
南开大学
Nankai University

南　　开　　大　　学

网　络　空　间　安　全　学　院

**计算机网络实验报告**

---

# 实验 3-4：基于 UDP 服务设计可靠传输协议并编程实现

---

姓名：王峥

学号：2211267

专业：信息安全

指导教师：吴英

2024 年 12 月 18 日

# 目录

# 一、 实验要求与实现目标

## (一) 实验要求

实验 3-4：基于给定的实验测试环境，通过改变网络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) 实现目标

本次实验将会基于前三次实验进行详细的性能分析，探究不同机制的区别，并尝试详细分析其背后的原理。

# 二、 前期实验完成概述

回顾之前的三次实验，我们已经分别实现二元序号的停等机制、多序号的流水线协议进行流量控制，并在此基础上实现累积确认的回退 N 帧协议 GBN 机制，同时在最后实现了基于 RENO 算法的拥塞控制算法，后续分析将通过课程前期提供的路由器来实现丢包和时延，我们先简单回顾三次实现的重点：

1. **Lab3-1 停等机制**
   - 未实现多线程，**超时重传机制主要是通过 while(true)+recv_from+ 非阻塞模式实现的**。
   - 停等机制主要参考 RDT 3.0 的**二元序号实现**（即 seq 仅有 0 和 1）.
   - 实现了包括**差错检测（检查校验和），接受确认等机制**，这部分机制也继续应用在后续实验中
   - 实现三**次握手**的建立连接机制以及**四次挥手**的断开连接机制
2. **Lab3-2 流水线协议下的 GBN**
   - 流水线协议的实现：流量控制与多序列号（即 **seq 依次递增**）的机制
   - **发送方窗口大小 >1，接收方窗口大小 =1**。累积确认：Go Back N
   - **发送方的多线程**（主线程发送 + 接收线程 + 日志线程）实现数据传输与保序的日志输出，接收方仍然采用单线程
   - 对用共享资源，包括发送缓冲区和发送端日志在多线程的作用下，将采用**锁机制**来避免竞争
3. **Lab3-3 在 Go-Back-N 机制下的 RENO 拥塞控制**
   - 实现**基于 RENO 的发送端动态窗**口，将根据发送情况与接收 ACK 情况动态调整发送的窗口大小
   - **发送方窗口大小将随着发送情况进行改变，接收方窗口大小始终保持为 1**
   - 在收到 3 个重复 ACK 时将会进行**快速重传**从而处理丢包的情况
   - 保持 3-2 下的多线程以及监控开始窗口的时间进行超时判断，并进行超时重传

### (一) 测试标准

- 控制变量法：对比时控制单一变量（算法、窗口大小、延时、丢包率）
- 延时、丢包率对比设置：设置梯度（例如 30ms,50ms, …；5%，10%，…）
- 测试文件：前期实验使用测试文件 1.jpg，发现测试数据的误差率较大（后续会简述），因此本次将主要采用测试文件 2，即总文件大小（包括文件路径）为 5898611Bytes.
- 性能测试指标：时延、吞吐率，会采用图表进行分析。

## 三、 停等机制与滑动窗口机制的性能对比

首先这里测试 Lab03-01 中的停等机制和 Lab03-02 的滑动窗口机制进行性能对比。

### (一) 控制时延为 0，改变丢包率对比停等机制和滑动窗口 GBN 的性能

在 3-2 的实验时我们采用 1.jpg 作为传输数据进行了分析，首先我们回顾一下：

| 停等机制 | | | | | | |
|---|---|---|---|---|---|---|
| 丢包率 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 375 | 2186 | 3549 | 5856 | 7078 | 9032 |
| 吞吐率（Bytes/ms） | 4952.97 | 849.664 | 523.349 | 317.173 | 262.414 | 205.643 |
| 窗口为8 | | | | | | |
| 丢包率 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 972 | 3552 | 3174 | 4970 | 4367 | 7781 |
| 吞吐率（Bytes/ms） | 1910.87 | 522.907 | 585.181 | 373.715 | 425.318 | 238.705 |
| 窗口为18 | | | | | | |
| 丢包率 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 2194 | 3558 | 3896 | 4367 | 4633 | 7534 |
| 吞吐率（Bytes/ms） | 846.566 | 522.025 | 476.736 | 425.318 | 400.899 | 246.531 |
| 窗口为30 | | | | | | |
| 丢包率 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 2431 | 3278 | 3541 | 5104 | 5352 | 7683 |
| 吞吐率（Bytes/ms） | 764.033 | 566.615 | 524.531 | 363.904 | 347.041 | 241.75 |

图 1: 停等机制和不同窗口大小的滑动窗口机制性能对比

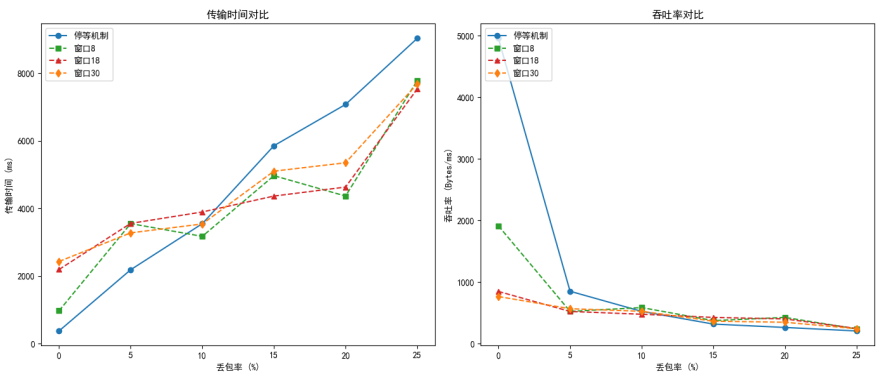


图 2: 停等机制和不同窗口大小的滑动窗口机制性能对比

1. **丢包率为 0**

我们看到丢包率为 0 或较低的时候，停等机制明显优于 Go-Back-N，我们猜测，一个是网络带宽受限，同时主要是我在实现过程中的一系列额外操作包括日志锁，时间锁等问题，而停等机制的实现当初非常简洁，RDT3.0 和双序列号加上较少的日志输出和单线程编程，反而在丢包率较低的时候性能更好。

2. **丢包率 >10**

我们观察到随着丢包率的不断攀升，停等机制的性能大幅度下降，但我们看到在滑动窗口的性能优于停等机制。按理说由于丢包率较大，因此会出现大量的重复 ACK，导致超时重传，重传的时候会将窗口中的所有数据包全部重新发送，一方面会占用带宽资源，另一方面很有可能再次出现丢包，所以理论上应该在丢包率高的时候，我们的滑动窗口性能降低。**实现方法而言**由于停等机制没有采用多线程，因此对于超时重传的处理可能会比较缓慢（**采用的是 while(true)+ 非阻塞模式 +recv_from**），而对于**超时重传实现采用了多线程的 GBN** 来说，当超时重传多次出现，可能会带来优势。

由于我在之前实验的时候发现路由器控制丢包的稳定性一般，所以实验结果会存在一些偶然性，因此接下来我们使用 2.jpg 再进行测试，进行验证，本次测试我们直接控制窗口大小为 16：

| 停等机制 | | | | | | |
|---|---|---|---|---|---|---|
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 2336 | 7491 | 10981 | 17414 | 21405 | 27443 |
| 吞吐率（Bytes/ms） | 2525.05 | 787.414 | 537.157 | 338.723 | 275.567 | 214.937 |
| 窗口大小为16的滑动窗口机制 | | | | | | |
| 丢包 | 0 | 5 | 10 | 15（17） | 20 | 25 |
| 传输时间（ms） | 8009 | 11726 | 12055 | 15064 | 14935 | 23929.1 |
| 吞吐率（Bytes/ms） | 736.486 | 503.029 | 489.3 | 391.564 | 394.946 | 246.531 |

图 3: 对比结果 1

以下是生成的对比图象：

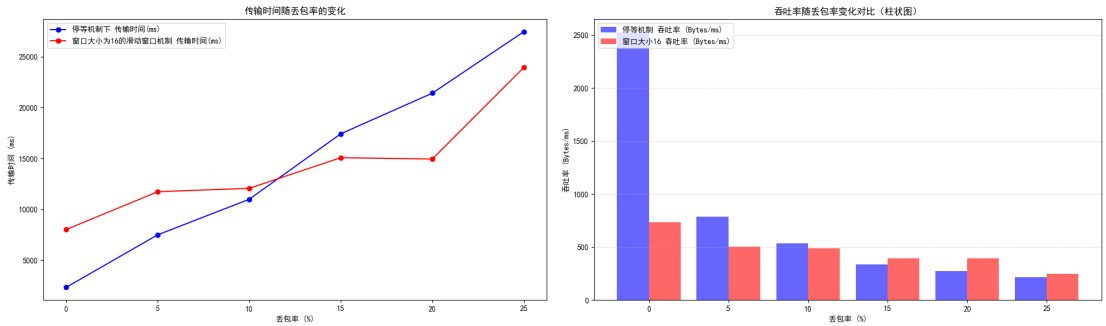


图 4: 传输时间与吞吐率对比分析

还要再说明一下，因为我使用的是助教提供的丢包工具，这会造成我在大于 25% 后（我们本来想测试 30%）会出现重复丢包的情况，即超时重传后丢的包还是那个之前丢失的包，所以结果会有很大的误差，因此本次我还是检测到了 25% 的情况，因为从已有的数据我们就能看到对应的情况。

**结论**

- 丢包率较小（小于 10）：**停等机制明显优于 Go-Back-N**
  - 网络带宽受限

  - 日志线程的引入以及剩余窗口内容的显示（这个需要遍历整个容器）
  - 多线程后为了避免死锁问题，加入的计时器 Timer 类和 Buffer 类在操作前后都会加锁

- 丢包率增大（大于 10）：**GBN 的性能**
  - 停等机制没有采用多线程,因此对于超时重传的处理可能会比较缓慢(采用的是 while(true)+非阻塞模式 +recv_from)
  - GBN 使用多线程实现超时重传，当超时重传多次出现，可能会带来优势。

## (二) 控制丢包率为 0，改变时延对比停等机制和滑动窗口 GBN 的性能

这里我们还是像上次一样选用滑动窗口为 16 的窗口进行比较:

| 停等机制 | | | | | | |
|---|---|---|---|---|---|---|
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 1006 | 14870 | 25512 | 32829 | 45683 | 59841 |
| 吞吐率（Bytes/ms） | 5863.34 | 396.672 | 231.206 | 179.674 | 129.118B | 98.5698 |
| 窗口大小为16的滑动窗口机制 | | | | | | |
| 丢包 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 8016 | 49472 | 122035 | 199651 | 276060 | 517933 |
| 吞吐率（Bytes/ms） | 735.843 | 119.229 | 48.3346 | 29.5441 | 21.3668 | 11.3886 |

图 5: 改变时延对应实验结果

我们首先分别分析:

1. **停等机制：**

   随着延时的增加，停等机制的传输时间明显增加，而吞吐率逐渐下降。这是因为停等机制需要在发送一个数据包后等待确认，较大的延时导致等待时间增加，降低了吞吐率。

2. **滑动窗口机制**

   时延为 0 或很小的时候，我们看到其性能较好，但随着时延增大，我们发现其性能降低的很快，因为时延越大，需要等待超时才能重传，极大的影响效率。

我们综合对比分析结果:

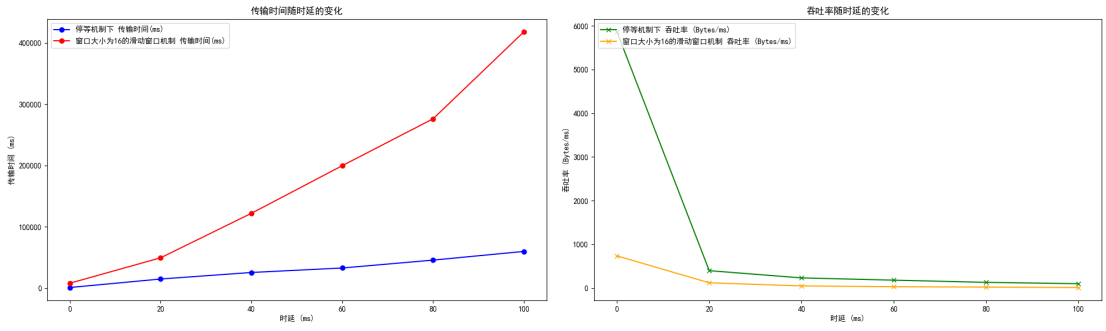


图 6: 传输时间与吞吐率对比分析

我们发现很不符合常理的结果，我们看到时延低的时候二者的性能都不错，但是反而是停等机制效率更好，这个不难理解，毕竟我们在实现滑动窗口采用多线程的时候，为避免共享资源混乱多次使用互斥锁，这将会降低一些性能，我们看到其实总体差距不是很大。

当时延越来越大的时候，二者的性能也是都在降低，但滑动窗口机制的效率下降的很快，这主要因为时延增大会导致两边都会产生超时重传，但是对于超时重传，停等机制的影响不大，滑动窗口机制则会产生不必要的超时重传，并且会重传窗口中的全部数据包，这会导致较大的额外开销影响性能，因此可以看出当时延增大达到超时重传的时间之后，滑动窗口机制的性能迅速下降，甚至不如停等机制。

总体来看，在没有丢包的情况下，停等机制表现更好，尤**其在丢包和时延都为零的时候，停等机制传输效率很高，也是因为其设计简单，没有复杂的锁机制的原因**，在时延增加后，停等机制和滑动窗口机制的性能都有所下降，但由于增大时延后，**滑动窗口机制会产生不必要的超时重传**，并且会重传窗口中的全部数据包，这会导致较大的额外开销影响性能，所以滑动窗口机制性能波动要比停等机制更剧烈。

# 四、 滑动窗口机制中不同窗口大小对性能的影响

在之前的 3-2 实验中，我们控制一定的小丢包率，探索了窗口变化时的性能变化：

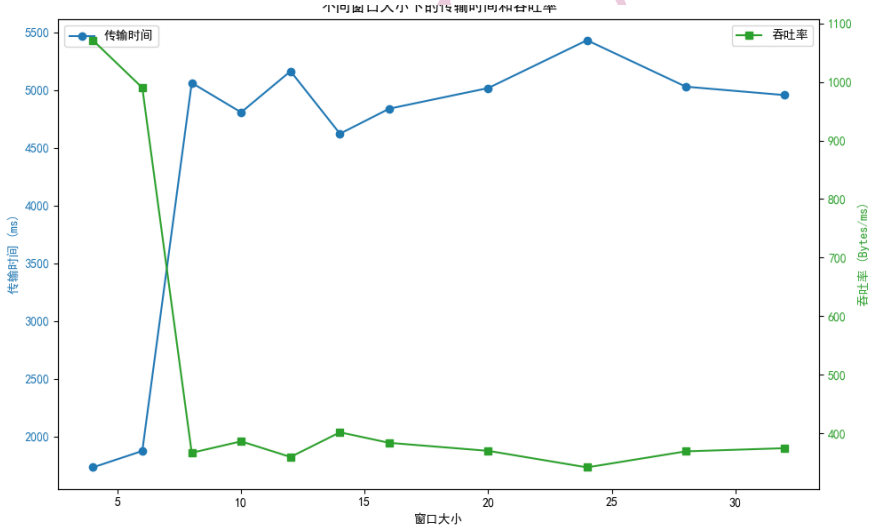


图 7: 不同窗口大小的传输时间与吞吐率

我们得出了与理论相悖的结论，即窗口较小时性能较高，而窗口较大时性能较低，理论上来说，没有丢包率或丢包率较低的时候（我设置的是 3% 的丢包率）时候，影响 GBN 的非必要重传很少发生，因此 GBN 的性能由于流水线机制随着窗口大小的增加而有更多数据报落入窗口可以被发送而增加。

这一点我推测与我前面反复再说的实现细节有关，在 **GBN 窗口越大，每次还需要对其进行遍历输出窗口内容（即输出窗口内的数据报）也越多**，在这种没有意义的事情上花费的时间越多，同样此时占据着缓冲区锁的时间也越来越久。导致主线程在发送数据等时间被拖延，当然也可能与我们的一些客观的因素包括网络带宽有限等等。

## (一) 控制延时为 0，改变丢包率，对比不同滑动窗口大小的传输时间与吞吐率

这里要说的是之前我们为了**控制变量**，我们滑动窗口运行的时候没有嵌入日志管理的程序（即程序没有开启日志锁），接下来我们实验的结果将会使用有日志管理的程序，从而配合我们 3-3 也实现了日志锁管理，控制变量。

| 窗口大小4 | | | | | | |
|---|---|---|---|---|---|---|
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 993 | 6940 | 13213 | 36991 | 71848 | 56581 |
| 吞吐率（Bytes/ms） | 5940.1 | 849.93 | 446.418 | 159.458 | 82.0972 | 104.23 |
| 窗口大小16 | | | | | | |
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 1845 | 21650 | 29385 | 25755 | 28687 | 55581 |
| 吞吐率（Bytes/ms） | 3197.03 | 272.449 | 200.732 | 229.024 | 205.616 | 106.125 |
| 窗口大小32 | | | | | | |
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 1928 | 19694 | 49501 | 27867 | 36830 | 54371 |
| 吞吐率（Bytes/ms） | 3059.4 | 299.508 | 119.16 | 211.666 | 160.155 | 108.46 |

图 8: 不同窗口在不同丢包率的情况下的结果

在实验中，通过控制变量的方式，探究了不同窗口大小对于累计确认的滑动窗口机制性能的影响。

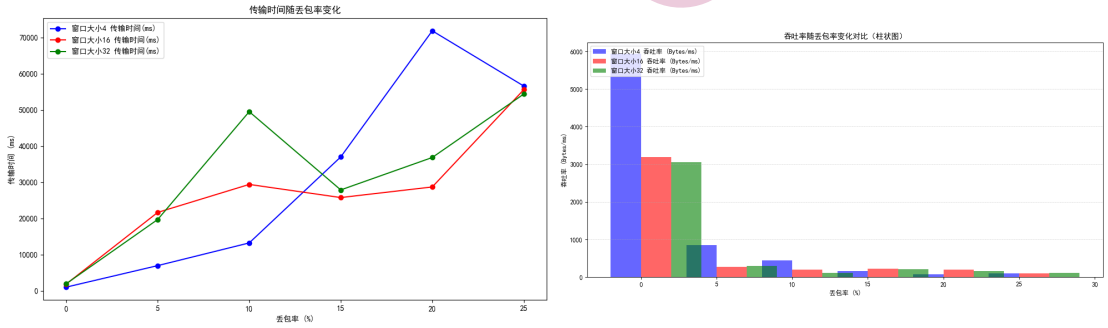


图 9: 传输时间与吞吐率对比分析

通过图表我们更清楚地发现其实有一些数据的偏差很大，我猜测这和我们使用的路由器丢包有关，我观察路由器的丢包率是按照百分比进行丢包，如果丢包率是 10% 那他就是 10 个数据包丢一次，这很容易造成一些误差，比如之前遇到的反复丢弃一个数据包而死锁的状态，这其实在现实中发生概率较低，所以我们分析的时候会忽略一些突变的数值。

我们观察到**丢包率从 0 增长到 5 时，效率降低的最大**，但是当我们再逐步增大丢包率时，效率虽然在逐步降低，但是降低的速率逐渐趋缓，**最终不同窗口在较大丢包率的情况下其差异也慢慢减小趋近相同**，这也是比较好理解的，这是因为从无到有的过程意味着一定会经历超时重传，所以每经过一个固定的时间就一定会经历超时重传，从而丢包率逐渐增大的过程只是经历超时重传的频率增大，所以不会引起大规模的效率降低。

- 在所有窗口大小下，随着丢包率的增加，吞吐率普遍呈下降趋势。
- 较小的窗口大小可能更敏感于丢包率的增加，导致吞吐率下降更为明显。
- 较大的窗口大小在一定程度上能够缓解丢包率增加对吞吐率的影响，但在高丢包率下仍然受到限制。

- 随着丢包率的不断加大，不同窗口大小间的差异变小。三者趋于平稳。

## （二） 控制丢包率为 0，改变延时，对比不同滑动窗口大小的传输时间与吞吐率

| 窗口大小4 | | | | | | |
|---|---|---|---|---|---|---|
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 606 | 35611 | 98032 | 36991 | 149804 | 211864 |
| 吞吐率（Bytes/ms） | 9733.53 | 165.637 | 60.1693 | 159.458 | 39.3749 | 27.8411 |
| 窗口大小16 | | | | | | |
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 2129 | 39888 | 111761 | 156873 | 206328 | 264562 |
| 吞吐率（Bytes/ms） | 2770.56 | 147.877 | 52.778 | 37.6006 | 28.5881 | 22.5764 |
| 窗口大小32 | | | | | | |
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 2052 | 39808 | 107553 | 154221 | 199629 | 274638 |
| 吞吐率（Bytes/ms） | 2874.52 | 148.174 | 54.8429 | 38.2472 | 29.5474 | 21.4774 |

图 10: 不同窗口在不同时延的情况下的结果

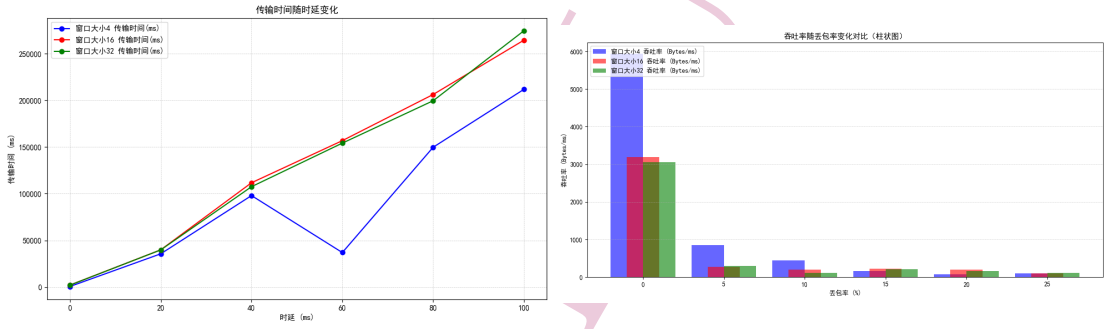


图 11: 传输时间与吞吐率对比分析

首先一些具有较大偏差的数据我们忽略不计，因为路由器本身会造成这一情况，我们主要分析的是数据的统一发展趋势：

我们看到延时较大时，会造成大量的超时重传，导致三个窗口大小范围的 GBN 都会因为内大量重传而性能不断恶化。并且三者差异不大，都处于较低水平。窗口大小即使增大带来的以此多发数据报也没有办法挽救过多的超时重传。

# 五、 有拥塞控制和无拥塞控制的性能比较

还是再说明一下，在实验 3-3 和实验 3-2 都实现了我们的日志管理，因此该部分比较数据与和停等机制比较的数据会有一定的出入。

## （一） 控制延时为 0，改变丢包率，对比有拥塞机制和无拥塞机制

这里我们选择滑动窗口为 16 的 GBN 和有拥塞控制的 RENO 进行比较

| 拥塞控制 | | | | | | |
|---|---|---|---|---|---|---|
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 2303 | 4635 | 8760 | 15599 | 6173 | 25749 |
| 吞吐率（Bytes/ms） | 2561.23 | 1272.6 | 673.347 | 378.134 | 300.885 | 229.078 |
| 窗口大小为16的滑动窗口机制 | | | | | | |
| 丢包 | 0 | 5 | 10 | 15 | 20 | 25 |
| 传输时间（ms） | 1845 | 21650 | 29385 | 25755 | 28687 | 55581 |
| 吞吐率（Bytes/ms） | 3197.03 | 272.449 | 200.732 | 229.024 | 205.616 | 106.125 |

图 12: 对比有拥塞机制和无拥塞机制

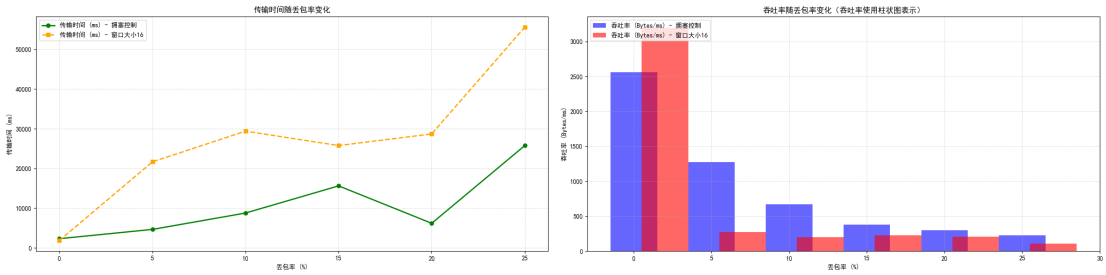


图 13: 传输时间与吞吐率对比分析

我们分析得到，在丢包率为 0 的时候，由于 GBN 的窗口固定为 16，能高效的发送数据包且由于不触发丢包的情况，传输效率高；拥塞机制下的窗口其窗口大小增长偏慢，传输效率不如窗口大小固定的 GBN 我们也简单分析了 RENO 一开始较慢的原因：

- **恢复慢：**当一个数据包丢失时，Reno 会将慢启动阈值（ssthresh）设置为当前窗口的一半，然后重新进入拥塞避免阶段。这意味着，如果网络带宽很大，一个小的丢包就会导致窗口大幅度缩小，需要花费很多时间才能恢复到丢包前的窗口大小。尤其是在高带宽的环境下，这种恢复速度过慢，导致网络的带宽利用率大打折扣。
- **带宽利用率低：**由于每次丢包时窗口缩小的幅度过大，导致带宽利用率不能达到理想状态。尤其是在带宽较高的情况下，单个丢包的影响是显著的，使得有效吞吐量难以持续保持在较高水平。

但是一旦丢包率增大后，我们可以看到虽然 RENO 算法仍导致其窗口一直被限定在较小范围内，对信道的利用率降低，但是减少了一个窗口反复重传的情况，反观我们的无拥塞控制的程序，因为我们的窗口固定，一旦发生超时会导致窗口中所有数据包重传造成无效重传，并且我在设计程序的时候在重传是增加了较多的日志，这部分的重复操作（打印缓冲区从而遍历缓冲区）也放大了这一问题。

## （二） 控制丢包率为 0，改变延时，对比有拥塞机制和无拥塞机制

| 拥塞控制 | | | | | | |
|---|---|---|---|---|---|---|
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 2326 | 33650 | 98032 | 121756 | 149804 | 211864 |
| 吞吐率（Bytes/ms） | 2535.91 | 175.29 | 60.1693 | 46.0818 | 39.3749 | 27.8411 |
| 窗口大小16 | | | | | | |
| 时延 | 0 | 20 | 40 | 60 | 80 | 100 |
| 传输时间（ms） | 2129 | 39888 | 130849 | 156873 | 206328 | 264562 |
| 吞吐率（Bytes/ms） | 2770.56 | 147.877 | 45.0788 | 37.6006 | 28.5881 | 22.5764 |

图 14: 不同时延下的统计结果

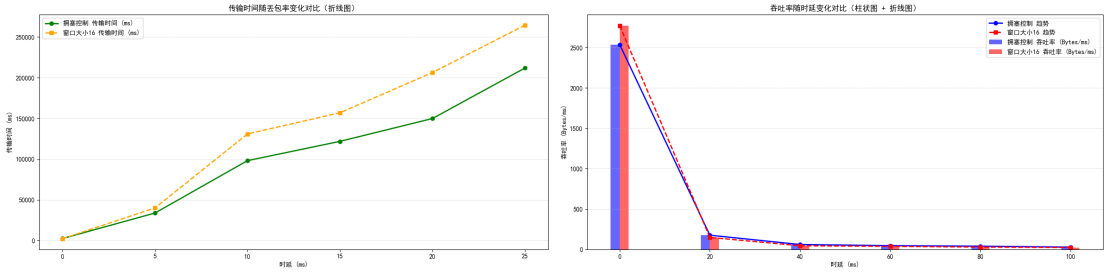


图 15: 传输时间与吞吐率对比分析

我们观察到时延对于 TCP RENO 拥塞机制算法的影响：

- 在高时延下，TCP RENO 更加依赖 RTT（往返时间）来确认数据包是否成功送达。当时延增加时，RTT 也会显著增加，这导致 ACK 确认的延迟变大，窗口增长变得更慢。
- 高时延会增加网络出现丢包的概率，TCP RENO 一旦检测到丢包，通常会触发快速恢复阶段。

我们再分析快速恢复的机制为何导致性能降低：

- 在快速恢复阶段，窗口大小会减小为原来的一半，并进入拥塞避免阶段。
- 在高时延条件下，快速恢复被频繁触发，导致窗口增长的速度变得极慢，从而直接限制了吞吐率。

而针对于无拥塞机制下的时延越大之后性能急剧下降的原因之前我们也分析过，是由于其不断地快速重传，我们看到时延超过 20ms 后二者的性能趋近于稳定，二者在较高时延下的表现趋近一致。

我们再简单看看 0-20ms 的时延下二者的性能：

| 拥塞控制 | | | | | |
|---|---|---|---|---|---|
| 时延 | 0 | 5 | 10 | 15 | 20 |
| 传输时间（ms） | 2326 | 11319 | 28670 | 29093 | 36338 |
| 吞吐率（Bytes/ms） | 2535.91 | 521.116 | 205.738 | 202.747 | 162.324 |
| 窗口大小16 | | | | | |
| 时延 | 0 | 5 | 10 | 15 | 20 |
| 传输时间（ms） | 2129 | 23284 | 30867 | 37470 | 38689 |
| 吞吐率（Bytes/ms） | 2770.56 | 253.329 | 191.095 | 157.42 | 152.46 |

图 16: 0-20ms 时延的统计结果

以下是对比分析图：

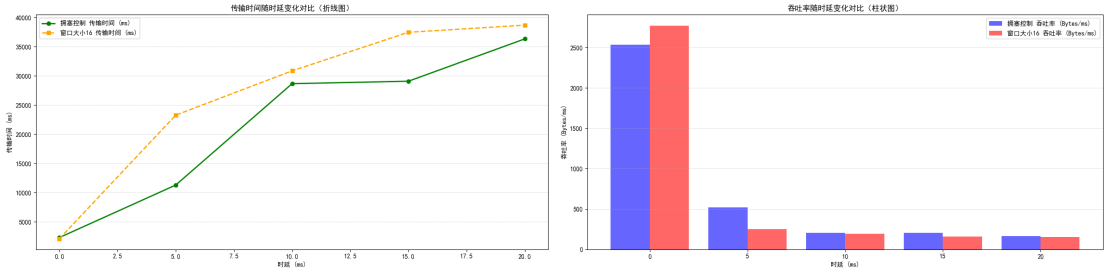


图 17: 传输时间与吞吐率对比分析

我们看到在没有时延和发生时延直接有较大的性能差异，这很好理解：

- 窗口大小为 16 时，由于我设置的超过 120ms 就进行超时重传，在窗口中的已发送未确认数据包不断增大时，后面的数据包会因为前面的累计时延势必会造成超时重传，而超时重传的时候会将窗口中所有数据包全部发送从而造成性能的降低。
- 拥塞机制下的程序也会发生这一较大波动是因为一方面随着窗口增大，也会像上面一样后续数据包会产生时延，并且不同于丢包产生的快速重传，产生超时重传后窗口会急剧缩小，造成性能降低，在下一次窗口数目增多的时候又会因为时延堆积造成超时，因此会性能急剧降低。

我们看到而这降低的数量级基本一致，差距不大，而拥塞控制稍好于我们无拥塞控制的原因，我猜测是由于拥塞控制的窗口始终在一个较合理的范围内波动（几乎没超过 15），所以超时重传的时候，重传全部数据包的数目较少，因此整体性能会稍好一些。

# 六、 实验改进

在实验过程中，我每次实验后也会进行实验性能简单的分析，并根据分析结果进行调整：

## 1. 日志混乱的解决

像下图展示的，在 3-2 引入了多线程后，我的日志就很混乱：

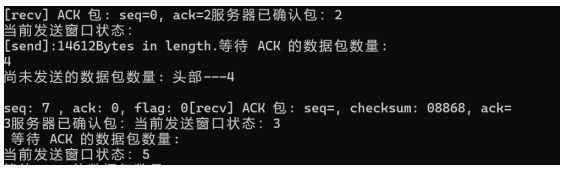


图 18: 直接打印日志混乱

所以我们将日志创建成共享资源，访问日志资源的时候需要先上锁，防止日志记录混乱，同时为了尽可能避免日志带来的效率问题，我们创建一个进程去打印日志，这样可以一边文件接收传输，一边打印日志，提升效率，但其实这样实现还是有一定的弊端，由于本次实验我们使用的互斥锁很多，主要在日志记录，还有部分在 timer 计时问题，所以仍然会我们的效率有影响，我对进行日志管理的程序与未进行日志管理的程序进行了简单的对比分析：

| 有日志管理（设置互斥锁等） | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 窗口大小 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 20 | 24 | 28 | 32 |
| 传输时间（ms） | 1734 | 1876 | 5062 | 4807 | 5163 | 4622 | 4839 | 5017 | 5431 | 5029 | 4956 |
| 吞吐率（Bytes/ms） | 1071.14 | 990.067 | 366.923 | 386.388 | 359.745 | 401.853 | 383.832 | 370.214 | 341.993 | 369.331 | 374.771 |
| 无日志管理（日志乱序输出） | | | | | | | | | | | |
| 窗口大小 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 20 | 24 | 28 | 32 |
| 传输时间（ms） | 1872 | 1699 | 3085 | 2922 | 2940 | 2913 | 3171 | 2917 | 2928 | 2798 | 3063 |
| 吞吐率（Bytes/ms） | 992.182 | 1093.21 | 602.063 | 635.649 | 631.757 | 637.612 | 585.735 | 636.738 | 634.346 | 663.819 | 606.388 |

图 19: 有日志管理与无日志管理下不同窗口大小的传输时间与吞吐率

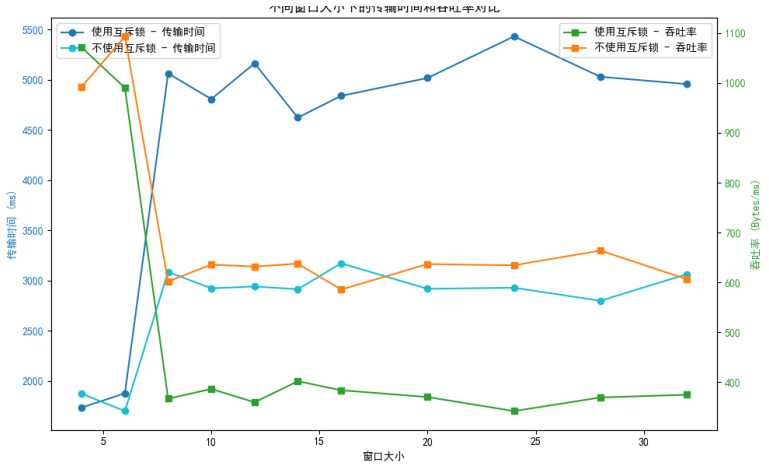


图 20: 有日志管理与无日志管理下不同窗口大小的传输时间与吞吐率

抛开之前分析的窗口大小与吞吐率之间的关系，我们对比发现，在日志记录不进行互斥锁的情况下性能要优于我们设置了日志锁的情况，所以我们在本次实验分析的时候也考虑了这一点，比如 3-1 与 3-2 进行对比的时候我们进倾向于使用无日志管理的程序进行对比，在 3-3 由于是我们沿用了 3-2 的大部分内容，所以我们就使用有日志管理的来进行比较。

## 2. 快速重传

因为在 3-3 实验中，我们没有在接收端实现乱序数据包的管理，还是按顺序接收，因此本次实现 reno 的时候，我尝试选择快速重传所有的数据包，这类似于超时重传，这是因为如果我们不进行所有数据包的快速重传，仅仅重传滑动窗口的第一个数据包，这样后面（除了第一个数据包）势必会引发超时重传并没有完全解决仅因丢包而造成的拥塞：

```
[重复ACK] 计数: 1
[收到ACK]: seq: 0 , ack: 103, flag: 2, checksum: 65418
[重复ACK] 计数: 2
[收到ACK]: seq: 0 , ack: 103, flag: 2, checksum: 65418
[重复ACK] 计数: 3
[快速重传] ssthresh 设置为 3, cwnd 设置为 6.000000
[快速重传] 重传包 seq: 104
[收到ACK]: seq: 0 , ack: 103, flag: 2, checksum: 65418
[重复ACK] 计数: 4
[快速恢复] cwnd 增加到 7.000000
[收到ACK]: seq: 0 , ack: 103, flag: 2, checksum: 65418
[重复ACK] 计数: 5
[快速恢复] cwnd 增加到 8.000000
[收到ACK]: seq: 0 , ack: 103, flag: 2, checksum: 65418
[拥塞避免] cwnd 增加到 8.125000
[快速恢复] 退出快速恢复, cwnd 设置为 3.000000
[收到ACK]: seq: 0 , ack: 104, flag: 2, checksum: 65417
[超时] ssthresh 设置为 2, cwnd 重置为 1.000000
[超时重传] 重传包 seq: 105
[超时重传] 重传包 seq: 106
[超时重传] 重传包 seq: 107
[超时重传] 重传包 seq: 108
[超时重传] 重传包 seq: 109
[超时重传] 重传包 seq: 110
```

图 21: 只快速重传一个数据包最终还是引发超时

因此本次我尝试快速重传所有的数据包：

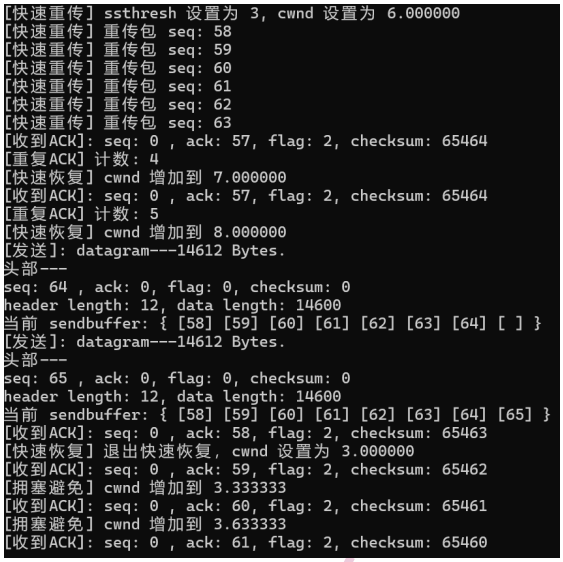


图 22: 重传全部的已发送未确认的数据包

我们简单看看两种情况下的传输性能：

| 传输1.jpg，丢包率3 | | | | | | |
|---|---|---|---|---|---|---|
| 快速重传只重传丢失的数据包 | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 传输时间（ms） | 1812 | 1841 | 1926 | 2125 | 1665 | 1873.8 |
| 吞吐率（Bytes/ms） | 1025.04 | 1008.89 | 964.364 | 874.054 | 1115.53 | 997.57 |
| 快速重传所有已发送未确认的数据包 | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 传输时间（ms） | 1774 | 1689 | 1569 | 1839 | 1909 | 1756 |
| 吞吐率（Bytes/ms） | 1046.99 | 1099.68 | 1183.79 | 1009.99 | 972.952 | 1062.68 |

图 23: 比较两种情况

我们看到每种情况都进行了 5 次实验，从而比较平均值而减小其他误差，但其实我们其实看到数据的波动性是很大的，在测试的时候我也发现了，不断用一个路由器传输越传效率越差，所以接下来我们使用 3.jpg（数据更大）来进行实验，每次实验都会重启新的路由器：

| 传输3.jpg，丢包率3 | | | | | | |
|---|---|---|---|---|---|---|
| 快速重传只重传丢失的数据包 | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 传输时间（ms） | 11912 | 10980 | 12909 | 11050 | 11100 | 11590 |
| 吞吐率（Bytes/ms） | 1004.79 | 1090.07 | 927.183 | 1083.17 | 1078.29 | 1036.7 |
| 快速重传所有已发送未确认的数据包 | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 传输时间（ms） | 9505 | 10352 | 9198 | 9421 | 9299 | 9555 |
| 吞吐率（Bytes/ms） | 1259.23 | 1156.2 | 1301.26 | 1270.46 | 1287.13 | 1254.65 |

图 24: 利用 3.jpg 进行对比

我们可以看到相比于刚才，这次的数据相对更加稳定，很大一部分是因为本身传输的数据（3.jpg）比较大，从而降低了一部分误差。

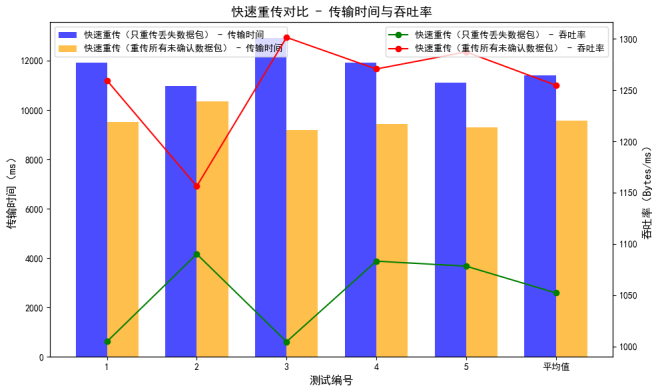


图 25: 利用 3.jpg 进行对比

从目前的结果来看，我们可以初步认为我们快速重传全部的已发送未确认的 ACK 相比之下，效率更高。

但如果发送丢包的时候窗口中数据包已经很多了，进入快速恢复后还在接收重复的 ACK 的时候增加我们的窗口大小，那我们泽佳窗口大小后发送的数据包很容易会因为前面数据太多而超时，或者在快速重传的时候丢包，后续还是会进入超时重传。

# 七、 实验总结和心得体会

## (一) 实验问题

本次实验，我认为最大的问题还是在使用路由器进行丢包的时候，我有的时候会因为路由器始终丢一个包而无法传输完成，所以实验中有些数据我会标注实际丢包率比如 15（17）表示的是在设置 17 的丢包率时能正常传输。

同时我发现路由器会有一定的误差，在同一个实验程序，同一个传输样本，同样的丢包率和时延（都是 0 啊）但是传数据的结果有时会差出大概三分之一，而且有时一直开着路由器传输，会发生越传越慢的情况，因此为尽量避免误差，我每次传输都会重启路由器（妈呀累死我了），然后尽量多次实验，以及不使用之前的数据，每次都重复测试（这也是为啥不同是实验但是同一个情况下实验结果不大一样）。

但因为有的时候误差确实大的离谱，所以我后期（尤其在后两组对比）我尝试自己在程序中实现丢包和延时，丢包的实现就是采用随机数的架构实现的，而延时主要是通过 sleep() 函数来实现，这也相应的减小了一定的误差。

## (二) 实验心得

这次实验是我前三次实验的归纳和总结，不仅是进一步改进程序以便测试，还包括实际测试时候的种种与理论不符合的情况。分析这些现象后，不仅加深了我对这部分知识点的理解，更让我对自己的实现有了更多的思考，找到了更多改进的方向。

在这次大作业中，我在进行实验的同时我也不断地思考改进实验的完成功能，我也在这次实验渐渐意识到在理论研究和动手实践中也会有很大的差距，所以在未来的学习中，我不仅会仅仅动手实践同时会勤反思从而更深入的学习知识！