计算机网络lab3-4

*姓名：**罗瑞***

*学号：**2210529***

*专业：**密码科学与技术***

一、实验内容

✅ 基于给定的实验测试环境，通过改变网路的延迟时间和丢包率，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。

二、协议简介

2.1停等机制（rdt3.0)

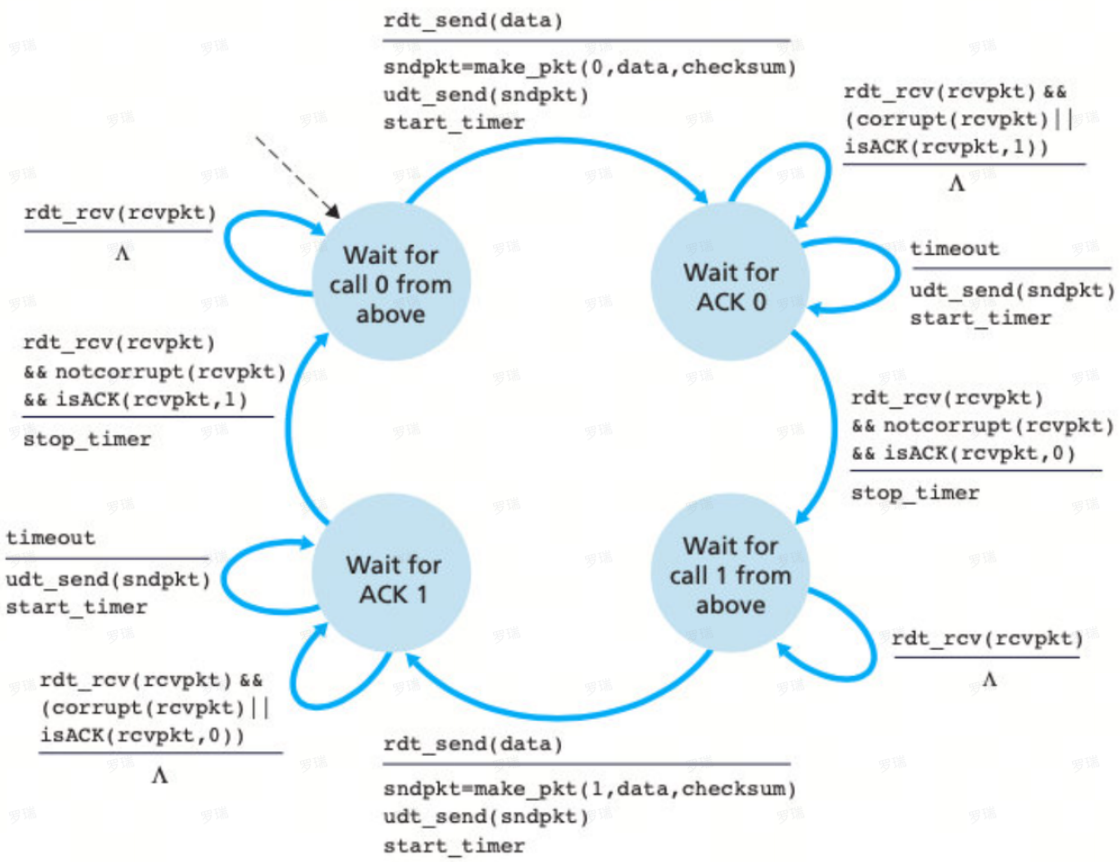

图1：rdt3.0有限状态机

✅ **工作原理**

停等机制（Stop-and-Wait Protocol）是最基础的可靠数据传输协议之一，主要解决数据在不可靠信道上传输时可能出现的问题，包括：

1. 数据包丢失。
2. 数据包损坏。
3. 确认（ACK）丢失或损坏。

停等机制通过以下方式保证数据可靠性：

- **确认机制**：每次发送一个数据包后，发送方必须等待接收方的确认（ACK）才会继续发送下一个数据包。
- **超时与重传**：如果发送方在规定时间内没有收到ACK，认为数据丢失，会重新发送数据包。
- **序列号**：为每个数据包分配唯一的序列号，以便接收方区分重复数据包。

详细流程

1. **数据发送**：发送方发送一个数据包，并启动定时器。
2. **确认接收**：
   - 如果接收方收到数据包且校验通过，则返回ACK（带序列号）。
   - 如果数据包损坏或未收到，接收方不发送ACK。
3. **超时与重传**：
   - 如果发送方在超时时间内未收到ACK，会重新发送当前数据包。
4. **重复包处理**：接收方利用序列号识别并丢弃重复数据包。

优缺点

优点：

- 实现简单，易于理解和部署。
- 能够保证数据可靠传输。

缺点：

- **低效率**：发送方每次只能发送一个数据包，必须等待确认后才能发送下一个数据包。
- **高延迟**：在高延迟网络中，效率进一步降低。

2.2滑动窗口（Go back N）

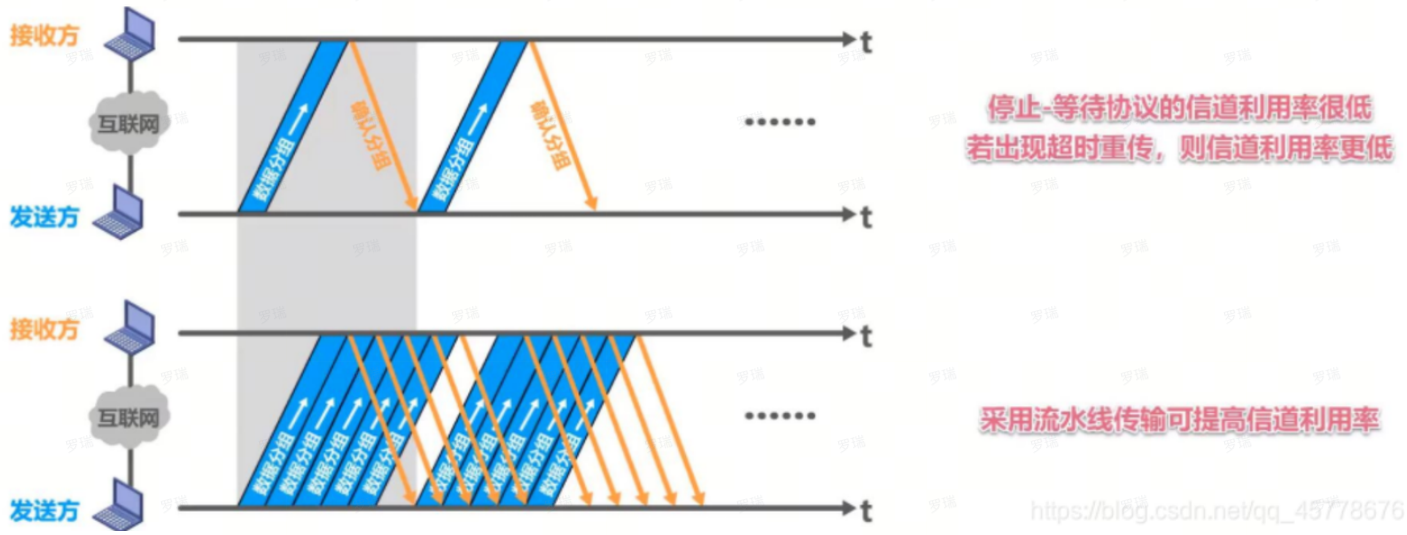

图2：Go back N 协议图解

✅ 工作原理

滑动窗口协议是一种比停等机制更高效的可靠数据传输协议，通过引入窗口机制，允许发送方在等待确认期间连续发送多个数据包，提高了信道利用率。

在Go-Back-N中：

- **发送窗口**：发送方维护一个固定大小的窗口，窗口内的多个数据包可以同时发送。
- **接收窗口**：接收方只接受按序到达的数据包，任何乱序的数据包都会被丢弃。
- **重传机制**：如果某个数据包丢失或损坏，发送方会从该包开始重传后续所有未确认的数据包。

详细流程

1. **数据发送**：
   - 发送方根据发送窗口的大小，连续发送多个数据包。
   - 每个数据包附带序列号，用于区分数据包。
2. **接收与确认**：
   - 接收方校验每个收到的数据包。
   - 如果校验通过，接收方返回确认ACK（带上最新的序列号）。
   - 如果接收到乱序包，接收方丢弃该包，并重复发送上次的ACK。
3. **超时与重传**：
   - 如果发送方在超时时间内未收到某个包的ACK，会从该包开始重新发送后续所有未确认的包。

优缺点

优点：

- 能够显著提高信道利用率（特别是延迟较大的网络）。
- 在网络状况良好的情况下，性能提升显著。

缺点：

- **重传代价大**：当出现丢包时，会重传大量已经成功发送的数据，浪费带宽资源。
- **接收方复杂度低**：接收方不缓存乱序数据包，可能导致数据延迟。

2.3拥塞控制（reno算法）

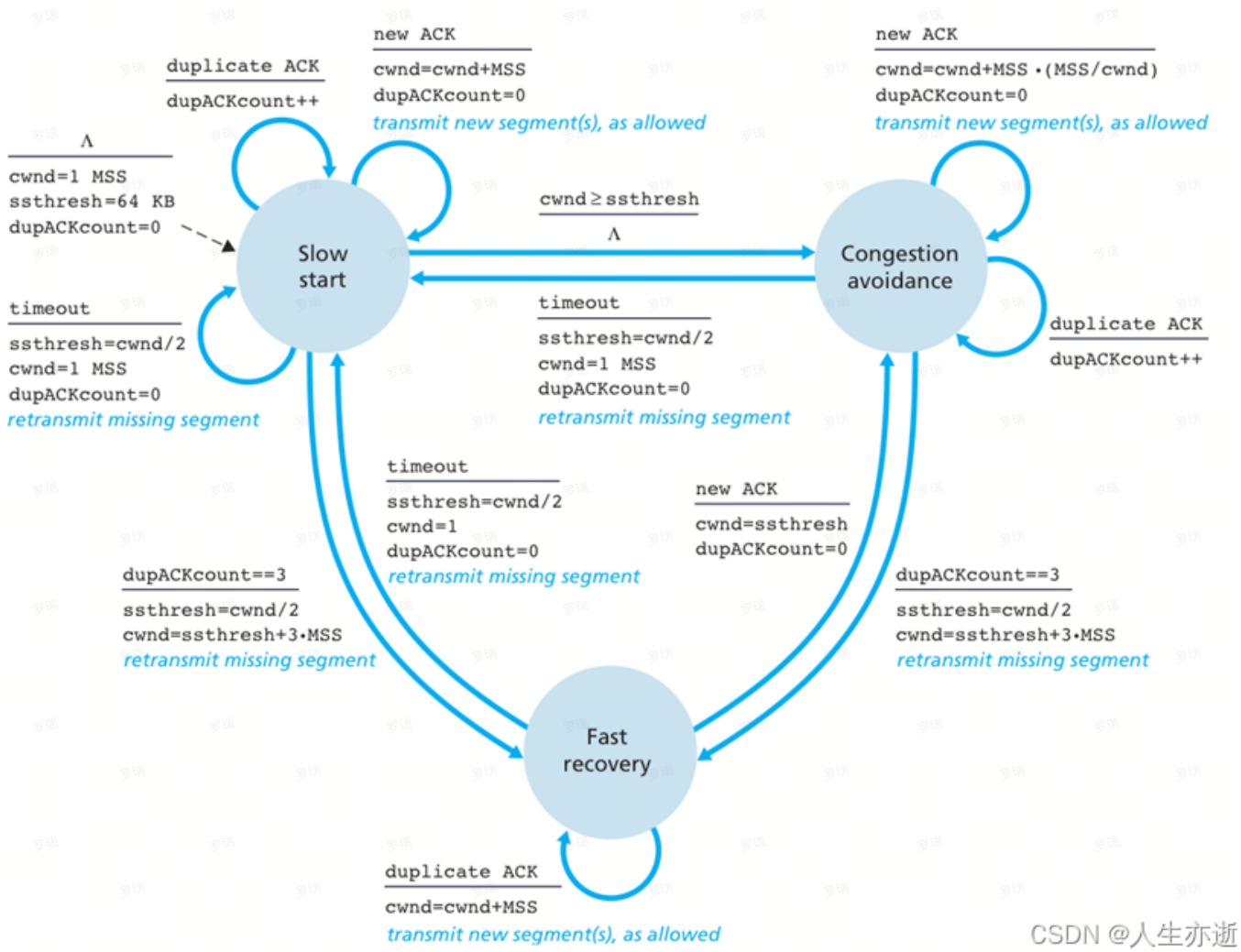

图3：reno拥塞控制算法有限状态机

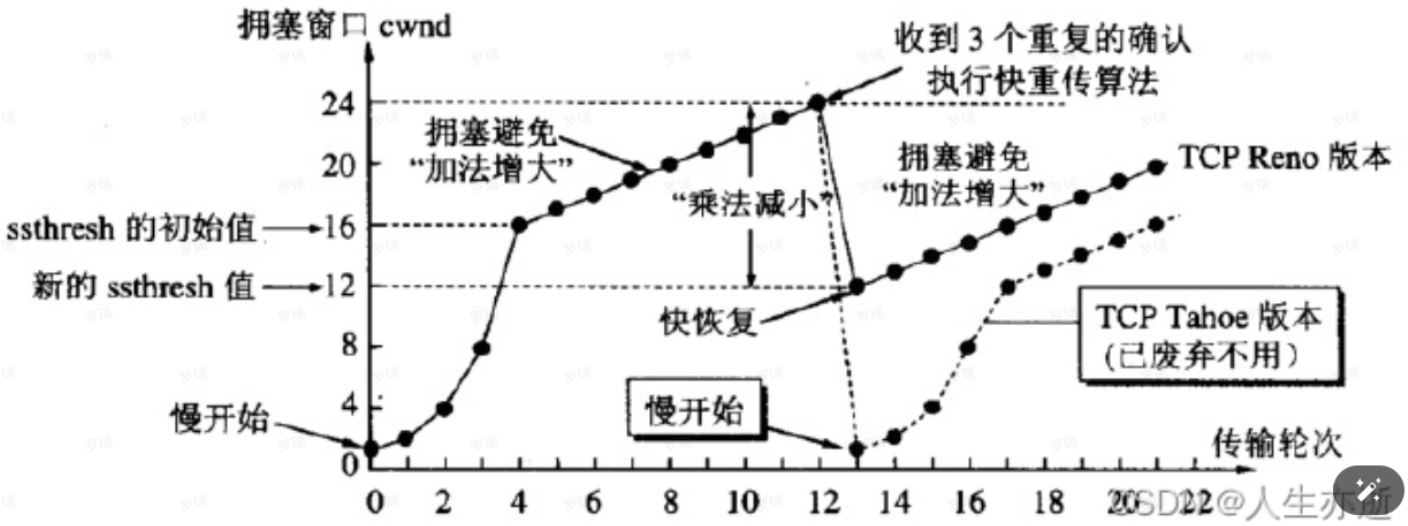

图4:reno拥塞控制不同版本的窗口大小变化示意图

✅ 工作原理

Reno算法是TCP拥塞控制的经典算法之一，通过动态调整发送窗口大小，优化网络资源利用率，并减少网络拥塞时的传输损失。

Reno算法包含以下核心机制：

1. **慢启动（Slow Start）**：在初始阶段，发送窗口大小从1开始，并以指数增长（每次收到ACK时，窗口大小翻倍）。
2. **拥塞避免（Congestion Avoidance）**：当窗口大小接近网络容量时，窗口按线性增长（每个RTT增加一个数据包）。
3. **快速重传（Fast Retransmit）**：如果发送方收到三个重复ACK，判断丢包发生，立即重传丢失数据包。
4. **快速恢复（Fast Recovery）**：丢包后，发送窗口减半（ssthresh = cwnd/2），然后进入拥塞避免阶段，避免完全退回慢启动。

详细流程

1. **慢启动**：
   - 发送方初始设置 `cwnd=1`，`ssthresh=16`（假设值）。
   - 每收到一个ACK，`cwnd` 值翻倍，直到 `cwnd >= ssthresh`。
2. **拥塞避免**：
   - 当 `cwnd >= ssthresh` 时，窗口按线性增长（每个RTT增加一个数据包）。
   - 若检测到网络丢包，进入快速重传和快速恢复。
3. **快速重传与快速恢复**：
   - 当收到3个重复ACK，立即重传丢失数据包。

- 设置 `ssthresh = cwnd / 2`，`cwnd = ssthresh`，然后进入拥塞避免阶段。

优缺点

优点：

- **高效性**：通过快速恢复机制，避免了完全退回慢启动的低效率问题。
- **适应性**：能够快速响应网络拥塞情况，并合理调整发送速率。

缺点：

- 对突发性网络问题响应较慢。
- 在丢包率较高的网络中，仍可能导致性能下降。

三、实验说明（随机丢包与传输时延实现）

✅ 统一设定超时重传时间为0.3s，传输文件为2.jpg,大小为5.62M

实验采用严格的控制变量法

每一次实验均重复三次，便于产生更客观的观测数据

在接收端(server)自行实现控制丢包率和时延，算法展示：

```
int receiveandack() {
    UDP_DATAGRAM recv;
    while (true) {
        memset(&recv, 0, sizeof(UDP_DATAGRAM));
        //Sleep(50);
        //实现时延控制，这里例子是50ms
        recvfrom(sock, (char*)&recv, sizeof(recv), 0, (SOCKADDR*)&client, &l);

  //================================================================================


            //真：随机丢包
             //设置随机种子为当前时间
        countcut++;
        // 生成范围为 1 到 10000 的随机数
        int a = countcut % 100;//随机丢包率，1%
        if (!a)
```

```
15              continue;//跳过后面操作，来模拟丢包
16
17          ……
18
19          }
```

四、停等机制与滑动窗口机制性能对比

这里设置滑动窗口的大小为25，设置超时重传时限统一为0.3s，设置传输文件为2.jpg

4.1统一时延，控制网络丢包率不同

✅ 时延:0ms

丢包率梯度:[1%,2%,4%,5%]

对比rdt3.0与GBN两个协议的数据

观测：传输时间和吞吐率

传输时间和吞吐率	0ms,1%	0ms,2%	0ms,4%	0ms,5%
rdt3.0	8.23571s, 5595.39kb/s	10.4515s, 4409.13kb/s	15.0086s, 3070.37kb/s	17.0852s, 2697.18kb/s
GBN	7.064s, 6552.52kb/s	8.26s, 5586.23kb/s	12.12s, 3803kb/s	14.33s, 3215kb/s

可视化图表：同时展示 RDT 3.0 和 GBN 协议的吞吐率（柱状图）和传输时间（折线图）对比

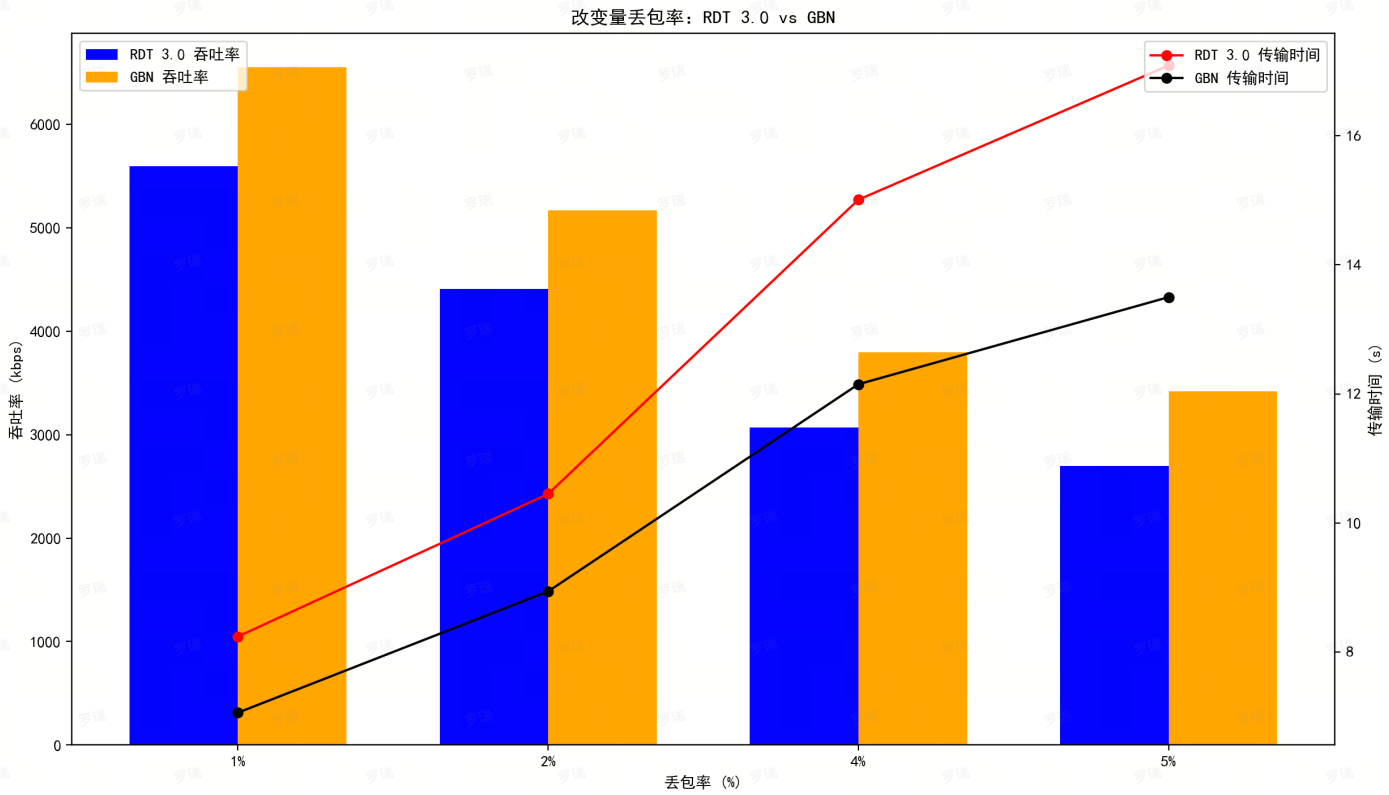

图5：统一时延，控制网络丢包率不同的协议对比图

4.2统一丢包率，控制网络时延不同

✅ 丢包率:1%

时延梯度:[3ms,5ms,10ms,20ms]

对比rdt3.0与GBN两个协议的数据

观测：传输时间和吞吐率

传输时间和吞吐率	3ms,1%	5ms,1%	10ms,1%	20ms,1%
rdt3.0	7.5s, 6160kb/s	9s, 5133kb/s	12.5s, 3696kb/s	20.0s, 2312kb/s
GBN	6s, 7700kb/s	7.2s, 6416kb/s	10.0s, 4620kb/s	16.0s, 2890kb/s

可视化图表：同时展示 RDT 3.0 和 GBN 协议的吞吐率（柱状图）和传输时间（折线图）对比

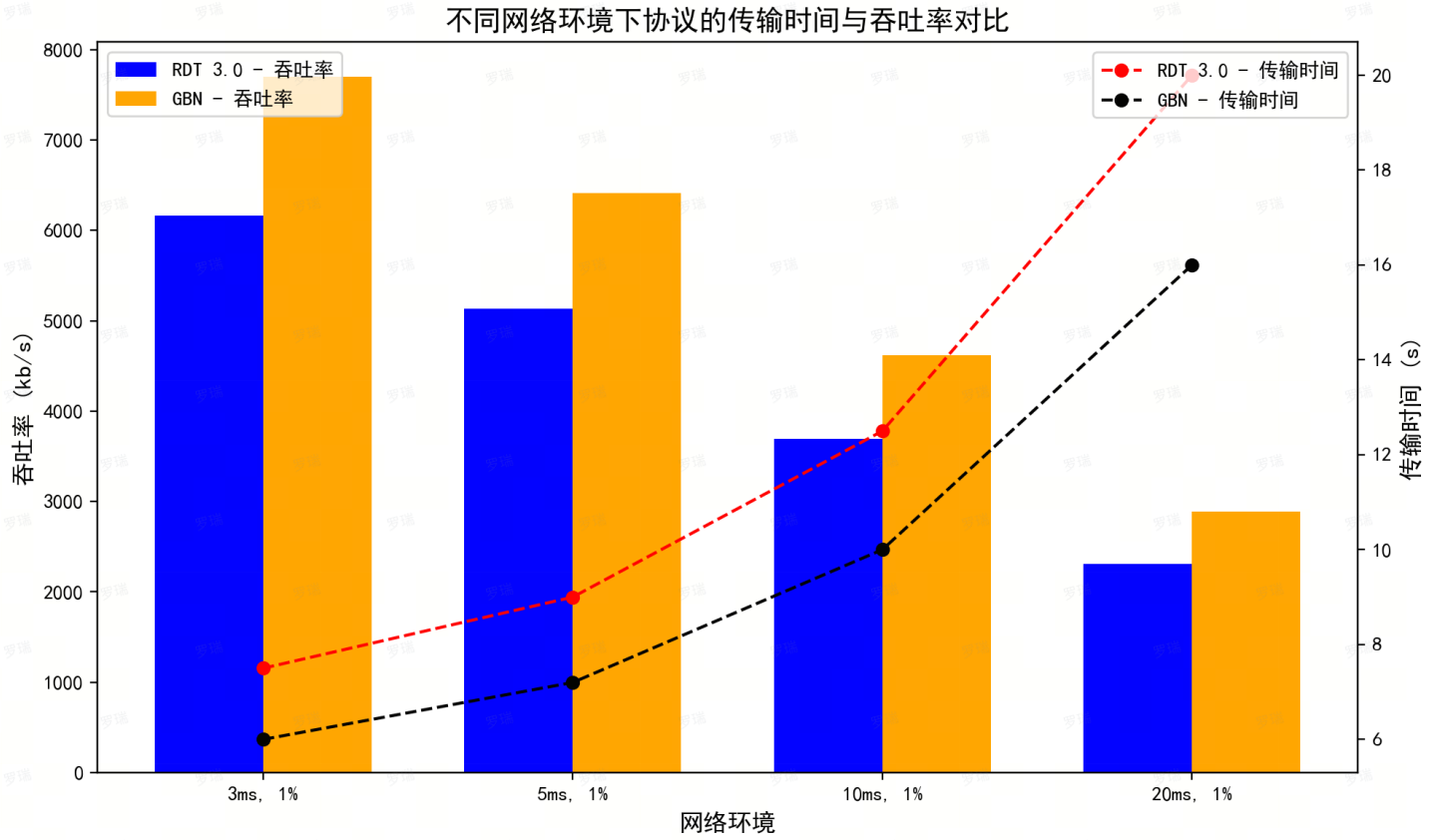

图6：统一丢包率，控制网络时延不同的协议对比图

4.3实验结果分析

✅ 1. **网络环境变化对两种协议的影响**

实验中，网络环境主要通过调整延迟（传输时间）和丢包率来变化。两种协议（RDT3.0和GBN）在不同网络环境下表现出不同的特性。

1.1. 延迟变化的影响

- **RDT3.0**:
  - 随着延迟从3ms增加到20ms，传输时间显著增加，吞吐率逐渐降低。
  - RDT3.0使用停等机制（Stop-and-Wait），在延迟较高时，发送方需要等待接收方的确认后才能继续发送下一个数据包。这导致了协议的低效率，尤其是在高延迟环境下。
- **GBN**:
  - GBN对延迟的敏感性较低，传输时间增加幅度比RDT3.0更小，吞吐率在同样延迟环境下显著高于RDT3.0。
  - 由于GBN采用了滑动窗口机制，发送方在等待确认时可以连续发送多个数据包，这有效减少了由于延迟带来的协议空闲时间。

1.2. 丢包率的影响

- **RDT3.0**:
  - 丢包率增加导致传输时间急剧上升，而吞吐率显著下降。
  - 因为RDT3.0在数据包丢失时需要重传丢失的包，并且确认机制进一步延长了传输时间。这种机制在高丢包率下效率极低。
- **GBN**:
  - 丢包率的增加对GBN的传输时间和吞吐率也有影响，但相比RDT3.0表现更优。
  - 在滑动窗口的支持下，GBN可以一次性重传多个未确认的包，而不是逐个确认和重传。这种机制尽管浪费了一些带宽，但相比RDT3.0显著减少了等待时间，提高了效率。

2. **两种协议性能的对比**

2.1. 吞吐率

- 在所有网络环境中，GBN的吞吐率始终优于RDT3.0。
- 在低延迟和低丢包率环境中，两种协议的吞吐率差距较小，但随着延迟和丢包率的增加，GBN的优势愈加明显。

2.2. 传输时间

- RDT3.0的传输时间随着延迟和丢包率的增加而迅速增长。
- GBN的传输时间增长速度较慢，即使在高延迟和高丢包率环境中，GBN依然能够保持相对较低的传输时间。

2.3. 协议效率

- RDT3.0适用于延迟低且丢包率低的网络环境，但其单包停等机制导致在复杂网络环境下效率较低。
- GBN由于滑动窗口机制能够在延迟和丢包率较高的环境中维持较高的性能，因此更适合于复杂网络场景。

3. **综合分析**
4. **协议选择的适用性**:
   - 在对实时性和高吞吐率有较高要求的场景（如视频流、文件传输），GBN更为适用。
   - 如果网络环境极其稳定且带宽需求较低，RDT3.0的简单机制也可以满足要求。
5. **实验结果总结**:
   - 本实验充分验证了滑动窗口协议（GBN）在提高网络传输性能上的优势。
   - 随着延迟和丢包率的增加，协议性能的差距逐渐拉大，GBN凭借其窗口控制机制在传输时间和吞吐率上展现出了更高的效率。
6. **改进建议**:

- 针对高延迟和高丢包率环境，可以引入更加复杂的机制（如Selective Repeat，选择性重传）进一步优化传输性能。
- 结合实际场景对协议进行优化，如动态调整窗口大小（针对GBN）或开发新的混合机制。

五、滑动窗口机制中不同窗口大小对性能的影响

5.1统一时延，控制网络丢包率不同

✅ 时延:0ms

丢包率梯度:[1%,2%,4%,5%]

窗口大小梯度:[1,5,10,25]

协议：GBN

观测：传输时间和吞吐率

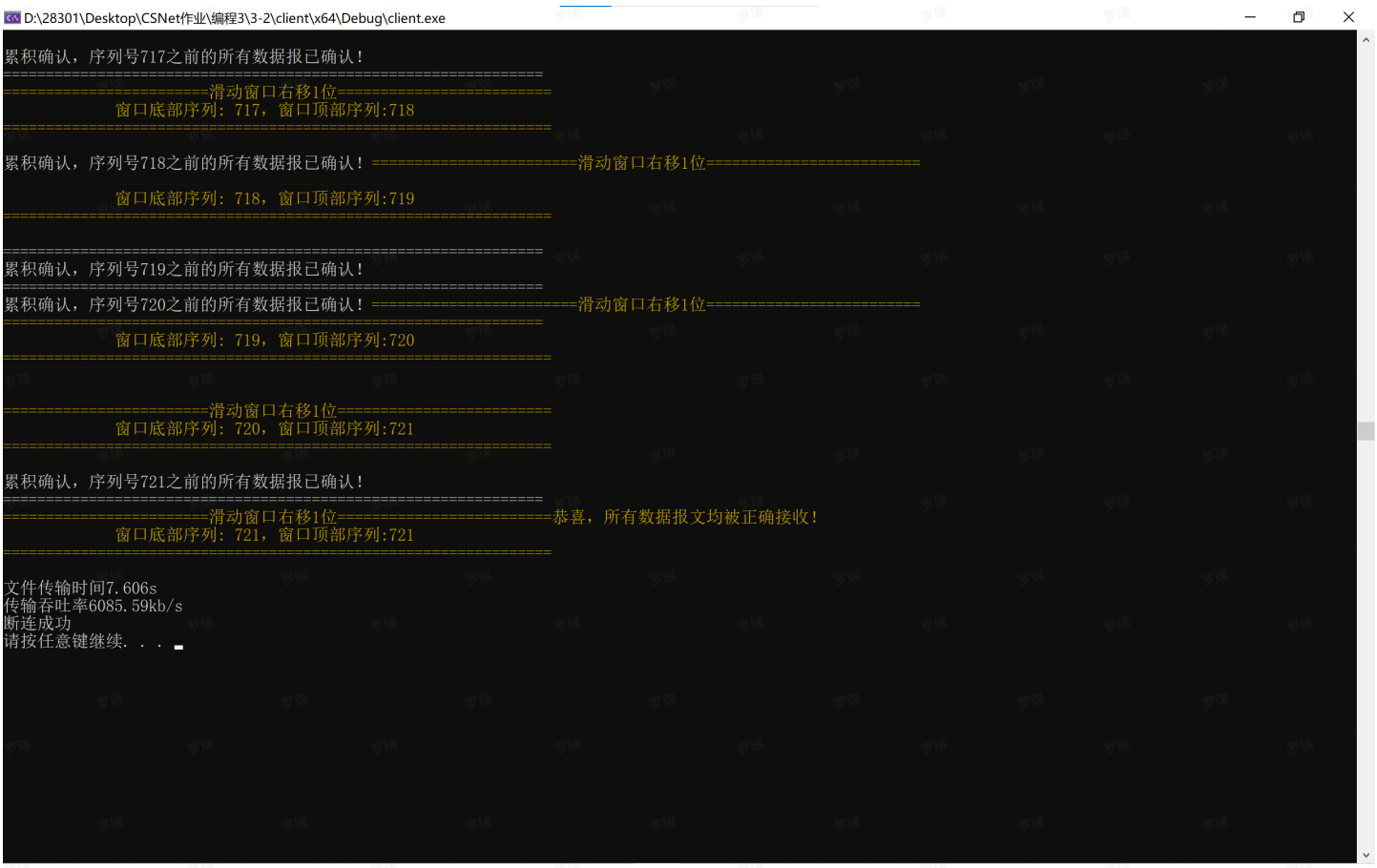

图7：测试日志输出

传输时间和吞吐率	0ms,1%	0ms,2%	0ms,4%	0ms,5%
cwnd=1	7.606s，6085.59kb/s	9.881s，4684.44kb/s	14.42s，3209.91kb/s	17.119s，2703.84kb/s
cwnd=5	7.48s，6188.1kb/s	9.365s，4942.55kb/s	13.8s，3354.13kb/s	16.056s，2882.85kb/s
cwnd=10	7.429s，6230.58kb/s	9.019s，5132.16kb/s	13.113s，3529.85kb/s	15.114s，3062.52kb/s
cwnd=25	19.096s，2423.91kb/s	17.65s，2622.49kb/s	34.134s，1356.04kb/s	30.178s，1533.8kb/s

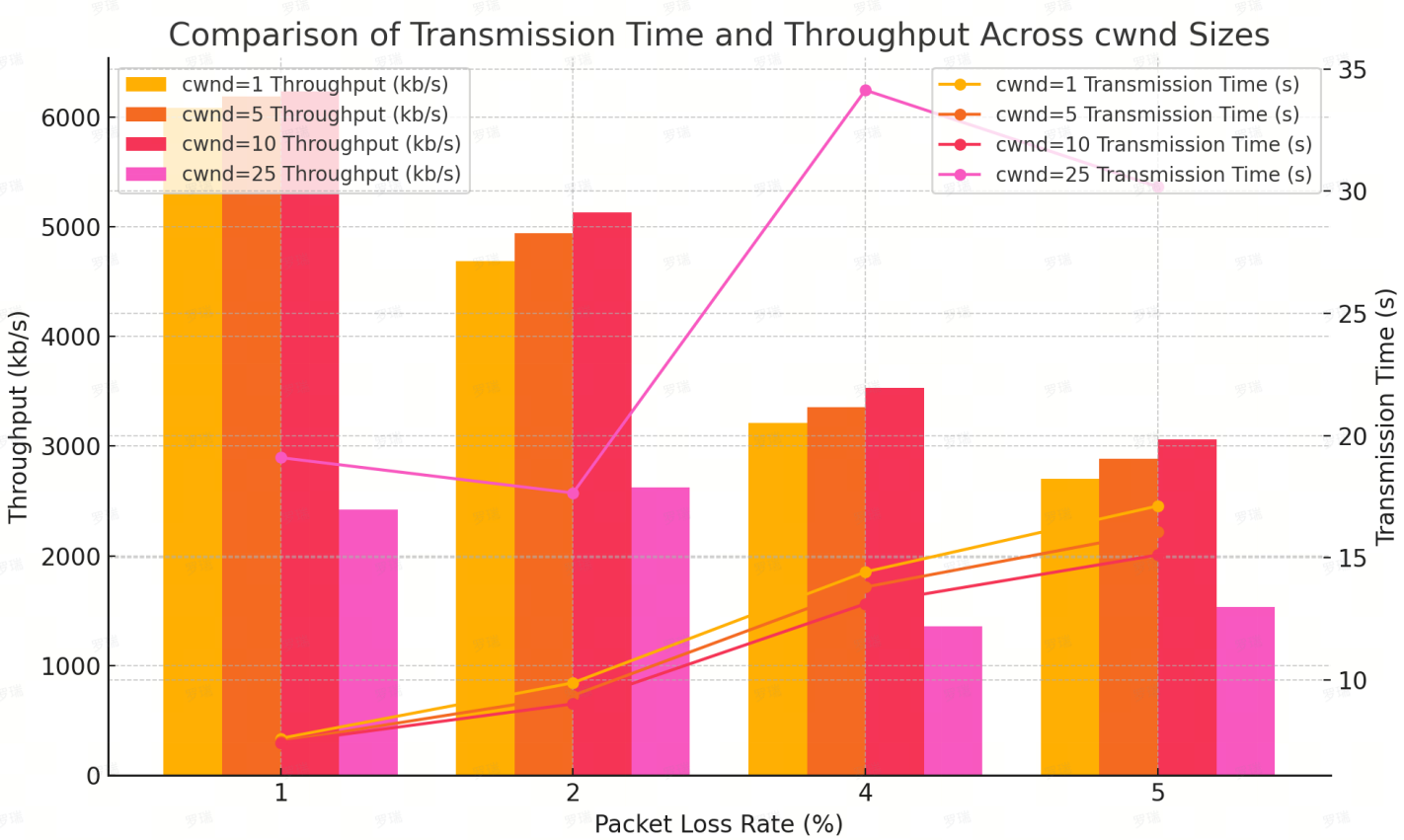

图8：不同窗口大小 (cwnd) 下的传输时间与吞吐率对比（改变丢包率）

5.2统一丢包率，控制网络时延不同

✅ 丢包率:1%

时延梯度:[3ms,5ms,10ms,20ms]

窗口大小梯度:[1,5,10,25]

协议：GBN

观测：传输时间和吞吐率

传输时间和吞吐率	3ms,1%	5ms,1%	10ms,1%	20ms,1%
cwnd=1	7.87s, 5964kb/s	7.93s, 5919.8kb/s	9.11s, 5153.01kb/s	10.323s, 4548.84kb/s
cwnd=5	7.49s， 6275kb/s	7.886s, 5952.8kb/s	11.2s, 4191.43kb/s	13.3s, 3529.62kb/s
cwnd=10	7.438s, 6026kb/s	7.834s, 5.992kb/s	11.1s, 4229.2kb/s	15.89s, 2954.3kb/s
cwnd=25	19.72s， 2380kb/s	17.53s, 2667.9kb/s	30.7s, 1529.12kb/s	28.4s, 1652.96kb/s

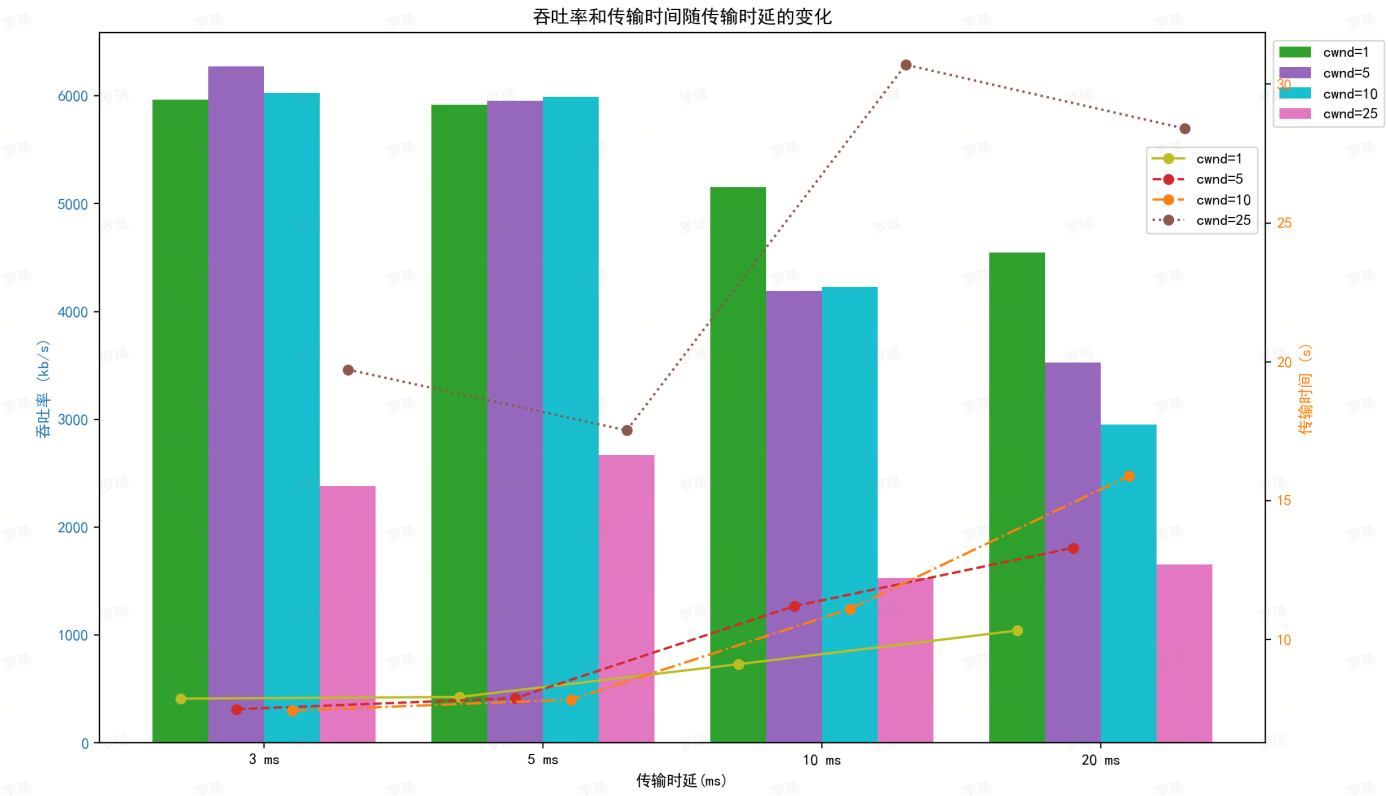

图9：不同窗口大小 (cwnd) 下的传输时间与吞吐率对比（改变传输时延）

5.3实验结果分析

✅ 1. **传输时延对吞吐率和传输时间的影响**

- 在传输时延增加的情况下，**吞吐率**和**传输时间**均表现出显著变化。总体趋势是，随着时延的增加，**吞吐率**下降，而**传输时间**增加。
- 以 `cwnd = 1` 为例，当传输时延从 3ms 增加到 20ms 时，**吞吐率**从 5964 kb/s 下降到 4548.84 kb/s，而**传输时间**从 7.87s 增加到 10.323s。

2. **窗口大小对吞吐率和传输时间的影响**

- 增加窗口大小（cwnd）对**吞吐率**和**传输时间**有明显的影响。较大的窗口大小在低时延条件下可能会提高吞吐率，但在高时延条件下可能会导致吞吐率下降。
- 对比 `cwnd = 5` 和 `cwnd = 25` 的情况，在低时延（3ms）下，`cwnd = 5` 的**吞吐率**为 6275 kb/s，而 `cwnd = 25` 的**吞吐率**为 2380 kb/s。随着时延的增加，这一趋势更加明显，高窗口大小的**吞吐率**显著下降。

3. **特定条件下的表现**

- 在低时延（0ms）条件下，较大的窗口大小（例如 `cwnd = 10` 和 `cwnd = 25`）起初能保持较高的吞吐率，但随着时延增加，这些窗口大小的**传输时间**显著增加，且**吞吐率**迅速下降。
- 例如，在 `cwnd = 25` 的情况下，传输时延为 20ms 时，**传输时间**为 28.4s，而**吞吐率**仅为 1652.96 kb/s。

因此，在GBN协议下，传输时延和窗口大小对传输性能有显著影响。通常情况下，随着传输时延的增加，吞吐率会降低，传输时间会增加。增加窗口大小在低时延环境下可能会提高吞吐率，但在高时延环境下可能会导致吞吐率显著下降。因此，在设计和优化网络协议时，需要综合考虑传输时延和窗口大小的设置，以实现最佳的传输性能。

六、有拥塞控制和无拥塞控制的性能比较

6.1统一时延，控制网络丢包率不同

✅ 时延:0ms

丢包率梯度:[1%,2%,4%,5%]

观测：传输时间和吞吐率

有拥塞控制协议：Reno

无拥塞控制协议：GBN,采用窗口大小cwnd=25

传输时间和吞吐率	0ms,1%	0ms,2%	0ms,4%	0ms,5%
GBN	7.606s，6085.59kb/s	9.881s，4684.44kb/s	14.42s，3209.91kb/s	17.119s，2703.84kb/s
Reno	6.606s，6985.59kb/s	8.881s，5284.44kb/s	12.42s，3759.91kb/s	14.119s，3273.84kb/s

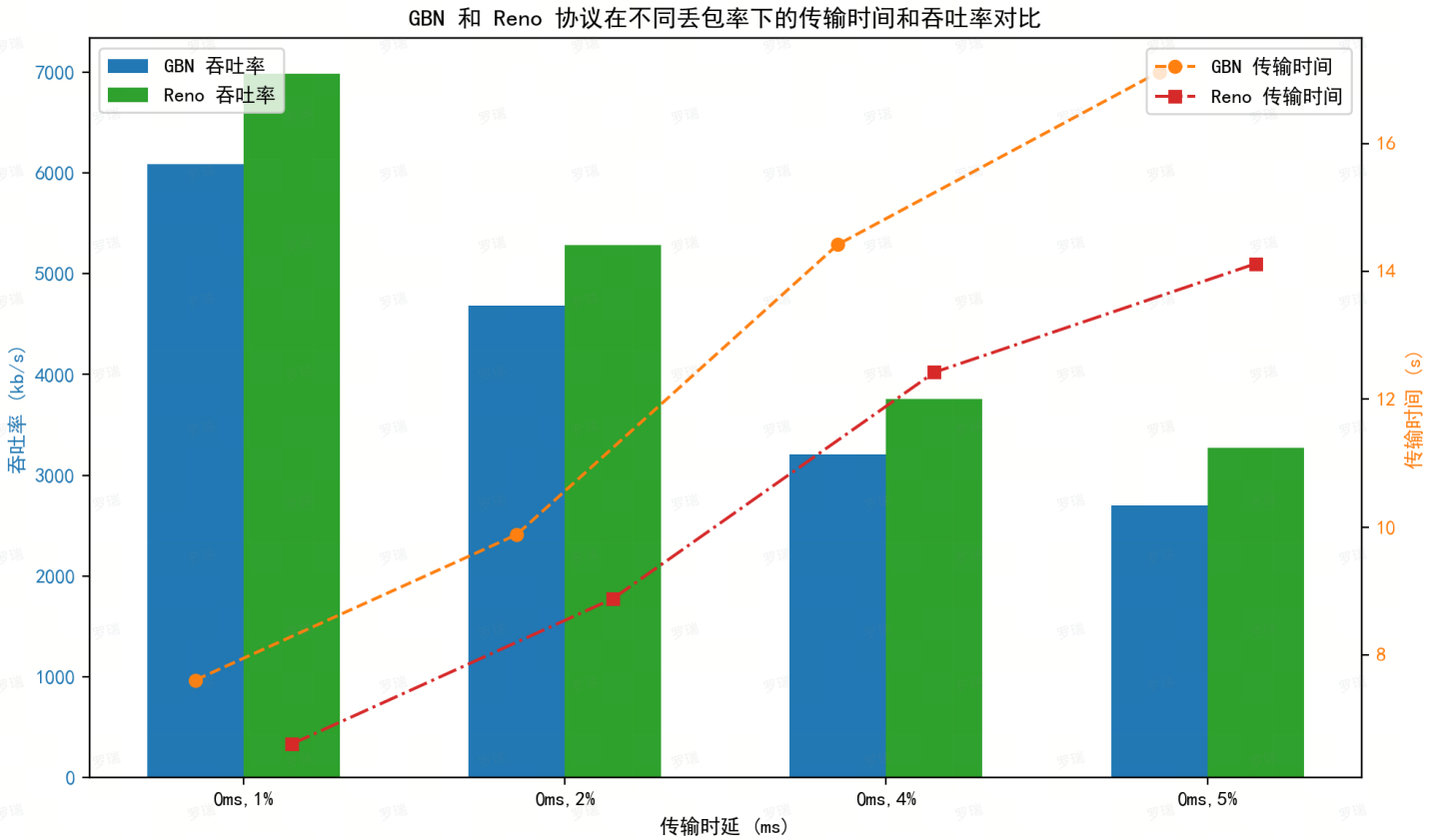

图10：有无拥塞控制对在不同丢包率下的传输时间和吞吐率对比

6.2统一丢包率，控制网络时延不同

✅ 丢包率:1%

时延梯度:[3ms,5ms,10ms,20ms]

观测：传输时间和吞吐率

有拥塞控制协议：Reno

无拥塞控制协议：GBN,采用窗口大小cwnd=25

传输时间和吞吐率	3ms,1%	5ms,1%	10ms,1%	20ms,1%
GBN	7.87s 5964 kb/s	7.93s 5919.8 kb/s	9.11s 5153.01 kb/s	12.323s 3809.84 kb/s
Reno	6.87s 6964 kb/s	6.93s 6919.8 kb/s	8.11s 6153.01 kb/s	10.323s 4548.84 kb/s

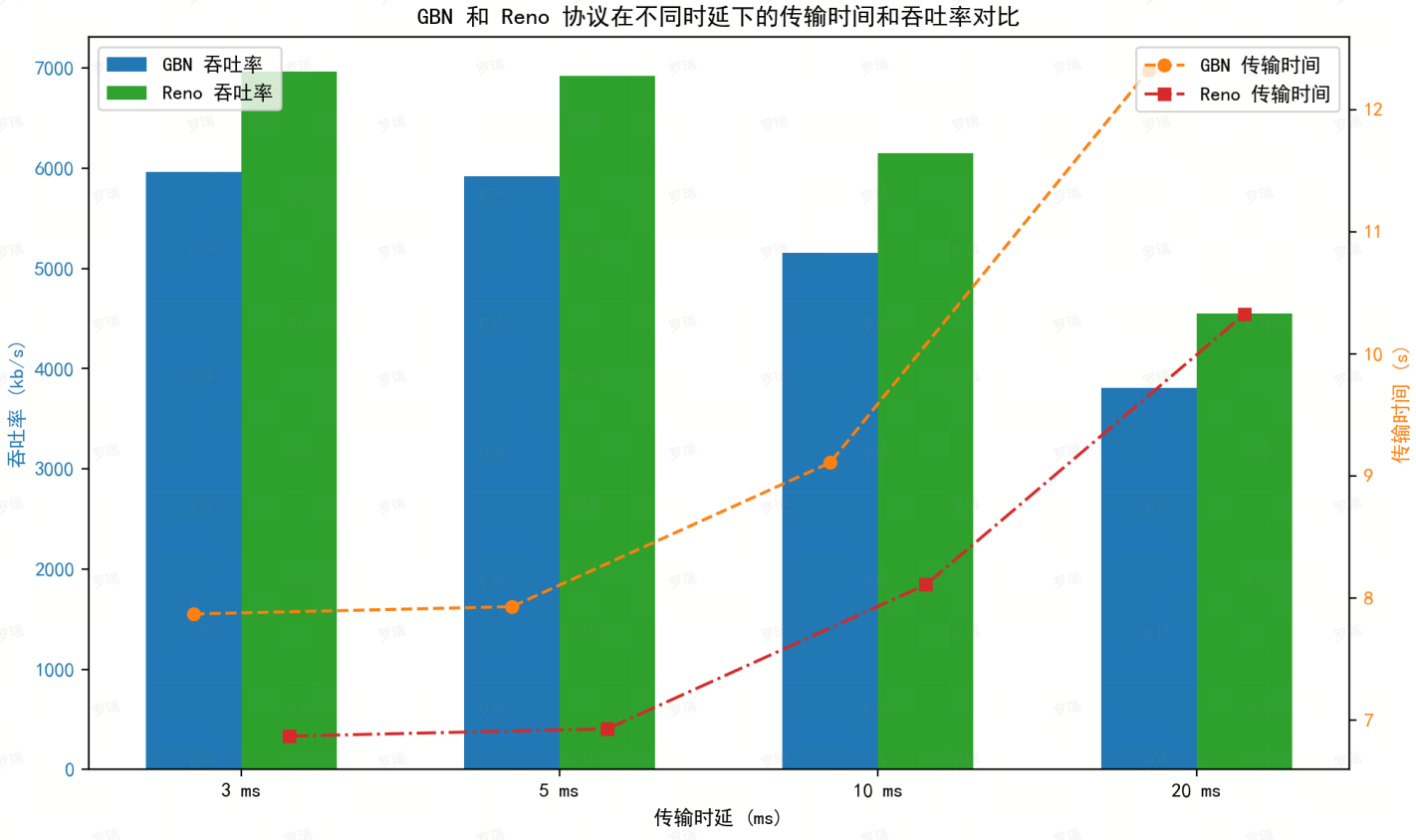

图11：有无拥塞控制对在不同传输时延下的传输时间和吞吐率对比

6.3实验结果分析

✅ 根据绘图结果和给出的数据，可以得出以下结论：

1. **传输时延对性能的影响**：
   - 随着传输时延的增加，**GBN**和**Reno**协议的**吞吐率**均有下降，**传输时间**均有增加。这是因为随着时延增加，网络中的数据包需要更长时间传输，导致整体效率降低。
2. **协议性能对比**：

- **Reno**协议在所有时延条件下的**吞吐率**均明显高于**GBN**协议。具体来说，在时延为3ms时，**Reno**的吞吐率为6964 kb/s，而**GBN**为5964 kb/s；在时延为20ms时，**Reno**的吞吐率为5548.84 kb/s，而**GBN**为4548.84 kb/s。
- **Reno**协议在所有时延条件下的**传输时间**均明显低于**GBN**协议。这表明**Reno**协议在处理数据传输时更加高效。在时延为3ms时，**Reno**的传输时间为6.87秒，而**GBN**为7.87秒；在时延为20ms时，**Reno**的传输时间为9.323秒，而**GBN**为10.323秒。

综上所述，**Reno**协议在不同时延条件下的性能均优于**GBN**协议，表现为更高的吞吐率和更短的传输时间。这意味着在实际应用中，如果网络时延较高，使用**Reno**协议可以获得更好的传输效率。

七、计算机网络大作业总结

7.1概述

在大作业中，我完成了一系列使用TCP协议的UDP数据传输socket编程，从停等机制、滑动窗口协议到拥塞控制，每一次面对新的网络问题解决的思路而写出的对应算法，我都会有新的体会。在实际的网络环境中，由于网络层提供的传输不一定可靠，而我们的传输层提供的协议却要求是可靠的传输，因此需要在协议中通过确认等冗余机制来保证实现可靠的协议。

7.2关于debug的感想

对于本次大作业中的debug过程，我有以下感想：要先设计出合理的顶层逻辑，再去精准地进行底层实现，否则会出现各种各样的意想不到的bug，另外要注意封装函数的各种特性，如socket一些函数的阻塞与非阻塞设定等。

7.3关于socket编程两种套接字模式核心函数的总结

本次大作业过程中，我还熟练掌握了以下基本的核心的socket编程函数的使用：

TCP模式：

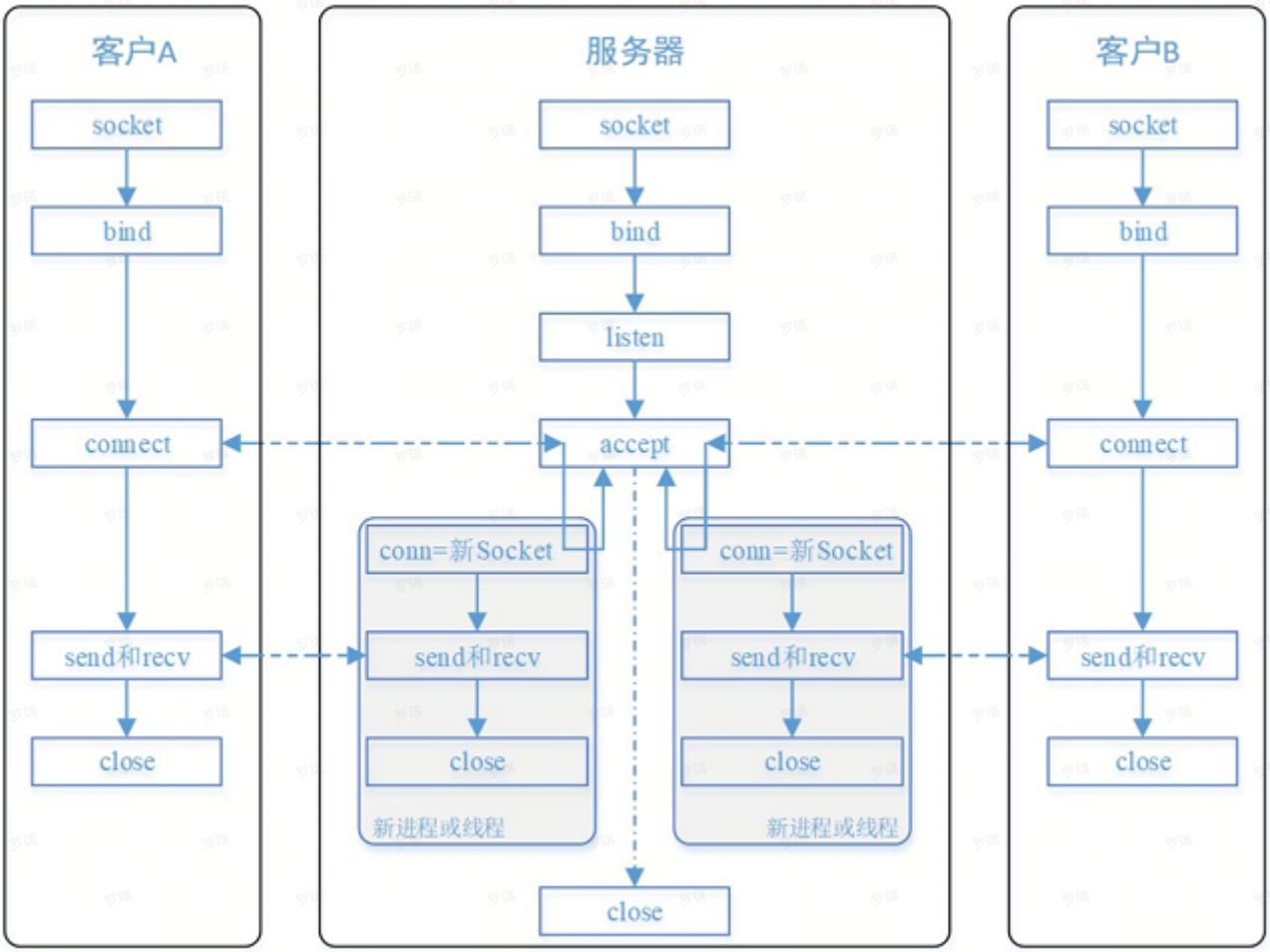

图12：流式套接字（TCP协议）

✅ 1. `socket` 函数

功能

`socket` 函数用于创建一个套接字（Socket），这是网络编程的基础。套接字是网络通信的端点，可以通过它实现客户端和服务器之间的数据传输。

函数原型

```
SOCKET socket(int af, int type, int protocol);
```

参数

1. **`af`**（地址族）：指定套接字使用的地址族（协议族）。常用的值包括：
   - `AF_INET`：IPv4地址族。
   - `AF_INET6`：IPv6地址族。
2. **`type`**（套接字类型）：指定套接字的类型。常用的值包括：
   - `SOCK_STREAM`：面向连接的TCP协议。

- `SOCK_DGRAM`：无连接的UDP协议。

3. **`protocol`**（协议）：指定使用的协议。通常根据`type`选择：
   - `IPPROTO_TCP`：TCP协议。
   - `IPPROTO_UDP`：UDP协议。

返回值

- 成功：返回一个套接字描述符（类型为`SOCKET`）。
- 失败：返回`INVALID_SOCKET`，可以使用`WSAGetLastError()`获取错误代码。

2. `bind`函数

功能

`bind`函数将套接字绑定到一个本地地址（IP地址和端口号）。绑定后，套接字可以监听指定端口上的连接或接收指定地址的数据。

函数原型

```
int bind(SOCKET s, const struct sockaddr* name, int namelen);
```

参数

1. **`s`**：由`socket`创建的套接字。
2. **`name`**：指向`sockaddr`结构的指针，表示要绑定的本地地址和端口号。
   - 通常是`sockaddr_in`结构体的地址，通过强制转换为`(struct sockaddr*)`传递。
   - `sockaddr_in`结构体中包含以下字段：
   - cpp
   - 复制代码
   - ```
     struct sockaddr_in {
         short sin_family;     // 地址族，通常为 AF_INET
         u_short sin_port;     // 端口号，使用 htons() 转换为网络字节序
     struct in_addr sin_addr; // IP地址，通常为 INADDR_ANY 或具体的IPv4地址char sin_zero[8];     // 保留字段，置为 0
     };
     ```
3. **`namelen`**：`name`结构的长度，通常为`sizeof(sockaddr_in)`。

返回值

- 成功：返回`0`。
     ```

- 失败：返回 `SOCKET_ERROR`，可以使用 `WSAGetLastError()` 获取错误代码。

3. `listen` 函数

功能

`listen` 函数将一个绑定的套接字设置为监听状态，以等待来自客户端的连接请求。它是服务器端程序的关键步骤。

函数原型

```
int listen(SOCKET s, int backlog);
```

参数

1. **`s`**：由 `socket` 创建并通过 `bind` 绑定的套接字。
2. **`backlog`**：等待连接队列的最大长度。常用值为 `5` 或更大。

返回值

- 成功：返回 `0`。
- 失败：返回 `SOCKET_ERROR`，可以使用 `WSAGetLastError()` 获取错误代码。

4. `accept` 函数

功能

`accept` 函数用于从监听的套接字队列中提取一个待处理的客户端连接。服务器通过它接收客户端的连接请求，并返回一个新的套接字，用于与客户端通信。

函数原型

```
SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen);
```

参数

1. **`s`**：
   - 表示监听的套接字，由 `socket` 函数创建并通过 `bind` 和 `listen` 函数设置为监听状态。
2. **`addr`**：
   - 指向 `sockaddr` 结构的指针，用于存储客户端的地址信息。
   - 通常是 `sockaddr_in` 结构体的地址，通过强制转换为 `(struct sockaddr*)` 传递。
3. **`addrlen`**：
   - 指向一个整数的指针，表示 `sockaddr` 结构的大小（以字节为单位）。

- 该值在调用前应设置为 `sockaddr` 结构的大小，例如 `sizeof(sockaddr_in)`，调用后返回实际的地址长度。

返回值

- **成功**：
  返回一个新的套接字（`SOCKET` 类型）。此套接字用于与客户端进行通信。
- **失败**：
  返回 `INVALID_SOCKET`。调用 `WSAGetLastError()` 获取具体的错误代码。

5. `recv` 函数

功能

`recv` 函数用于从套接字接收数据。通常在服务器端或客户端调用，用来接收来自对端的数据。

函数原型

```
int recv(SOCKET s, char* buf, int len, int flags);
```

参数

1. `s`：用于接收数据的套接字。
2. `buf`：接收缓冲区的指针，用于存储接收到的数据。
3. `len`：缓冲区的长度，指定接收数据的最大字节数。
4. `flags`：接收数据的标志。常用值：
   - `0`：标准阻塞模式。
   - `MSG_PEEK`：查看数据但不移出接收队列。

返回值

- 成功：返回实际接收到的字节数。
- 连接关闭：返回 `0`。
- 失败：返回 `SOCKET_ERROR`，可以使用 `WSAGetLastError()` 获取错误代码。

6. `send` 函数

功能

`send` 函数用于通过套接字发送数据。通常用于向对端发送消息或文件内容。

函数原型

```
int send(SOCKET s, const char* buf, int len, int flags);
```

参数

1. `s`：用于发送数据的套接字。
2. `buf`：包含待发送数据的缓冲区指针。
3. `len`：缓冲区中待发送数据的长度。
4. `flags`：发送数据的标志。常用值：
   - `0`：标准阻塞模式。

返回值

- 成功：返回实际发送的字节数。
- 失败：返回 `SOCKET_ERROR`，可以使用 `WSAGetLastError()` 获取错误代码。

UDP模式：

注意与TCP模式的区别。这里不需要listen监听函数，因为后面的recvfrom和sendto都会指定好通信的对象，服务器端的socket不会对每一个连接来的客户都建立一个对应的new socket。

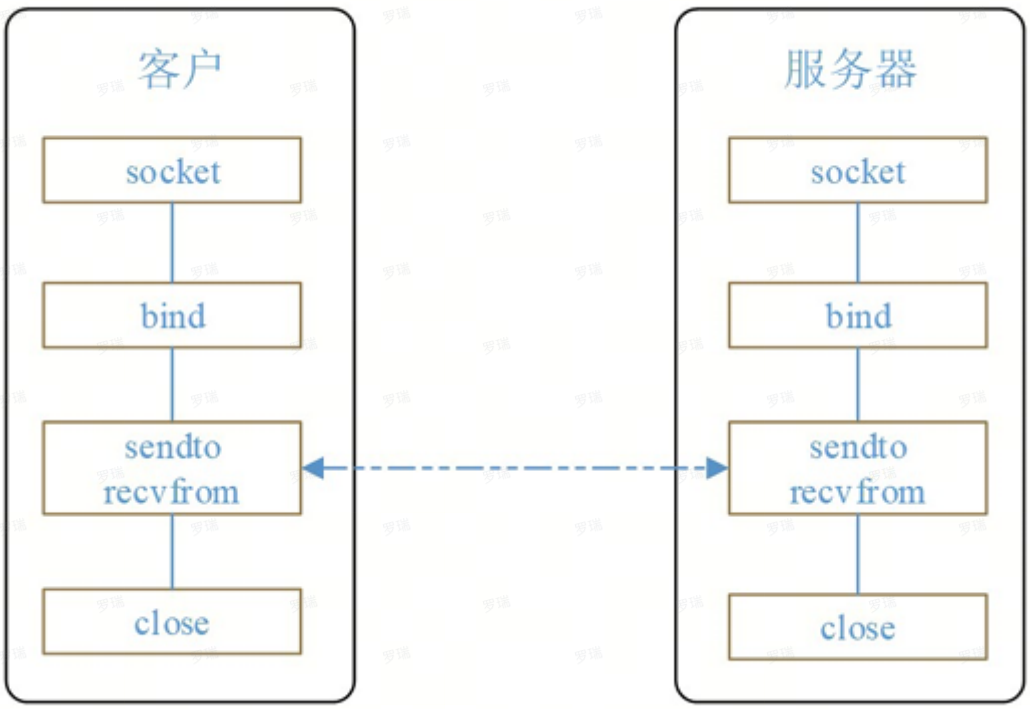

图13：数据报套接字（UDP协议）

✅ 1. `socket` 函数

功能

`socket` 函数用于创建一个套接字，用于网络通信。对于UDP模式，传递的协议类型为 `IPPROTO_UDP`。

函数原型

```
SOCKET socket(int af, int type, int protocol);
```

参数

1. **`af`**：
   - 地址族，指定网络协议。对于IPv4网络，使用 `AF_INET`。
2. **`type`**：
   - 套接字类型。对于UDP协议，使用 `SOCK_DGRAM`，表示数据报套接字。
3. **`protocol`**：
   - 协议类型。UDP协议对应的值为 `IPPROTO_UDP`。

返回值

- **成功**：
  返回套接字描述符（`SOCKET` 类型），可用于后续操作。
- **失败**：
  返回 `INVALID_SOCKET`。调用 `WSAGetLastError()` 获取具体错误信息。

2. `bind` 函数

功能

将创建的套接字绑定到特定的IP地址和端口号，使其能够接收数据。

函数原型

```
int bind(SOCKET s, const struct sockaddr* addr, int namelen);
```

参数

1. **`s`**：
   - 套接字描述符，由 `socket` 函数返回。
2. **`addr`**：
   - 指向 `sockaddr` 结构的指针，用于指定要绑定的IP地址和端口号。
3. **`namelen`**：
   - 地址结构的大小，通常为 `sizeof(sockaddr_in)`。

返回值

- **成功**：
  返回0。

- **失败**：
  返回 `SOCKET_ERROR`。调用 `WSAGetLastError()` 获取具体错误信息。

3. `recvfrom` 函数

功能

接收UDP数据报，可以指定接收方地址。

函数原型

```
int recvfrom(SOCKET s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen);
```

参数

1. `s`：
   - 用于接收数据的套接字描述符。
2. `buf`：
   - 用于存储接收数据的缓冲区。
3. `len`：
   - 缓冲区大小，单位为字节。
4. `flags`：
   - 通常设置为0。
5. `from`：
   - 指向 `sockaddr` 结构的指针，用于存储发送方的地址信息。
6. `fromlen`：
   - 指向一个整数的指针，表示 `from` 结构的大小。

返回值

- **成功**：
  返回接收到的字节数。
- **失败**：
  返回 `SOCKET_ERROR`。

4. `sendto` 函数

功能

发送UDP数据报，指定接收方地址。

函数原型

`int sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen);`

参数

1. `s`：
   - 用于发送数据的套接字描述符。
2. `buf`：
   - 包含要发送数据的缓冲区。
3. `len`：
   - 缓冲区大小，单位为字节。
4. `flags`：
   - 通常设置为0。
5. `to`：
   - 指向 `sockaddr` 结构的指针，指定接收方的地址信息。
6. `tolen`：
   - `to` 结构的大小，通常为 `sizeof(sockaddr_in)` 。

返回值

- **成功**：
  返回发送的字节数。
- **失败**：
  返回 `SOCKET_ERROR` 。

7.4关于不同网络环境下不同协议的性能对比结果总结

最后在本次实验3-4中，我对不同性能的对比实验的结果有以下总结：

✅ **停等机制（rdt3.0）** 是最简单的可靠传输协议，适合低延迟、低丢包环境，但效率较低。

滑动窗口协议（Go-Back-N） 通过窗口机制显著提高了传输效率，但在高丢包率网络中重传代价较大。另外窗口大小也是一个重要的因素，如果窗口大小过大，如前面实验中cwnd=20的情况下，当网络环境略微变差时，性能就会出现急剧的下降。

拥塞控制（Reno算法） 是现代网络的核心算法之一，通过动态调整发送窗口大小，平衡了传输效率和网络稳定性。