1 时的效果跟停等机制的效果相近。

而且可以发现，在窗口小于等于 3 时，由于无法触发三次快速重传机制，只能触发超时重传机制，导致效果与停等机制的效果相近。

(2) 针对滑动窗口（选择确认）进行实验

延时 0ms	丢包率 0%	丢包率 1%	丢包率 3%	丢包率 6%	丢包率 10%
窗口 =1					
时延 ms	315	6247	18301	39004	61361
吞吐率 bytes/ms	5896.36	297.319	101.489	47.6196	30.2693
窗口 =4					
时延 ms	308	6206	17084	38627	60029
吞吐率 bytes/ms	6030.37	299.283	108.719	48.0843	30.9409
窗口 =8					
时延 ms	308	6142	16249	36161	59719
吞吐率 bytes/ms	6030.37	302.402	114.306	51.3634	31.1015
窗口 =16					
时延 ms	301	5988	15409	35628	52085
吞吐率 bytes/ms	6170.61	310.179	120.537	52.1318	35.6600
窗口 =32					
时延 ms	295	5894	15392	27296	39804
吞吐率 bytes/ms	6296.11	315.126	120.67	68.0449	46.6625

表 7: 改变丢包率：不同窗口大小的对比（滑动窗口：选择确认机制）

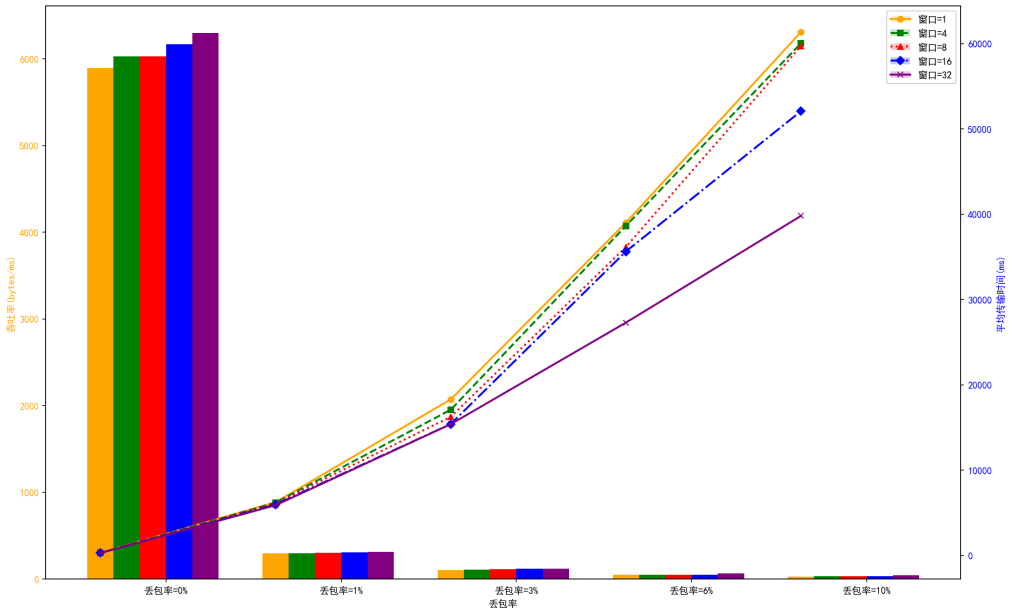


图 9: 改变丢包率：不同窗口大小的对比（滑动窗口：选择确认机制）

丢包率 0%	延时 0ms	延时 10ms	延时 20ms	延时 30ms	延时 40ms
窗口 =1					
时延 ms	315	3011	5827	6744	8723
吞吐率 bytes/ms	5896.36	616.856	318.749	275.408	212.926
窗口 =4					
时延 ms	308	2995	5423	6688	8802
吞吐率 bytes/ms	6030.37	620.151	342.496	277.714	211.015
窗口 =8					
时延 ms	308	2990	5900	6621	8429
吞吐率 bytes/ms	6030.37	621.188	314.806	280.525	220.353
窗口 =16					
时延 ms	301	2872	5741	6502	8401
吞吐率 bytes/ms	6170.61	646.711	323.524	285.659	221.087
窗口 =32					
时延 ms	295	2677	5312	6324	8207
吞吐率 bytes/ms	6296.11	693.819	349.652	293.699	226.313

表 8: 改变延时：不同窗口大小的对比（滑动窗口：选择确认机制）

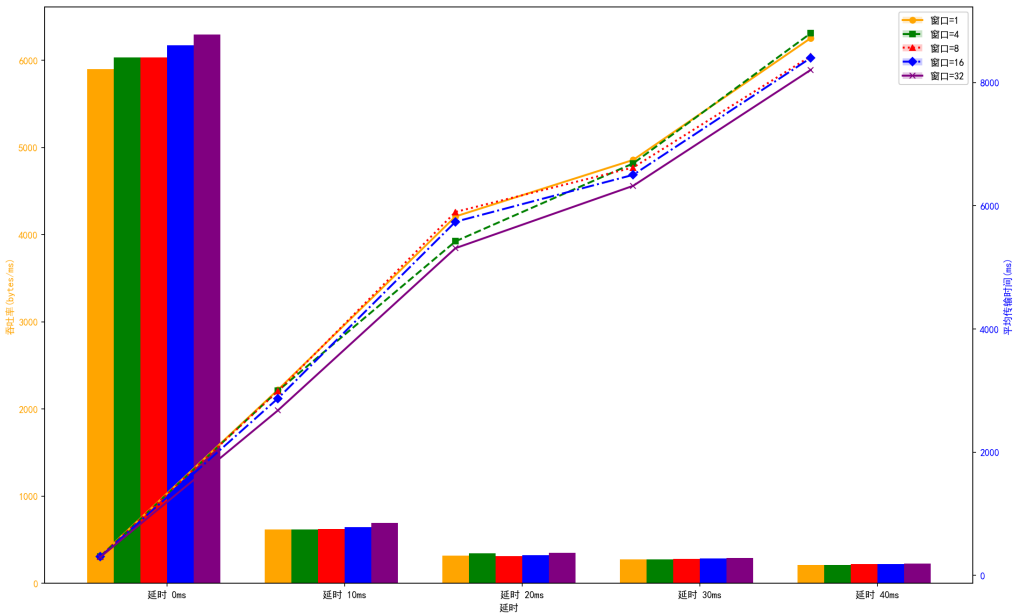


图 10: 改变延时：不同窗口大小的对比（滑动窗口：选择确认机制）

对上述结果一并分析：对于基于 SR 协议的滑动窗口机制。总体上来说，在不同的丢包率和延时下，窗口越大，时延越低、吞吐率越大。因为窗口越大，可以同时发送的报文就越多，有利于更加充分的利用信道，降低时延，提高吞吐率。这一点与 GBN 协议类似。

丢包率为 0% 时，只改变延时，滑动窗口的加速比同样未达到窗口大小比。这是因为当滑动窗口达到平衡时，窗口内一收到一个确认报文，就会有一个空闲位置，也会立即发送新报文段，

所以达到一个动态平衡的状态。(这主要是因为在本程序中，没有上层应用缓存报文的时延，导致一有空位就可以发送，而在实际中，存在上层应用产生和给派报文的过程，以及窗口内的缓存时间)。

窗口大小为 1 时的效果跟停等机制的效果相近。

而且可以发现，由于这里仅实现了超时重传机制，所以在即使窗口大于 3 后，也只能触发超时重传机制，加速效果并不明显。

但是我**细心观察**发现了，在对于丢包率为 6% 时，从窗口 16 到窗口 32 存在较大的加速；在对于丢包率为 10% 时，从窗口 8 到窗口 16 到窗口 32 均存在较大的加速。这是由于窗口大小覆盖了丢包间隔，导致可以有时候可以卡一次超时重传时间可以一次性重传多个丢失的数据包，节省了超时等待的时间。

实验三 滑动窗口机制中相同窗口大小情况下，累计确认和选择确认的性能比较

这部分需要测试的数据，已经均在实验二中测试过了，为了更具有代表性和节省篇幅，这里仅展示窗口大小为 8 和 16 时候的情况（实际上是一起观察分析的)：

延时 0ms	丢包率 0%	丢包率 1%	丢包率 3%	丢包率 6%	丢包率 10%
GBN 协议，窗口 =8					
时延 ms	294	325	347	348	389
吞吐率 bytes/ms	6317.53	5714.93	5352.60	5337.22	4774.69
SR 协议，窗口 =8					
时延 ms	308	6142	16249	36161	59719
吞吐率 bytes/ms	6030.37	302.402	114.306	51.3634	31.1015
GBN 协议，窗口 =16					
时延 ms	297	306	322	348	375
吞吐率 bytes/ms	6253.71	6069.78	5768.18	5337.22	4952.94
SR 协议，窗口 =16					
时延 ms	301	5988	15409	35628	52085
吞吐率 bytes/ms	6170.61	310.179	120.537	52.1318	35.6600

表 9: 改变丢包率：相同窗口大小下，GBN 协议和 SR 协议的对比

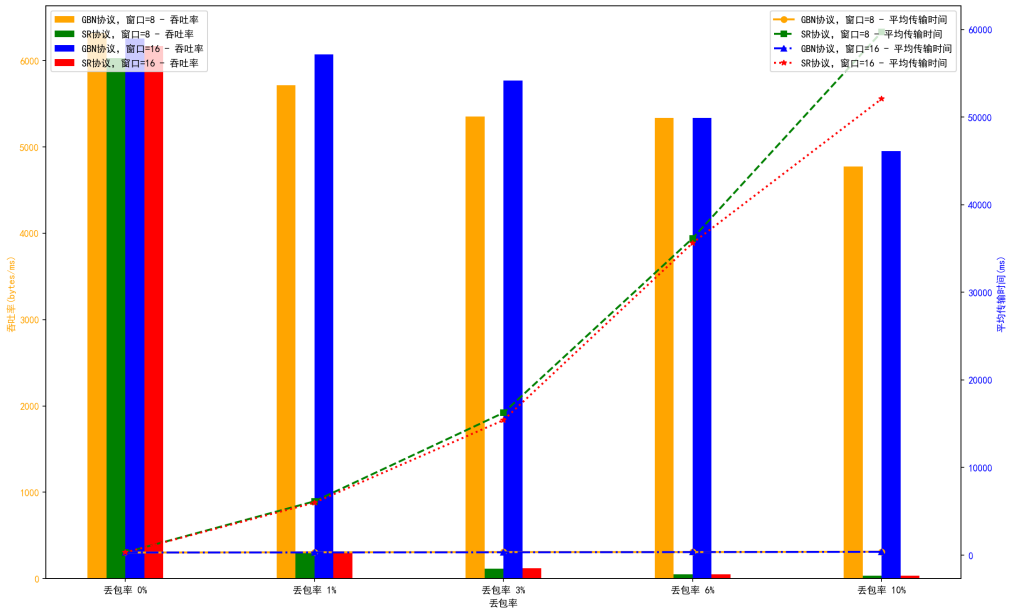


图 11: 改变丢包率：相同窗口大小下，GBN 协议和 SR 协议的对比

丢包率 0%	延时 0ms	延时 10ms	延时 20ms	延时 30ms	延时 40ms
GBN 协议，窗口 =8					
时延 ms	306	2934	5720	6654	8888
吞吐率 bytes/ms	6069.78	633.045	324.712	279.133	208.973
SR 协议，窗口 =8					
时延 ms	308	2990	5900	6621	8429
吞吐率 bytes/ms	6030.37	621.188	314.806	280.525	220.353
GBN 协议，窗口 =16					
时延 ms	297	2766	5729	6335	8761
吞吐率 bytes/ms	6253.71	671.494	324.202	293.189	212.002
SR 协议，窗口 =16					
时延 ms	301	2872	5741	6502	8401
吞吐率 bytes/ms	6170.61	646.711	323.524	285.659	221.087

表 10: 改变延时：相同窗口大小下，GBN 协议和 SR 协议的对比

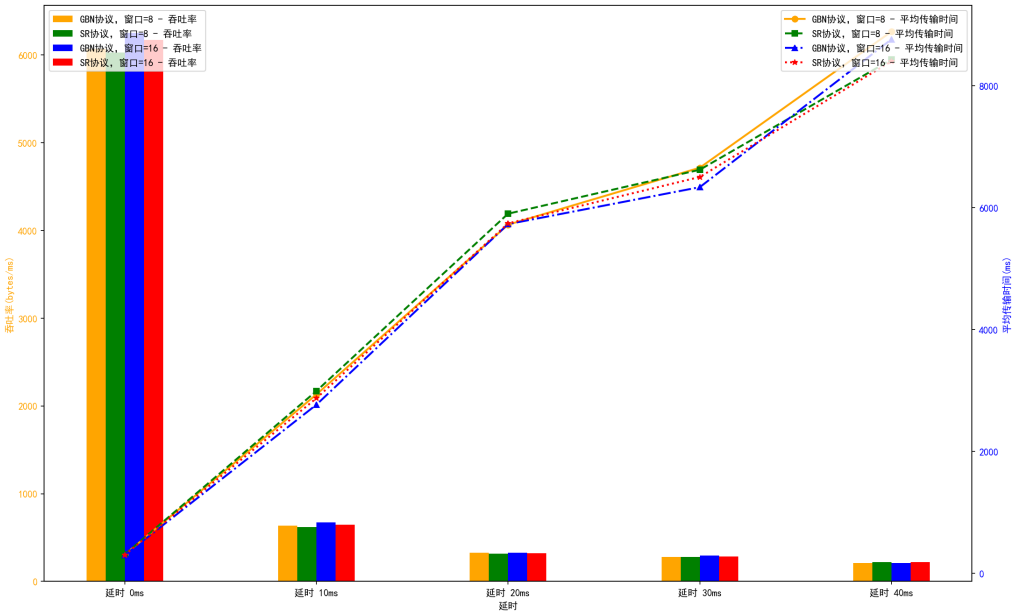


图 12: 改变延时：相同窗口大小下，GBN 协议和 SR 协议的对比

对上述结果一并分析:

1. 不同丢包率的时候，可以发现在窗口为 1 时，两者效果相近无法区分，这是因为窗口为 1 时，两者都近似与停等机制。在窗口大于等于 4 的时候，可以看到由于基于 GBN 协议实现的滑动窗口传输速率远大于基于 SR 协议实现的滑动窗口，这是由于触发了三次快速重传，而 SR 协议仅能触发超时重传协议。（说明，这里并没有很明显的体现 SR 协议在重传的时候由于 GBN 协议的优势，这是由于这里重传报文数量增加对时间影响不大，另外一方面主要是由于 GBN 协议实现了三次快速重传，而 SR 机制并没有实现，所以影响对比结果）

2. 不同延时的时候，可以发现总体上传输速率是两者相近，这是由于由于没有丢包，无法体现 GBN 协议和 SR 协议的本质区别。

由于上面的结果并没有很好的体现 SR 协议在重传时候的优势，下面针对不同丢包率下，相同窗口下，两个协议重新进行测试对比，需要**注意**的是：这次我删除了在 GBN 协议中的三次重传机制，仅保留超时重传机制，其他参数设置一致，（这里**没有进行针对不同延时的测试**是因为快传并不会影响延时的性能表现，所以删除快速重传仅会影响到不同丢包率下的测试结果）结果如下：

延时 0ms	丢包率 0%	丢包率 1%	丢包率 3%	丢包率 6%	丢包率 10%
GBN 协议，窗口 =8					
时延 ms	294	6307	18762	38294	64098
吞吐率 bytes/ms	6317.53	294.491	98.9955	48.5025	28.9768
SR 协议，窗口 =8					
时延 ms	308	6142	16249	36161	59719
吞吐率 bytes/ms	6030.37	302.402	114.306	51.3634	31.1015
GBN 协议，窗口 =16					
时延 ms	308	6261	18027	38244	63021
吞吐率 bytes/ms	6030.37	296.654	103.032	48.5659	29.472
SR 协议，窗口 =16					
时延 ms	301	5988	15409	35628	52085
吞吐率 bytes/ms	6170.61	310.179	120.537	52.1318	35.6600
GBN 协议，窗口 =32					
时延 ms	302	6101	17629	37584	60223
吞吐率 bytes/ms	6150.18	304.434	105.358	49.4187	30.8413
SR 协议，窗口 =32					
时延 ms	295	5894	15392	27296	39804
吞吐率 bytes/ms	6296.11	315.126	120.67	68.0449	46.6625

表 11: 改变丢包率：相同窗口大小下，GBN 协议和 SR 协议的对比（GBN 协议删除三次重传机制）

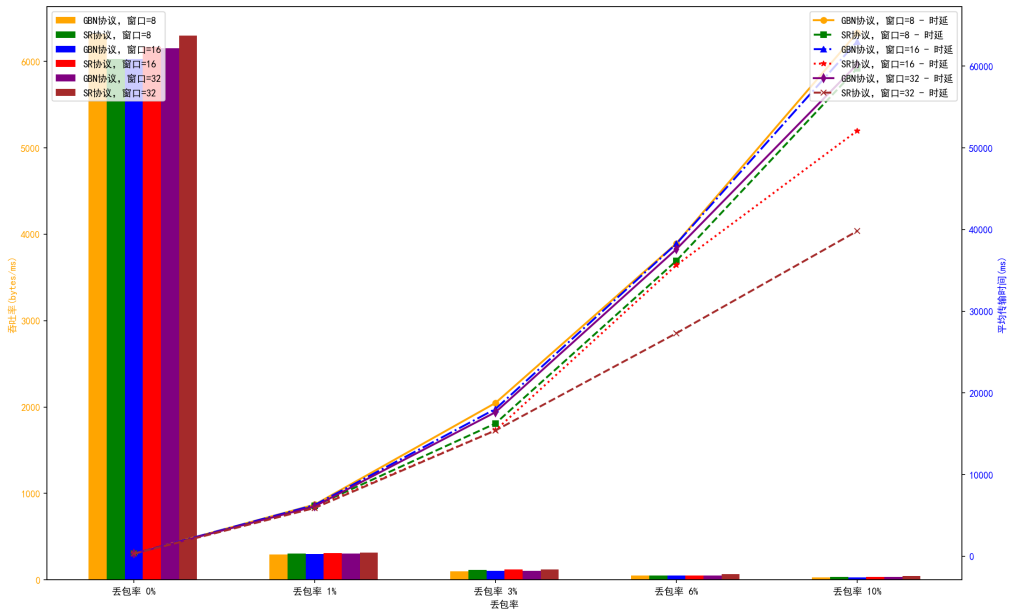


图 13: 改变丢包率：相同窗口大小下，GBN 协议和 SR 协议的对比（GBN 协议删除三次重传机制）

分析：可以发现总体上，不同丢包率下，都是 SR 协议的传输速率有小幅度的加速效果，这种加速效果随着丢包率增大加速效果更加明显，因为随着丢包率，GBN 协议会重传更加多的报文，在丢包率 10% 时，GBN 协议几乎每次卡住都会重传接近窗口大小那么多的报文，远远多于 SR 协议重传的报文数量，这一点体现了 SR 协议相对于 GBN 协议的优势。

三、 总结与问题分析

这次实验在编码方面几乎没有难度，只是自己额外的实现了一个 router 的 py 程序来针对性能进行测试。以及简单修改一下 3-2 实验中的代码。

但是我其实在没有实现新的 router 前遇到了较大的问题，因为路由器会发生莫名的失序，经过分析大概是 MFC 程序的 router 的随机延时性导致的。在用 py 编写 router 后成功避免了这种失序问题，让测试出的结果都较为准确。

遇到的问题就是针对各种性能测试中出现的一些意外情况进行分析等等，但是最后也都成功的分析出关键点，成功解决。

到此，整个学期的计算机网络实验就到此为止了。在这整个的实验过程中，我对 socket 编程更加的熟悉，使用起来也更加的得心应手。同时对于协议设计有了更加深刻的认识和了解。而且来对多线程编程也更加熟练了，可以说我在整个的实验过程中，收获了很多，也得到了很多不局限于学习的宝贵经验。

最后我要感谢我的助教学姐，她帮助了我很多，回答了我很多的问题（其中也包括一些蠢问题），十分感谢助教学姐的耐心回答！希望未来后会有期!