

南开大学

计算机网络课程实验报告

实验3-4



学院：网络空间安全学院

专业：信息安全

学号：2113997

姓名：齐明杰

班级：信安2班

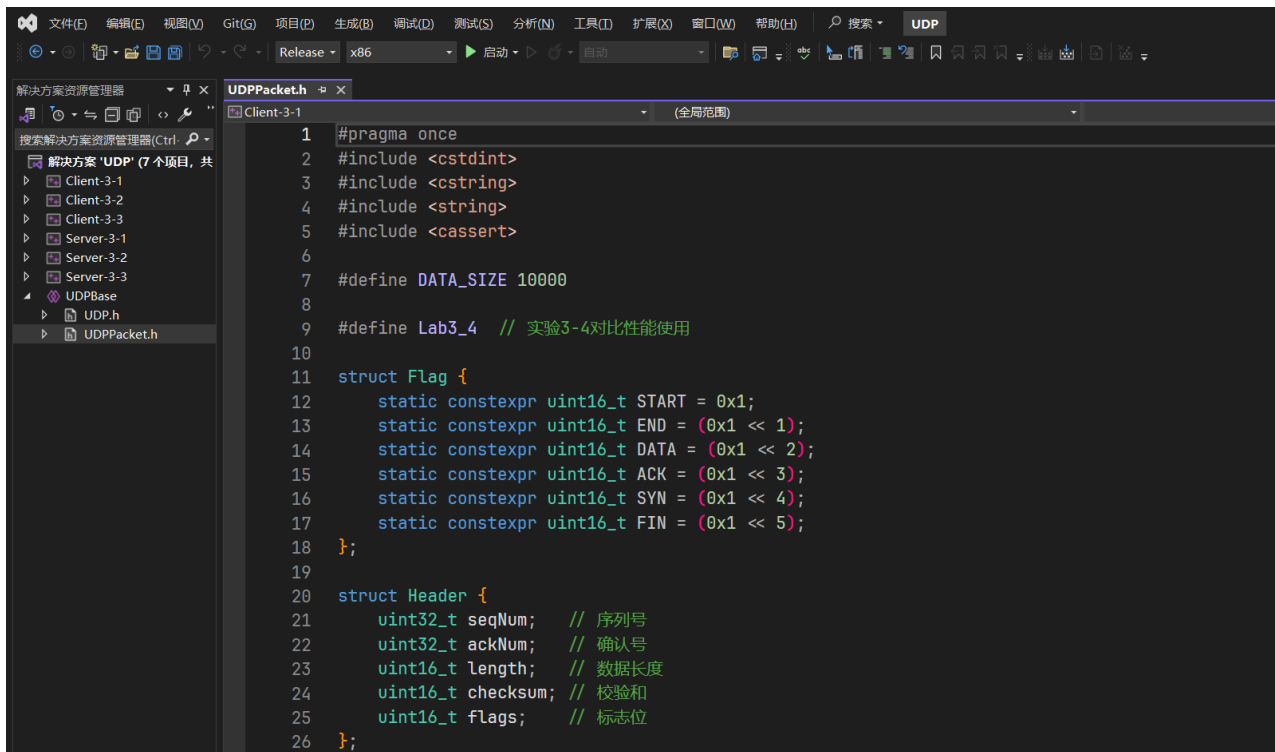
1 实验要求

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（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。
- 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）
 - 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
 - 延时、丢包率对比设置：要有梯度（例如 30ms, 50ms, ...；5%，10%，...）
 - 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）
 - 性能测试指标：时延、吞吐率，要给出图、表并进行分析

2 实验环境

Windows 10 + Visual Studio 2022 community



```
1  #pragma once
2  #include <stdint>
3  #include <cstring>
4  #include <string>
5  #include <cassert>
6
7  #define DATA_SIZE 10000
8
9  #define Lab3_4 // 实验3-4对比性能使用
10
11 struct Flag {
12     static constexpr uint16_t START = 0x1;
13     static constexpr uint16_t END = (0x1 << 1);
14     static constexpr uint16_t DATA = (0x1 << 2);
15     static constexpr uint16_t ACK = (0x1 << 3);
16     static constexpr uint16_t SYN = (0x1 << 4);
17     static constexpr uint16_t FIN = (0x1 << 5);
18 };
19
20 struct Header {
21     uint32_t seqNum; // 序列号
22     uint32_t ackNum; // 确认号
23     uint16_t length; // 数据长度
24     uint16_t checksum; // 校验和
25     uint16_t flags; // 标志位
26 };
```

3 实验背景

在之前的三个实验中，我们实现了**停等协议**，**GBN协议**，**SR协议**。这些协议分别展示了不同的数据传输方法和可靠性策略，从最基础的停等协议，到更复杂的滑动窗口协议，如GBN和SR。每一种协议都有其特定的适用场景和优缺点，能够帮助我们更深入地理解网络通信的基本机制和挑战。

3.1 停等协议

停等协议是最基础的数据传输方法，其核心思想是在发送每个数据包后等待确认回复，再发送下一个数据包。这种方法虽然简单，但效率较低，尤其是在高延迟的网络环境中。停等协议适用于简单的通信场景，但在需要高吞吐量和效率的应用中可能不太实用。

3.2 GBN协议

GBN (Go-Back-N) 协议引入了滑动窗口机制，允许发送端在等待确认的同时发送多个数据包。这大大提高了数据传输的效率。然而，**GBN协议在遇到丢包时需要重传整个窗口内的所有数据包**，可能导致重复传输已被接收端正确接收的数据，从而降低了网络资源的使用效率。

3.3 SR协议

SR (Selective Repeat) 协议也采用滑动窗口机制，但与GBN不同的是，它**允许接收端接收并缓存非顺序到达的数据包，并只请求重传丢失或错误的数据包**。这减少了重复传输，提高了协议的整体效率。SR协议更适用于那些需要高可靠性和高效率的网络应用。

通过这些实验，我们不仅掌握了不同类型的数据传输协议的实现，而且深入了解了它们在实际网络环境中的表现和适用性。本次实验将基于这些已有的知识，进行更深入的性能对比分析，以理解在不同网络条件下各协议的表现差异。

4 实验过程

4.1 前期准备：代码重构

为了方便本次实验对不同的参数(如延时，丢包率，窗口大小)进行测试，我调整了代码的日志输出，并将参数从命令行参数进行传入。

因为本次实验重在收集数据，如传输时间，吞吐率，而不在代码的正确性，因此我关闭了大量的日志输出，仅仅保留了少部分：

```

1  #define Lab3_4
2
3  void Print(const std::string& info, Level lv = NOP) const {
4  #ifndef Lab3_4
5      .....
6      ..... // 原有的日志输出代码
7  #endif
8  }

```

我在服务器端的接收函数处添加需要输出的信息，这样除了传输时间，吞吐率，其余的信息都不会输出：

```

1  // 监听并接收数据
2  void UDPServer::receiveData() {
3      bool receivingFile = false;
4      ULONGLONG startTime = 0;
5      ULONGLONG endTime = 0;
6
7      while (true) {
8          UDPPacket packet;
9          if (receivePacket(packet)) {
10             const Header& pktHeader = packet.getHeader();
11
12             // 检查是否是文件传输的开始
13             if (packet.isFlagSet(Flag::START)) {
14                 .....
15             }
16
17             // 如果是数据包，并且已经开始接收文件
18             if (packet.isFlagSet(Flag::DATA) && receivingFile) {
19                 .....
20             }
21
22             // 检查是否是文件传输的结束
23             if (packet.isFlagSet(Flag::END)) {
24                 endTime = GetTickCount64(); // 记录结束时间
25                 closeFile();
26                 receivingFile = false;
27                 double elapsed = static_cast<double>(endTime - startTime)
28                 / 1000.0;
29                 Print("Bytes Recv: " + std::to_string(totalBytesRecv) + "
30                 bytes", INFO);
31                 Print("Time Taken: " + std::to_string(elapsed) + "
32                 seconds", INFO);
33                 Print("Average Speed: " + std::to_string(totalBytesRecv /
34                 elapsed) + " bytes/s", INFO);
35             }
36         }
37     }
38 }

```

```

31 #ifdef Lab3_4    // 输出3-4的信息
32     std::cout << std::to_string(elapsed) << std::endl;
33     std::cout << std::to_string(totalBytesRecv / elapsed) <<
std::endl;
34     std::cout << "end";
35 #endif
36     continue;
37 }
38
39     // 处理其他请求
40     .....
41 }
42 }
43 }

```

我的延时和丢包是在服务器端(即接收端)模拟的，方式如下所示：

```

1  bool UDPServer::receivePacket(UDPPacket& packet) {
2      // 模拟丢包和延时处理参数
3      static std::default_random_engine generator_drop, generator_delay;
4      static std::uniform_int_distribution<int> delayDistribution(0, (UINT)
(1 / drop));
5      static std::uniform_int_distribution<int> lossDistribution(0, (UINT)(1
/ drop));
6      static bool drop = false;
7      static bool delay = false;
8
9      char* const buffer = new char[BUFFER_SIZE];
10     int addrLen = sizeof(clientAddr);
11     int recvLen = recvfrom(serverSocket, buffer, BUFFER_SIZE, 0, (struct
sockaddr*)&clientAddr, &addrLen);
12
13     if (recvLen > 0) {
14         packet.deserialize(std::string(buffer, recvLen));
15         Header pktHeader = packet.getHeader();
16
17         // 检查数据包的检验和
18         if (!packet.validChecksum()) {
19             Print("Checksum failed for packet with seq: " +
std::to_string(pktHeader.seqNum), WARN);
20             return false;
21         }
22
23         // 对数据包模拟丢包和延时
24         if (packet.isFlagSet(Flag::DATA)) {
25             drop = (lossDistribution(generator_drop) == 0);

```

```

26         delay = (delayDistribution(generator_delay) == 0);
27         // 随机选择Data包进行丢包
28         if (drop) {
29             Print("Simulating packet loss for packet seq: " +
std::to_string(pktHeader.seqNum), WARN);
30             return false; // 不处理该包, 模拟丢包
31         }
32         // 随机选择Data包进行延时
33         if (delay) {
34             Print("Delaying ACK for packet seq: " +
std::to_string(pktHeader.seqNum), WARN);
35             Sleep(Delay);
36         }
37     }
38     ackNum = pktHeader.seqNum + 1;           // 更新ACK
39     // 打印接收到的数据包信息
40     PrintPacketInfo(packet, RECV);
41     return true;
42 }
43 else if (recvLen == 0 || WSAGetLastError() != WSAEWOULDBLOCK) {
44     Print("recvfrom() failed with error code: " +
std::to_string(WSAGetLastError()), ERR);
45 }
46 delete[] buffer;
47 return false;
48 }

```

即根据丢包率，每 $\frac{1}{\text{丢包率}}$ 个包丢一个包，延时使用Sleep即可，当然我此处并不是每个包都有延时，日常网络的延时是随机的，因此我的延时也是随机抽取包进行延时，我设置为每10个包有一个包进行延时。

我的超时重传机制中超时时间设置为200ms.

在 `main` 函数中，使用命令行参数传入窗口，延时，丢包，如下面的3-3的服务器端所示：

```

1  int main(int argc, char* argv[]) {
2  #ifndef Lab3_4
3      try {
4          UINT port = 12720; // 可以根据需要修改端口号
5          uint32_t windowSize; // 窗口大小
6
7          std::cout << "Enter the Window Size: ";
8          std::cin >> windowSize;
9
10         UINT delay = 50;
11         double drop = 0.03;

```

```

12
13     UDPServer server(port, windowSize, delay, drop); // 创建UDP服务器实
例
14     server.Start(); // 启动服务器
15     server.Stop(); // 停止服务器
16 } catch (const std::exception& e) {
17     std::cerr << "Error: " << e.what() << std::endl;
18     return 1;
19 }
20 system("pause");
21 #else
22     try {
23         uint32_t windowSize = static_cast<uint32_t>(std::stoul(argv[1]));
24         UINT delay = static_cast<uint32_t>(std::stoul(argv[2]));
25         double drop = std::stod(argv[3]);
26         UDPServer server(12720, windowSize, delay, drop);
27         server.Start();
28         server.Stop();
29     } catch (const std::exception& e) {
30         std::cerr << "Error: " << e.what() << std::endl;
31         return 1;
32     }
33 #endif
34     return 0;
35 }

```

这样就实现了输出清晰简洁化，并且能够通过调整命令行参数来快速改变变量，方便做对比实验。对于3-1，3-2，3-3我的代码均编译在同一个文件夹下：

名称	日期	类型	大小	图标
Client-3-1.exe	2023-12-20 15:10	应用程序	38 KB	
Client-3-2.exe	2023-12-20 15:10	应用程序	45 KB	
Client-3-3.exe	2023-12-20 15:10	应用程序	45 KB	
helloworld.txt	2023-11-01 11:35	文本文档	1,617 KB	
Server-3-1.exe	2023-12-20 15:10	应用程序	41 KB	
Server-3-2.exe	2023-12-20 15:10	应用程序	42 KB	
Server-3-3.exe	2023-12-20 15:10	应用程序	46 KB	

我可以在该文件夹下命令行按如下格式输入：

```

1 3-1
2     Server: delay drop
3     Client:
4 3-2
5     Server: delay drop
6     Client: windowSize
7 3-3
8     Server: windowSize delay drop
9     Client: windowSize

```

即：

```

1 Server-3-1.exe 50 0.03
2 Client-3-1.exe
3
4 Server-3-2.exe 50 0.03
5 Client-3-2.exe 10
6
7 Server-3-3.exe 10 50 0.03
8 Client-3-3.exe 10

```

运行上述3-1的命令结果(传输helloworld.txt)如下所示：

```

管理员: C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.3803]
(c) Microsoft Corporation. 保留所有权利。

D:\study\vsproject\UDP\Release>Server-3-1.exe 50 0.03
ready
3.938000
420469.273743
end
D:\study\vsproject\UDP\Release>

```

```

管理员: C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.3803]
(c) Microsoft Corporation. 保留所有权利。

D:\study\vsproject\UDP\Release>Client-3-1.exe
ready
end
D:\study\vsproject\UDP\Release>

```

可以看到 `ready` 作为开始，`end` 作为结束，而服务端中间的两个数则分别是传输时间和吞吐率，非常直观，据此可以开始真正做实验了。

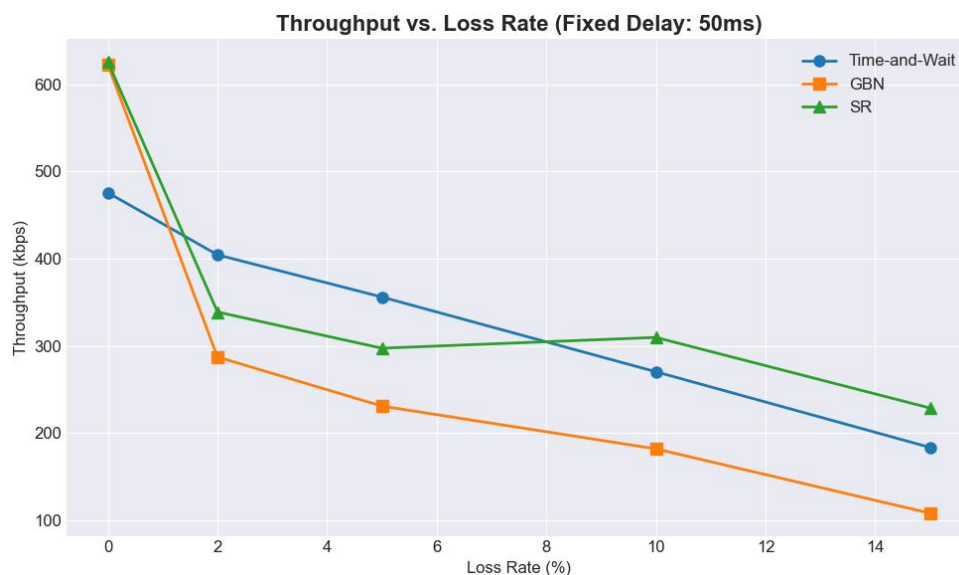
本次实验我均选用1.jpg作为传输文件。

4.2 停等机制与滑动窗口机制性能对比

在这部分实验中，目标是比较停等机制（实验3-1）和滑动窗口机制（实验3-2和3-3）在不同网络条件下的性能。滑动窗口大小设置为16，为了控制变量，我们将分为两种情况来进行比较：

4.2.1 延时固定为50ms，丢包率变化

丢包率	停等机制传输时间(s)	GBN传输时间(s)	SR传输时间(s)	停等机制吞吐率(kbps)	GBN吞吐率(kbps)	SR吞吐率(kbps)
0%	3.90	2.98	2.96	475.51	622.23	625.58
2%	4.59	6.47	5.48	404.30	287.12	338.69
5%	5.21	8.04	6.25	355.88	230.81	297.18
10%	6.88	10.22	6.00	270.16	181.77	309.56
15%	10.13	17.20	8.13	183.44	107.97	228.60



• 结果分析:

随着丢包率的增加，停等协议、GBN协议、SR协议的传输时间均增大，吞吐率均有所减小。

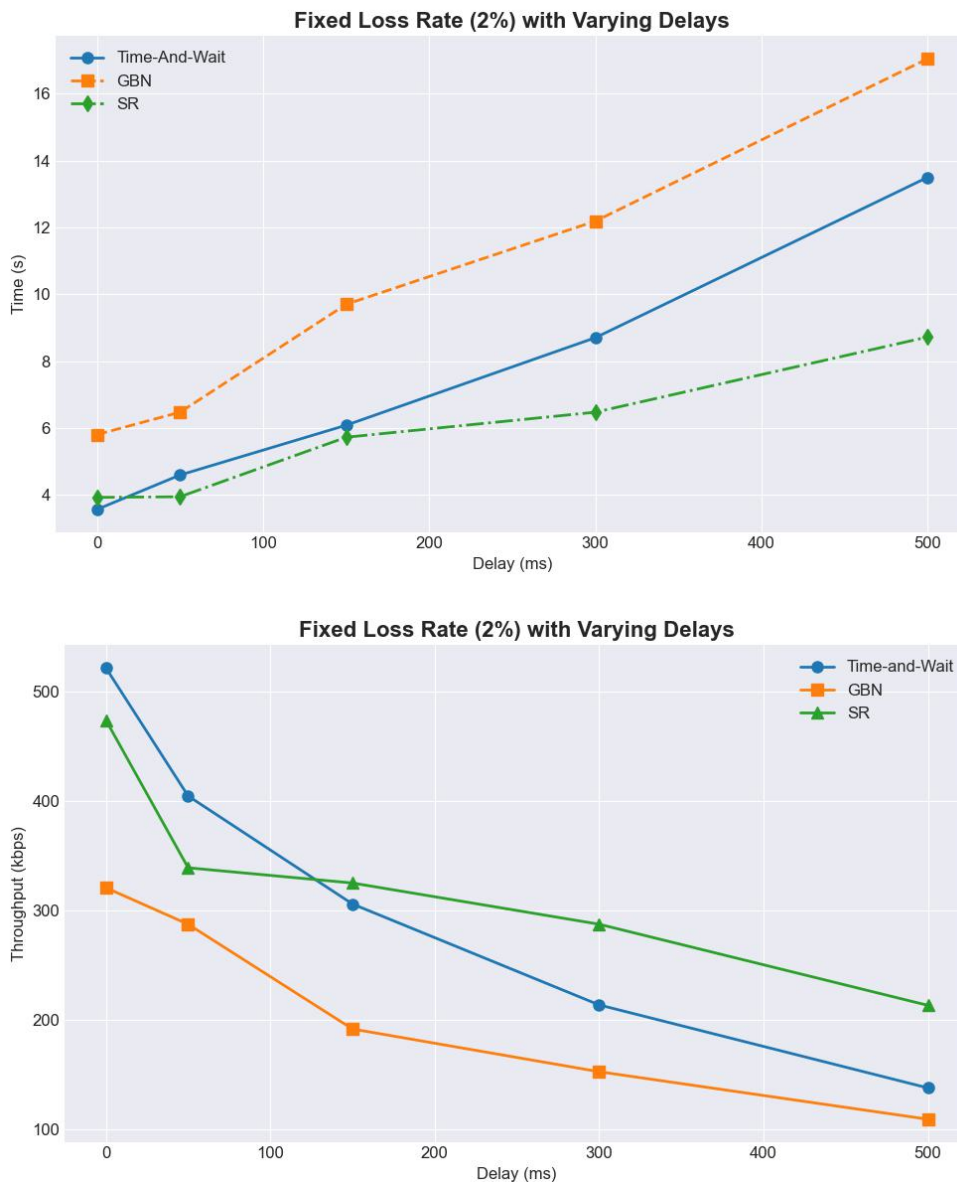
1、**停等协议**：在理论上，停等协议在低丢包率下，由于其简单性，可能表现出更好的稳定性，从而获得较高的吞吐率。然而，随着丢包率的增加，停等协议的性能会显著下降，这是由于每个丢失的数据包都需要重新发送，增加了总的传输时间，并且一旦丢包就会出现等待，只能“一来一回”，效率很低。

2、**GBN协议**：在0%丢包率时，可以看到GBN协议的性能优于停等协议。**值得注意的是，在2%丢包率之上时，停等协议的吞吐率高于GBN协议，这是一个异常情况**。可能的原因包括在GBN协议中，对于每个丢失的数据包，整个窗口内的数据包都需要重传，这在低丢包率时可能导致不必要的重传。而我的滑动窗口大小设置为16，在对比不同窗口大小的实验我们可以看到，对于GBN，窗口为4和窗口为16的时间是极为不同的，因为整个窗口内的数据包都得重传，滑动窗口的大小对于有丢包的GBN来说影响是巨大的，倘若窗口为64，由于丢包，可以推断出GBN将会更加慢上数倍。另外，由于GBN需要使用多线程，而停等协议实现起来相对简单，我认为线程锁机制带来的延时也是一大因素，在修改各种变量时，我不可避免地要加上线程锁，这样导致了代码停滞等待锁的释放。以上两点，是我认为导致GBN慢于停等协议可能的主要因素。

3、**SR协议**：在所有情况下，SR协议表现出较高的吞吐率和较短的传输时间。这是由于SR协议的选择性重传机制只重传丢失的数据包，允许接收方独立确认每个数据包，从而减少了重复传输。SR的重传不需要像GBN那样把整个窗口的内容都重传，因此其传输速率非常稳定，明显快于停等协议。至于为什么没有比停等快很多，我认为也是线程锁机制所导致的。

4.2.2 丢包率固定为2%，延时变化

延时 (ms)	停等机制传输 时间(s)	GBN传输 时间(s)	SR传输时 间(s)	停等机制吞吐 率(kbps)	GBN吞吐率 (kbps)	SR吞吐率 (kbps)
0	3.56	5.80	3.92	521.29	320.45	473.57
50	4.59	6.47	3.94	404.30	287.12	338.69
150	6.08	9.70	5.72	305.59	191.42	324.77
300	8.70	12.19	6.47	213.42	152.40	287.12
500	13.50	17.05	8.72	137.58	108.96	213.02



• 结果分析:

- 1、**停等机制**: 在增加的延时条件下, 停等机制的传输时间显著增加, 且吞吐率持续下降。这是因为在停等机制中, 每个数据包的确认都受延时的直接影响, 导致整体传输效率下降。
- 2、**GBN协议**: 相比于停等机制, GBN协议在延时增加时的性能下降更为显著。由于GBN协议在超时重传情况下和丢包同样需要重传整个窗口的数据, 同样根据上文的分析, 窗口大小较大和线程锁的使用是造成我的GBN吞吐率急速下降的因素, 延时的增加加剧了重传的影响, 导致传输时间增加和吞吐率降低, 慢于停等协议。
- 3、**SR协议**: SR协议相比GBN在延时增加的情况下表现出更好的稳定性, 在高延时(导致超时重传)下对比停等机制优势非常明显。尽管SR协议的传输时间和吞吐率也受到延时影响, 但由于其选择性重传机制, 减少了不必要的重传, 因此性能下降不如GBN协议显著。

但是出现了**另一个异常情况**：延时为0和50时，停等协议比GBN和SR都要快。其实，这个情况在前面丢包率对比已经出现(丢包率2%时停等最快)，我认为这跟我运行的环境和我代码的处理方式有关。我对延时的模拟是在接收端收到包后采用Sleep的方法进行，这意味着代码将陷入睡眠，无法按照逻辑处理期间其他的包，而现实中的延时并不会使代码陷入睡眠，收到的包也会立即处理。也就是说我模拟的延时并不是一个完美模拟(我认为如果要完美模拟延时需要多开一个线程专门做延时)，只是一个近似，这导致了误差。另外在停等协议中，每个数据包发送后都需要等待确认，这意味着在几乎零延迟的环境中，确认几乎可以立即回来，从而允许快速发送下一个数据包。而在GBN和SR协议中，即使在零延迟的情况下，也需要处理更多关于窗口管理和确认的逻辑，这可能导致额外的时间开销。尽管GBN和SR协议在处理丢包和延迟方面比停等协议有优势，但在特定条件下（如零延迟环境），这些协议的额外复杂性和处理开销可能导致它们的性能不如简单的停等协议。

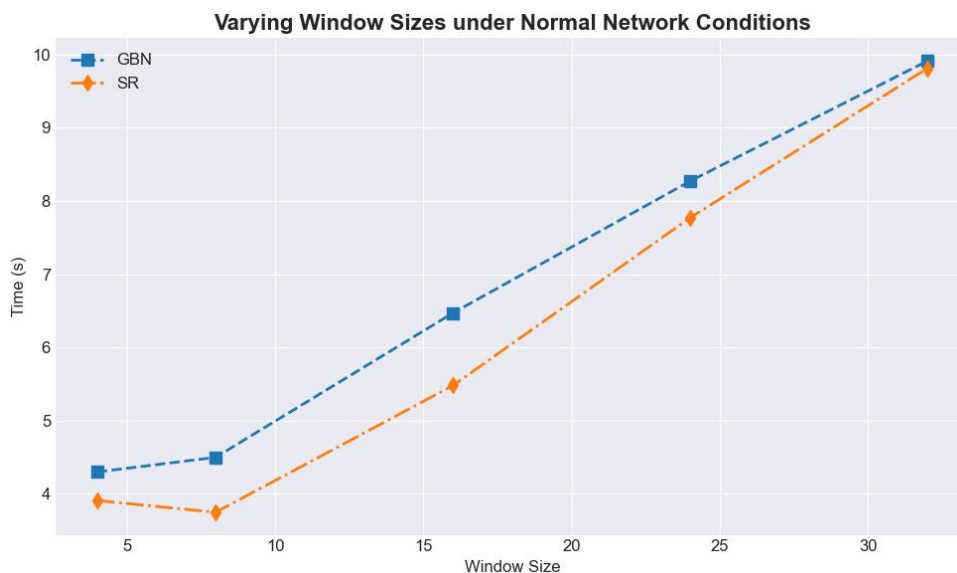
4.3 滑动窗口机制中不同窗口大小对性能的影响

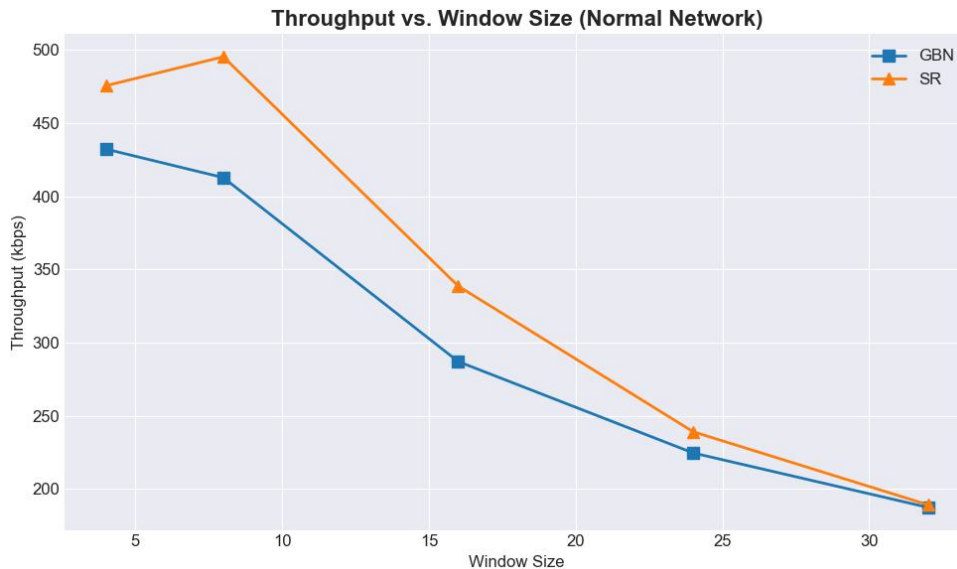
这部分实验将专注于分析窗口大小如何影响滑动窗口机制的性能。我们将通过更改窗口大小来进行实验，同时保持其他参数不变。

然而，在不同的延时和丢包率(主要是丢包率)下，不同窗口大小所造成的影响也不相同，例如网络通顺和网络拥堵的情况下，窗口大小的影响显著程度有所不同，因此我们分两种情况来讨论：

4.3.1 网络通畅：延时固定为50ms，丢包率固定为2%

窗口大小	GBN传输时间(s)	SR传输时间(s)	GBN吞吐率(kbps)	SR吞吐率(kbps)
4	4.30	3.91	432.24	475.51
8	4.50	3.75	412.75	495.29
16	6.47	5.48	287.12	338.69
24	8.27	7.77	224.70	239.20
32	9.91	9.81	187.50	189.29





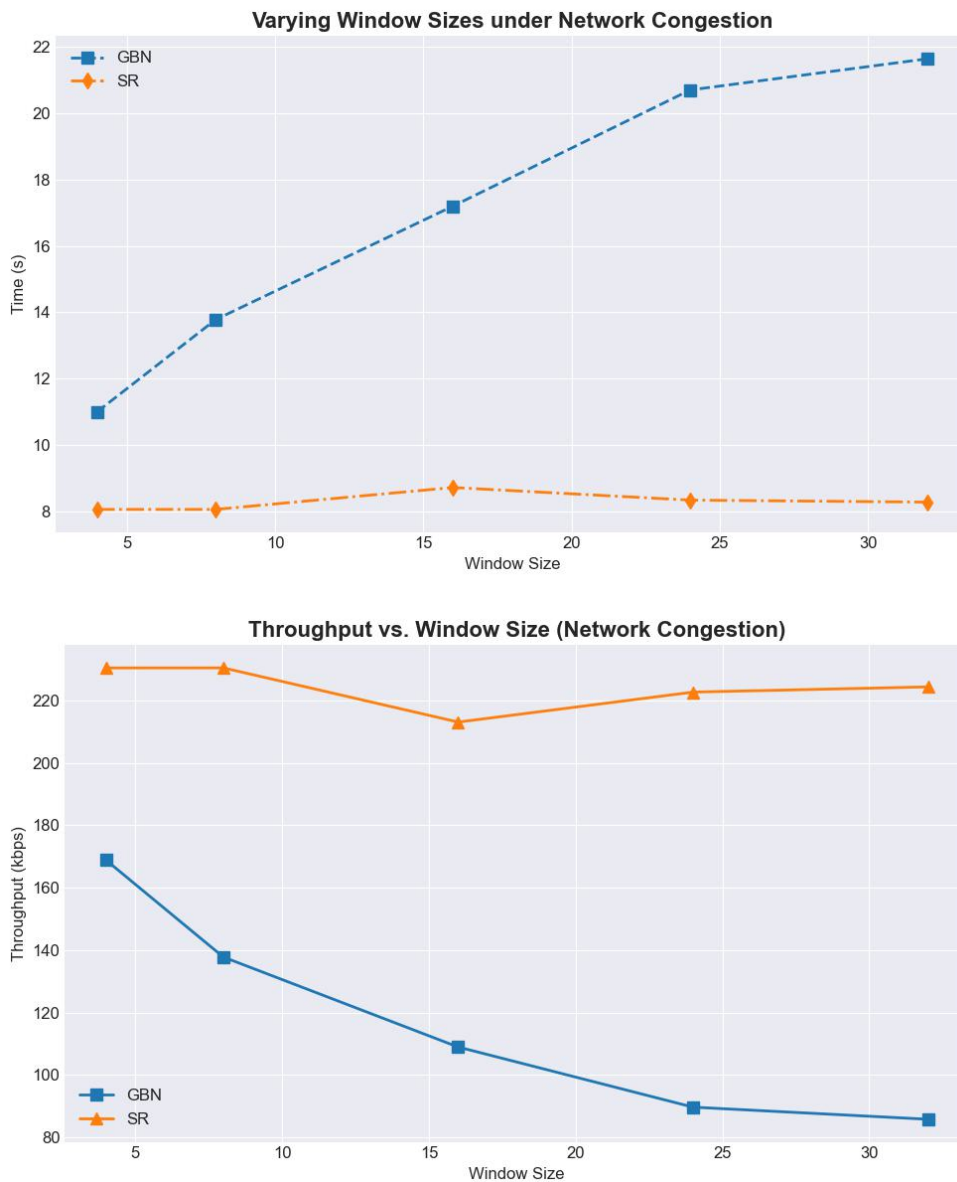
• 结果分析:

在网络通畅的条件下，我们可以观察到随着窗口大小的增加，GBN和SR协议的传输时间都有所增长，但SR协议相对于GBN表现出更好的稳定性。具体来看：

1. **GBN协议**：随着窗口大小的增加，GBN协议的传输时间显著增加，吞吐率显著下降。这可能是因为较大的窗口在丢包发生时需要重传更多的数据，从而导致整体传输时间的增加。此外，GBN协议在处理窗口内所有数据包的确认时可能存在效率问题，特别是在窗口较大时，其处理开销更加显著。**但这是在丢包的情况下才会显著**，我设置的丢包率2%概率很小，在窗口大小不同测试中几乎不会出现丢包，因此即便在窗口很大的情况，GBN仍然和SR差距很小。**这也是为什么我要把窗口大小的影响分为“网络通畅”和“网络拥堵”两种来测试。**
2. **SR协议**：相比GBN，SR协议在窗口大小增加时显示出更好的稳定性。尽管传输时间随着窗口的增大而增加，但增幅较小，吞吐率降低也较为平缓。这可能归因于SR协议的选择性重传机制，即使在窗口较大时，丢失的数据包也可以单独重传，不必像GBN那样重传整个窗口，从而在一定程度上减少了重复传输。

4.3.2 网络拥堵：延时固定为100ms，丢包率固定为15%

窗口大小	GBN传输时间(s)	SR传输时间(s)	GBN吞吐率(kbps)	SR吞吐率(kbps)
4	11.00	8.06	168.85	230.36
8	13.78	8.06	137.78	230.38
16	17.20	8.72	108.96	213.02
24	20.70	8.34	89.71	222.62
32	21.64	8.28	85.83	224.29



• 结果分析:

在网络拥堵的条件下，GBN协议的传输时间显著高于SR协议，且随着窗口大小的增加，这一差距更为明显。SR协议在所有窗口大小下都保持了较好的稳定性和较高的吞吐率。

- **GBN协议：**在高丢包率和延迟的环境下，GBN协议的性能显著下降。特别是在较大的窗口大小下，由于需要重传更多的数据包，导致传输时间大幅增加。这反映了GBN协议在处理高延迟和高丢包率时的不足，尤其是窗口较大时，其性能受到严重影响。
- **SR协议：**与GBN相比，SR协议在高延迟和高丢包率的条件下展现了更佳的性能。尽管随着窗口大小的增加，SR协议的传输时间也有所增长，但其吞吐率保持相对较高。这表明SR协议的选择性重传机制在复杂网络环境下更为有效，能够有效减少不必要的重传，从而保持较高的数据传输效率。

4.3.3 综合分析:

- 在网络通畅和网络拥堵的条件下，SR协议均表现出了相对于GBN更好的性能稳定性和效率。这说明在各种网络条件下，选择性重传机制的优势明显，特别是在面对高延迟和高丢包率时，SR协议能够有效地减少重传，提高数据传输的效率和可靠性。

- GBN协议在窗口较大时，由于其累积确认的特性，一旦发生丢包，整个窗口内的数据包都需要重传，这在网络拥堵的情况下尤为不利，导致其性能大幅下降。
- 这些观察结果强调了在设计和选择数据传输协议时，考虑网络环境的重要性，并且在特定情况下，选择合适的协议和参数调整（如窗口大小）对于优化性能至关重要。

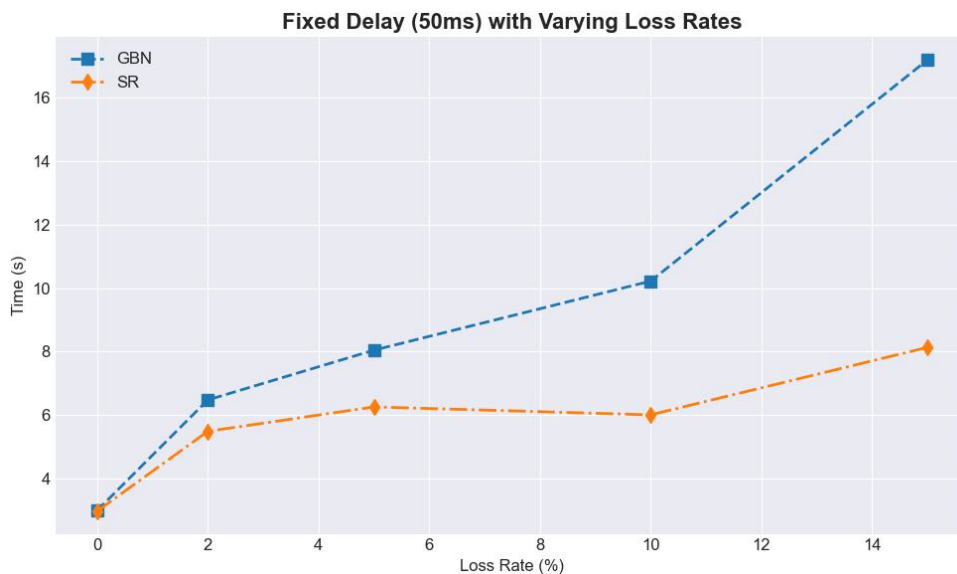
4.4 累计确认和选择确认的性能比较(滑动窗口大小相同)

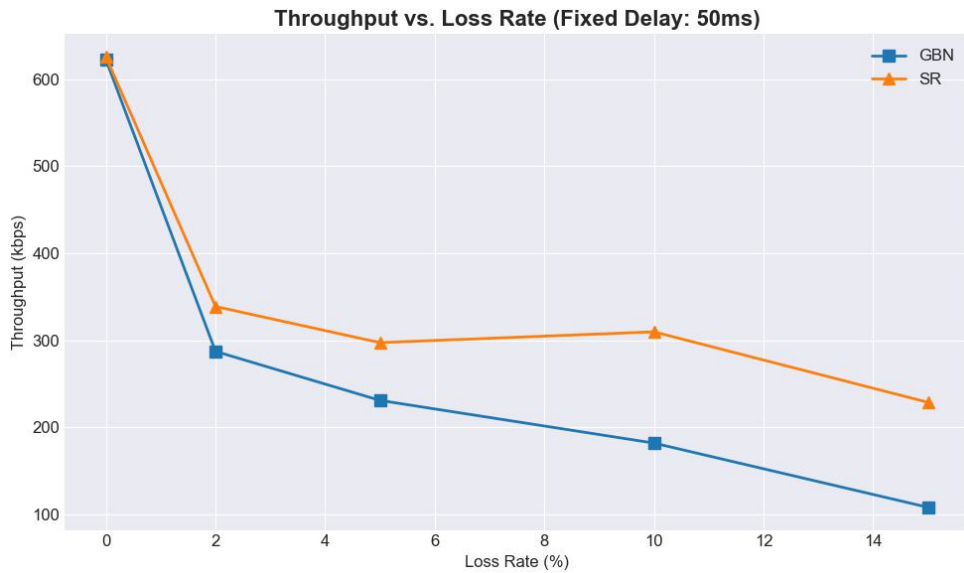
在这个实验中，我们将比较在相同窗口大小的条件下，累计确认（实验3-2）和选择确认（实验3-3）机制的性能差异，使用相同的窗口大小、调整延迟和丢包率进行比较。

滑动窗口大小设置为16。我将分为两种情况来对GBN和SR进行比较：

4.4.1 延时固定为50ms，丢包率变化

丢包率	GBN传输时间(s)	SR传输时间(s)	GBN吞吐率(kbps)	SR吞吐率(kbps)
0%	2.98	2.96	622.23	625.58
2%	6.47	5.48	287.12	338.69
5%	8.04	6.25	230.81	297.18
10%	10.22	6.00	181.77	309.56
15%	17.20	8.13	107.97	228.60





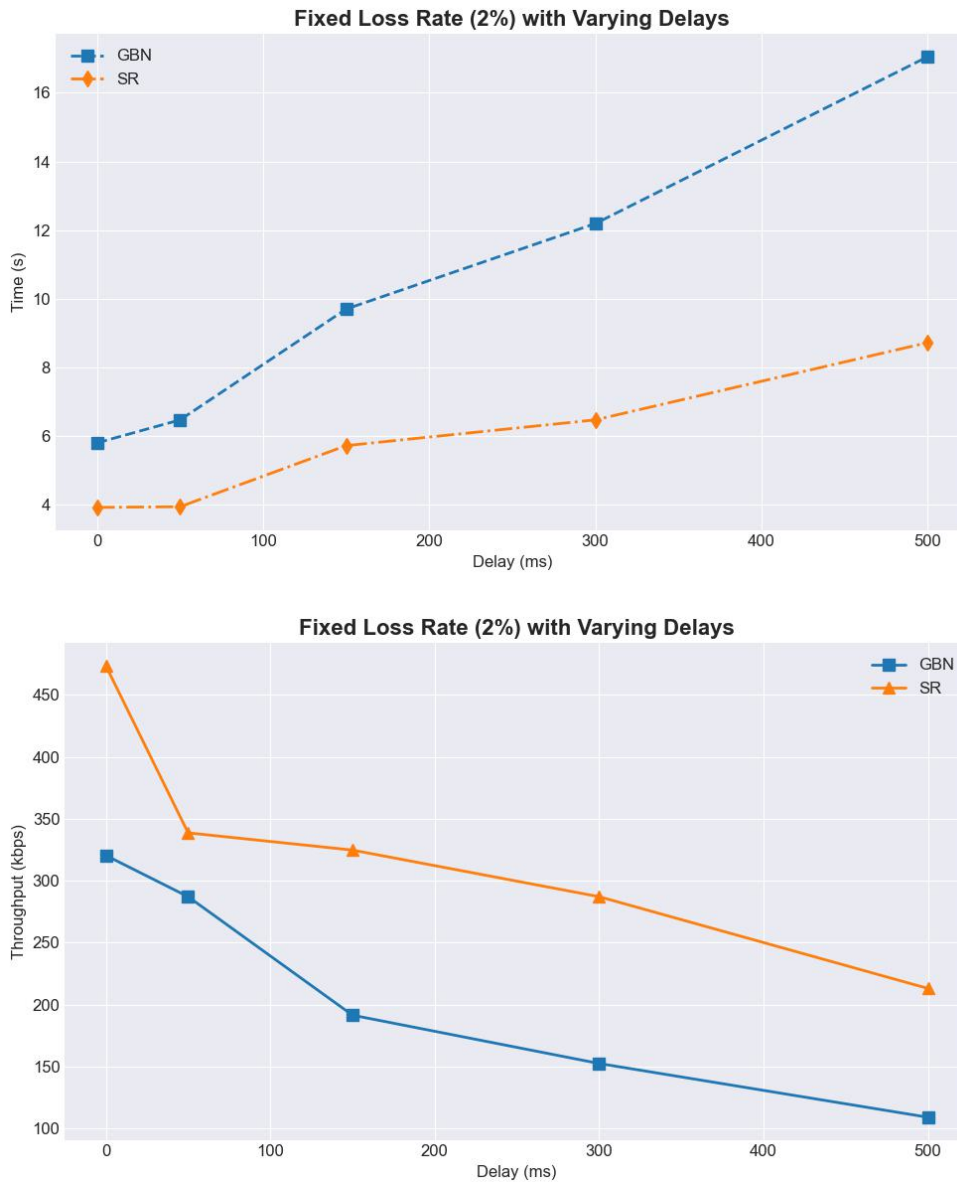
● 结果分析:

在相同的窗口大小设置下，SR协议在各种丢包率条件下均优于GBN协议，无论是在传输时间还是吞吐率方面。这一现象可以归因于两种协议处理数据丢失的机制差异：

1. **GBN协议**：在GBN中，一旦出现数据丢失，需要重传整个窗口内的所有数据包。随着丢包率的增加，重传次数增多，导致传输时间显著增长，吞吐率下降。这表明GBN在高丢包率下的效率大幅降低。
2. **SR协议**：SR协议采用选择性重传机制，仅需重传丢失的数据包。即使在高丢包率下，SR也能维持较高的传输效率，因为它避免了不必要的重传，减少了额外的网络负担。因此，SR在丢包率增加时表现出更好的稳定性和效率。

4.4.2 丢包率固定为2%，延时变化

延时(ms)	GBN传输时间(s)	SR传输时间(s)	GBN吞吐率(kbps)	SR吞吐率(kbps)
0	5.80	3.92	320.45	473.57
50	6.47	3.94	287.12	338.69
150	9.70	5.72	191.42	324.77
300	12.19	6.47	152.40	287.12
500	17.05	8.72	108.96	213.02



• 结果分析:

当丢包率固定而延时变化时，SR协议的性能仍然超过GBN协议。延时对于任何基于时间的网络协议都是一个挑战，但SR的表现尤为出色：

1. **GBN协议**：随着延时的增加，GBN协议的传输时间逐渐增长，吞吐率下降。这可能是因为GBN协议在面对延迟时需要等待整个窗口内所有包的确认，这在延时较高时可能导致效率低下。
2. **SR协议**：SR协议在面对延时时表现出较强的适应性。由于其能够单独确认每个数据包，即使在延时较高的情况下，也能有效地维持传输效率。这表明SR协议在处理网络延迟方面具有优势，能够更有效地适应网络条件的变化。

5 实验总结

本次实验通过对停等协议、GBN协议（累计确认），和SR协议（选择确认）的比较，提供了深入的理解关于不同数据传输协议在不同网络条件下的表现和适应性。以下是实验的关键发现和总结：

- **结论:**

1. **停等协议的适用性:** 停等协议在低丢包率和延时的网络环境中显示出了可接受的性能。然而,在高丢包率或高延时的情况下,其性能迅速下降。这种协议适用于简单或可靠性要求不高的场景,但在高效率和高可靠性需求的环境中不太实用。
2. **GBN协议的局限性:** GBN协议在某些条件下(如低丢包率和延时)表现良好,但在高丢包率或延时的网络中,其性能受到严重影响。特别是在高丢包环境中,由于需要重传整个窗口内的数据包,导致效率显著降低。
3. **SR协议的优越性:** SR协议在各种测试条件下均表现出优异的性能,尤其是在高丢包率或延时的情况下。选择性重传机制使得SR协议在丢包处理上更为高效,减少了不必要的重传,从而在各种网络条件下保持较高的吞吐率。
4. **窗口大小对性能的影响:** 实验数据显示,不同的窗口大小对GBN和SR协议的性能影响显著。在网络通畅的条件下,较大的窗口尺寸能提高传输效率;然而,在网络拥堵的情况下,过大的窗口尺寸反而导致性能下降,尤其对GBN协议而言。这强调了根据网络条件合理选择窗口大小的重要性。
5. **协议选择的重要性:** 本实验结果强调了在不同网络环境下选择合适的协议的重要性。根据网络的特定条件(如延时、丢包率),合理选择协议可以显著提高数据传输的效率和可靠性。

总结: 这次实验不仅增强了对不同数据传输协议特性的理解,也展示了在实际网络设计和应用中,根据特定网络条件选择合适协议的重要性。理解每种协议的优势和局限性对于设计高效、可靠的网络系统至关重要。通过这些实验,我们可以更加深刻地把握不同协议在实际应用中的适用性和性能表现,从而在实际网络设计中做出更明智的选择。

最后,感谢老师和助教的悉心指导!