计网Lab3_4研究报告

姓名：刘玉菡
学号：2213244
专业：物联网工程

一、实验内容

本次实验基于UDP服务设计可靠传输协议，并通过改变网络的延迟时间和丢包率，完成以下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。

实验要求实现单向传输，对于每一个任务给出详细的协议设计，说明拥塞控制算法的原理，完成给定测试文件的传输并显示传输时间和平均吞吐率，同时给出性能测试指标（包括吞吐率、文件传输时延等）的图形结果并进行分析。

实验环境：

windows 11 和 Visual Studio 2022

二、协议设计

（一）数据包结构和字段说明

通过Packet结构体实现数据包的组织，各字段功能如下表所示：

字段	数据类型	说明
seq	uint32_t	序列号，用于标识每个数据包的顺序，每发送一个数据包，序列号递增。
ack	uint32_t	确认号，表示接收到的上一个数据包的序列号，接收方用来告诉发送方自己已收到数据包。
flag	uint16_t	标志位，表示数据包的类型，如连接请求、数据包、确认包等。SYN建立连接；ACK确认包；FIN关闭连接 ； DATA数据包；FILENAME文件名包；CLOSE关闭连接包
len	uint16_t	数据部分的长度，表示data的有效字节数。
checksum	uint16_t	校验和，用于验证数据包的完整性。
data	char [BUF_SIZE]	数据部分，实际传输的内容。

（二）连接管理

- **连接建立**：类似于TCP**三次握手**
    1. **SYN**：客户端向服务器发送SYN包，要求建立连接。
    2. **SYN-ACK**：服务器收到SYN包后，返回SYN-ACK包，表示同意建立连接。
    3. **ACK**：客户端收到SYN-ACK包后，返回ACK包，完成连接的建立。
- **连接关闭**：类似于TCP**四次挥手**
    1. **FIN**：发送端发送FIN包，表示连接断开。
    2. **ACK**：接收方返回ACK包，确认断开请求。
    3. **FIN**：另一方也发送FIN包，表明同意断开连接。
    4. **ACK**：最后一方返回ACK包，连接断开。

（三） 差错检验

通过校验和机制（Checksum）保证数据的完整性：

1. 发送端在数据包发送前计算数据包内容的校验和，并将其写入Checksum字段。
2. 接收端在接收到数据包后重新计算校验和，若匹配则继续处理，否则丢弃该数据包。

校验和使用16位的“补码和”算法，遍历数据包的所有字节，对其进行累加并保留16位结果。

（四）数据传输

文件传输过程

1. **文件名传输**：首先，发送端传输文件名给接收端。文件名传输采用数据包格式中的DATA字段，发送文件名作为数据内容。
2. **文件内容传输**：发送端通过数据包依次传输文件内容，每发送一个数据包，发送端都会等待接收端的ACK确认，确认后再发送下一块。
3. **文件传输完成**：文件传输完成后，发送端发送FIN包，表示数据传输结束。

数据传输

在之前的三次实验中，我分别实现了**停等机制、滑动窗口机制（GO-BACK-N)、拥塞控制算法。**这些协议分别展示了不同的数据传输方法和可靠性策略，每一种都有特定的适用场景和优缺点。

1. 停等协议

（1）发送端每发送一个数据包后，必须等待接收端的ACK确认。如果在指定时间内未收到ACK，则重新发送该数据包。

（2）接收端在收到数据包后，验证其校验和是否正确，并返回ACK确认数据包的接收情况。

（3）若发送端未在设定的超时时间（我设置为1秒）内接收到接收端的ACK，则会触发重传机制。

2. 滑动窗口机制

（1）**发送条件**：发送端在任意时刻仅当 `nextSeq < base + WINDOW_SIZE` 时，才允许发送新的数据包。

- `base`：当前窗口的基序列号，表示最早未被确认的数据包的序列号。
- `nextSeq`：下一个待发送的数据包的序列号。

（2）**窗口滑动**：当接收端发送ACK确认后，发送端将 `base` 移动至新的序列号，从而释放窗口空间，允许发送更多的数据包。确保发送端不会以过快的速率发送数据，有效控制了数据流量。

采用Go-Back-N（GBN）机制来实现可靠的数据传输和错误恢复。当某个数据包或其ACK丢失时，接收端会持续发送相同的ACK，指示发送端重传从丢失点开始的所有未确认的数据包。

（3）**累积确认**：

- 接收端通过返回 `Ack = expectedSeq` 来累积确认所有序列号小于 `expectedSeq` 的分组已被正确接收。
- 若中间有分组丢失，接收端不会对之后的分组单独确认，而是继续发送对同一 `expectedSeq` 的ACK，阻止ACK的前移，迫使发送端进行重传。

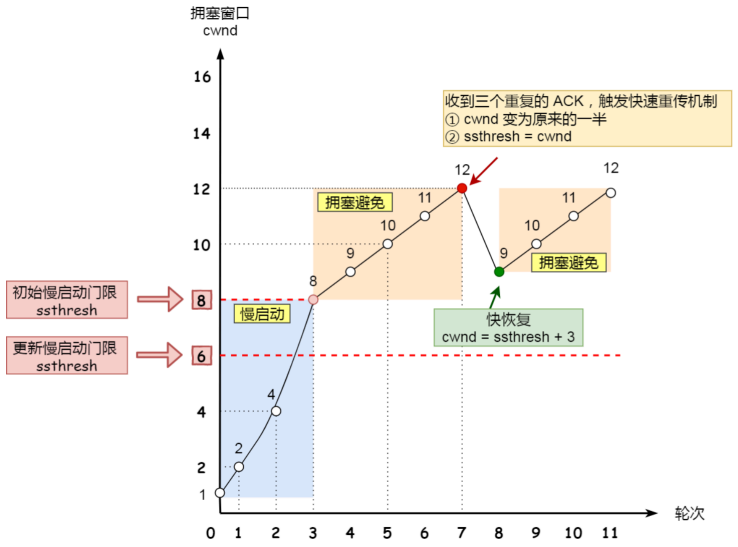
（4）**重传机制**：

- 在主循环的每次迭代结束时，发送端会遍历 `sendBuffer`，检查每个未确认的数据包是否超时，如果超时且未达到最大重传次数，则重传该数据包。
- 当某个数据包的重传次数达到 `MAX_RETRANSMISSIONS` 时，表示该数据包仍未得到确认，并且多次尝试重传也没有成功。这时，程序会打印错误信息，表示传输失败，通过设置 `connected = false` 来中断文件传输过程，结束连接，并且清空发送缓冲区，以避免继续在不可靠的网络环境中尝试传输更多的数据。

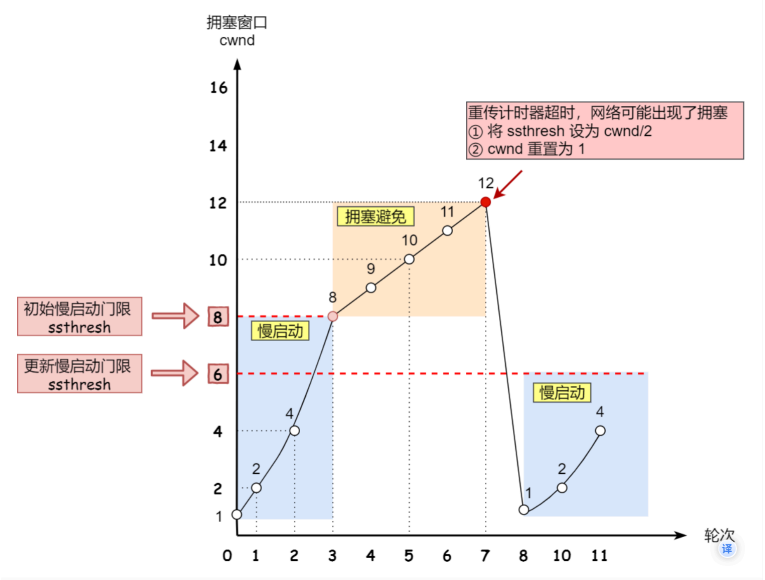
3. 拥塞控制

拥塞控制算法用于动态调整发送数据的速率，以避免网络拥塞。该算法通过维护两个关键参数和四个阶段来实现：

- **拥塞窗口（cwnd）**：控制可以发送的未确认数据包的数量。
- **慢启动阈值（ssthresh）**：当拥塞窗口达到该阈值时，算法从慢启动阶段切换到拥塞避免阶段。
- 四个阶段：**慢启动（Slow Start）**、**拥塞避免（Congestion Avoidance）**、**快速重传（Fast Retransmit）** 和 **快速恢复（Fast Recovery）**。下图可以明确四个阶段的流程，这里不过多赘述。



超时重传：



三、性能对比实验

实验中丢包率的单位为%，吞吐率的单位为 kbs，时延的单位为ms。采用控制变量法，对不同变量分别进行测试。

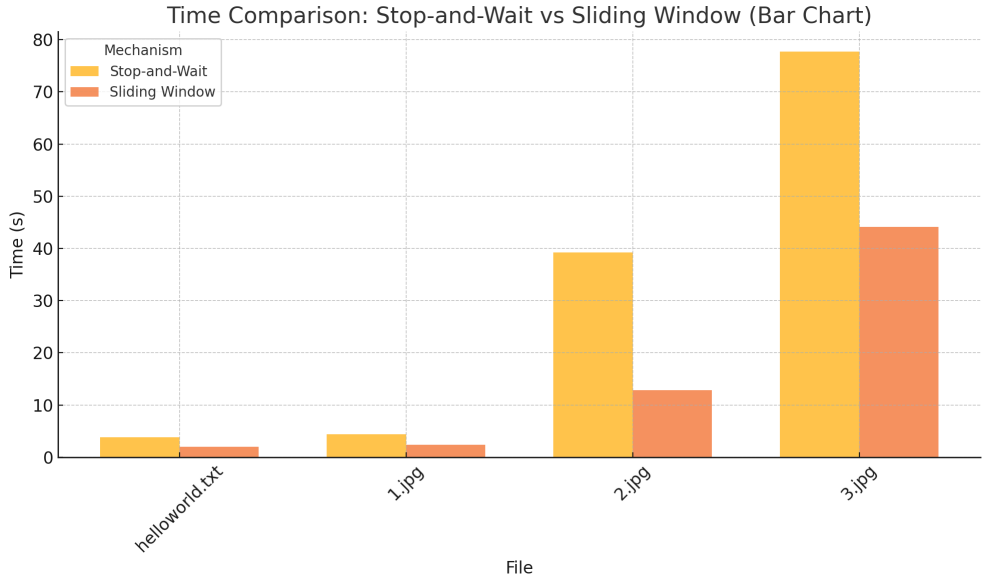
（一）停等机制与滑动窗口机制性能对比

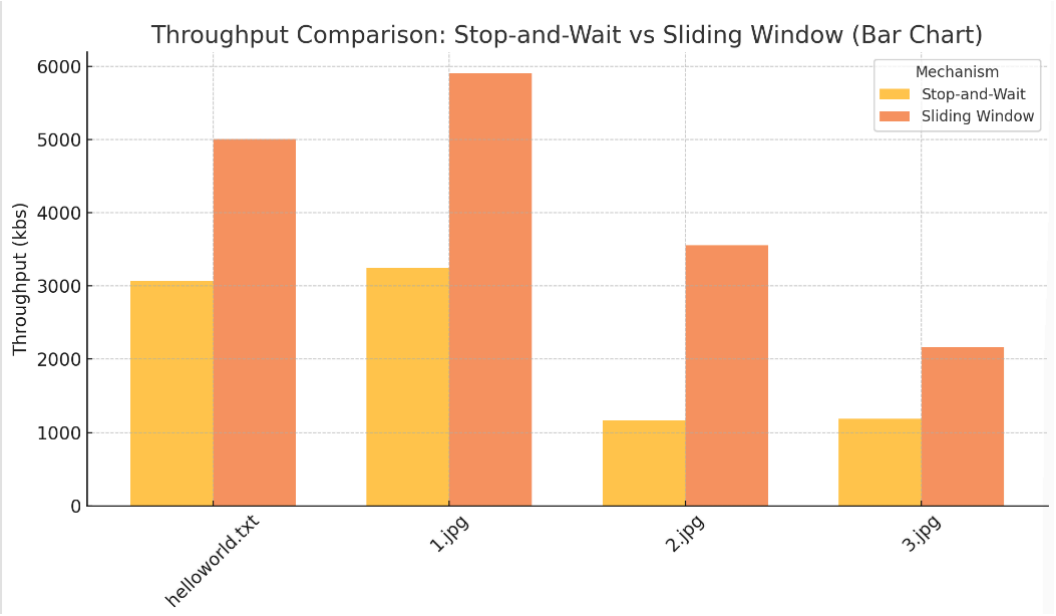
1. 丢包率为0，时延为0的理想情况

采用控制变量法，保证其他变量一致（如窗口大小设置为30），先将丢包率和延时都设置为0，分别使用停等机制与滑动窗口机制传输测试文件，得到如下结果：

机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	3066.11	3.79
停等机制	1.jpg(1.77MB)	3251.43	4.35
停等机制	2.jpg(5.62MB)	1161.86	39.26
停等机制	3.jpg(11.4MB)	1188.89	77.69
滑动窗口	helloworld.txt	5006.56	1.99
滑动窗口	1.jpg	5902.16	2.34
滑动窗口	2.jpg	3556.37	12.83
滑动窗口	3.jpg	2166.93	44.16

结果可视化如下所示





观察可视化数据和分析对应表格，可得如下结论：

（1）**传输时间分析**：

在丢包率为 0，时延为 0 的理想情况下，滑动窗口机制在所有文件的传输时间上均明显优于停等机制。例如，对于 helloworld.txt 文件，滑动窗口机制的传输时间为 1.99 s，而停等机制为 3.79 s。

造成这种结果是因为停等机制在每次发送数据包后都必须等待接收方的确认 ACK，这会导致每次数据包的发送与确认之间存在较长的空闲时间，尤其在大文件的情况下，空闲时间的累积会显著增加总的传输时间；相比之下，滑动窗口机制通过允许发送方在未收到前一个数据包的确认时继续发送多个数据包，显著减少了等待时间，因此在传输过程中能更高效地利用网络带宽。

（2）**吞吐率分析**：

对于吞吐率而言，滑动窗口机制相较于停等机制表现出了更好的性能。比如对于 helloworld.txt 文件，滑动窗口机制的吞吐率为 5006.56 kbs，明显高于停等机制的 3066.11 kbs。

这种差异的根本原因在于滑动窗口机制的高效利用了带宽。在滑动窗口机制下，多个数据包可以在确认收到前并行发送，确保数据传输的高效性，而停等机制则由于每次数据包的发送与确认之间的等待时间，导致了带宽的浪费。因此，在相同的网络条件下，滑动窗口机制能够实现更高的吞吐率，特别是在文件较大的情况下，优势更加明显。

2. 丢包率分别设置为10%、20%、30%进行测试，模拟现实中网络状况的波动

丢包率10%：

机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	2343.08	4.79
停等机制	1.jpg(1.77MB)	2433.27	4.92
停等机制	2.jpg(5.62MB)	1110.14	40.66
停等机制	3.jpg(11.4MB)	1013.44	81.06
滑动窗口	helloworld.txt	4792.71	2.08
滑动窗口	1.jpg	4688.54	2.57
滑动窗口	2.jpg	3477.14	13.81
滑动窗口	3.jpg	2038.92	45.05

丢包率20%：

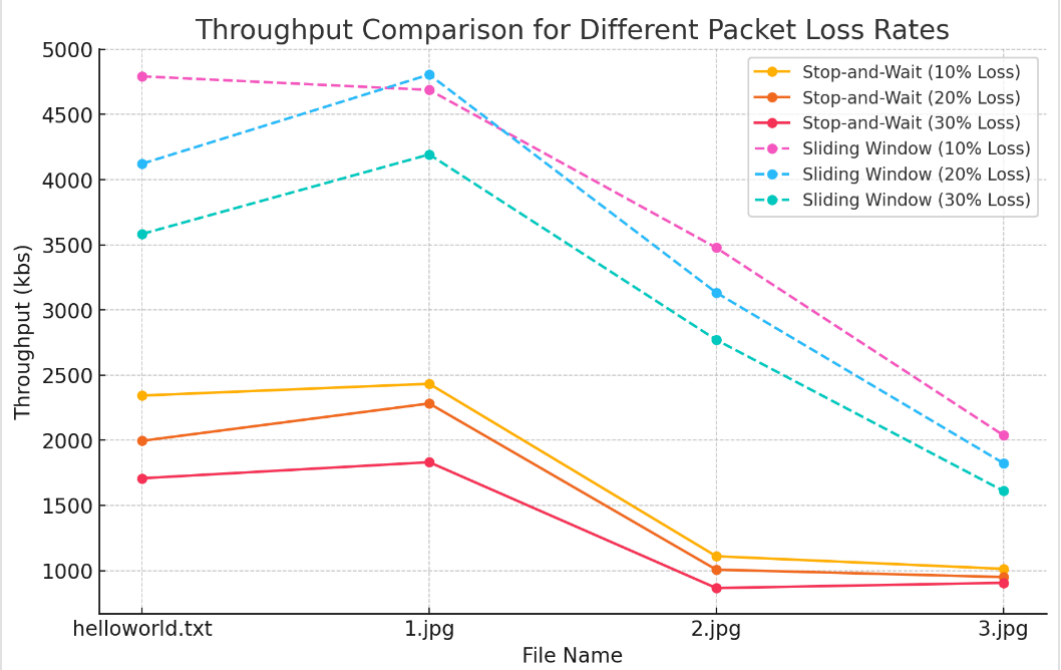
机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	1995.92	5.46
停等机制	1.jpg(1.77MB)	2282.31	5.88
停等机制	2.jpg(5.62MB)	1007.18	46.43
停等机制	3.jpg(11.4MB)	950.39	90.77
滑动窗口	helloworld.txt	4122.12	2.58
滑动窗口	1.jpg	4806.11	3.28
滑动窗口	2.jpg	3132.87	15.51
滑动窗口	3.jpg	1824.94	52.16

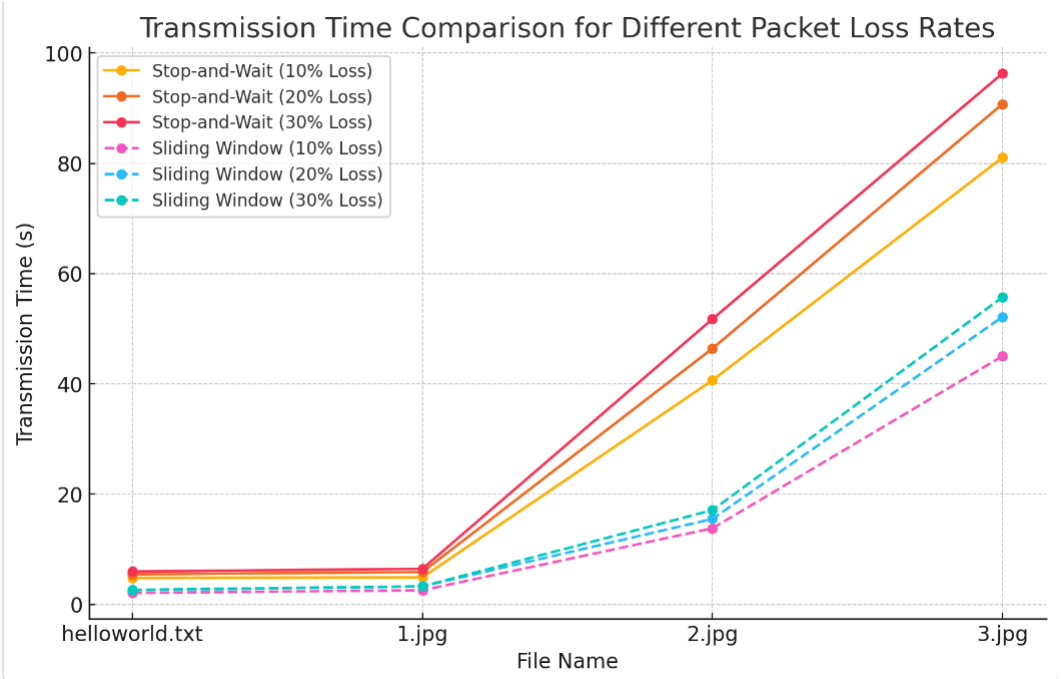
丢包率30%：

机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	1708.43	5.97
停等机制	1.jpg(1.77MB)	1831.62	6.47
停等机制	2.jpg(5.62MB)	865.91	51.77
停等机制	3.jpg(11.4MB)	906.03	96.32
滑动窗口	helloworld.txt	3583.14	2.64
滑动窗口	1.jpg	4193.31	3.28

机制	文件	吞吐率/kbs	时间/s
滑动窗口	2.jpg	2770.69	17.11
滑动窗口	3.jpg	1611.55	55.73

结果可视化如下所示：





总体根据测试结果而言，**丢包率越大，吞吐率越低，传输时间越长。**

时间：停等机制每发送一个数据包都要等待确认后才发下一个，丢包时，发送方需重传丢失数据包，且必须等重传完成才能继续，大文件时重传导致的延迟累积严重。而滑动窗口机制允许在未收到部分数据包确认时继续发送多个包，即便有丢包，可通过重传机制和窗口管理策略，更灵活安排重传和后续数据包发送，减少因丢包重传造成的等待时间，从而缩短传输时间。通俗来讲，停等机制是一个环节完成并确认无误后才进行下一个环节，一旦某个环节出错重来，后面环节都得等着；而滑动窗口机制是允许一定数量环节同时进行，某个环节出问题重处理时，其他环节还能继续推进，整体效率更高。

吞吐率：停等机制因频繁等待确认，网络带宽不能充分利用，丢包重传进一步占用带宽和时间，降低了单位时间内传输的数据量即吞吐率。滑动窗口机制可并行发送多个数据包，在一定程度上保持数据传输的连续性，即便有丢包重传，也能利用窗口空间合理安排，减少对整体传输效率的影响，使网络带宽得到更高效利用，实现更高吞吐率。就好比一条道路上，停等机制是一辆车完全通过一个路段后下一辆车才出发，路没被充分利用；滑动窗口机制相当于同时允许多辆车在不同路段行驶，只要路段有空余就可安排车辆进入，道路利用率高，运输能力（吞吐率）也就更强。

2. 延时分别设置为10ms、20ms、30ms进行测试，模拟现实中括传输延迟、路由延迟、处理延迟等网络延时

10ms延时：

机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	3076.11	4.19
停等机制	1.jpg(1.77MB)	3261.43	4.65
停等机制	2.jpg(5.62MB)	1171.86	39.96
停等机制	3.jpg(11.4MB)	1198.89	78.89
滑动窗口	helloworld.txt	5016.56	2.29
滑动窗口	1.jpg	5912.16	2.84
滑动窗口	2.jpg	3566.37	13.93
滑动窗口	3.jpg	2176.93	45.26

20ms延时：

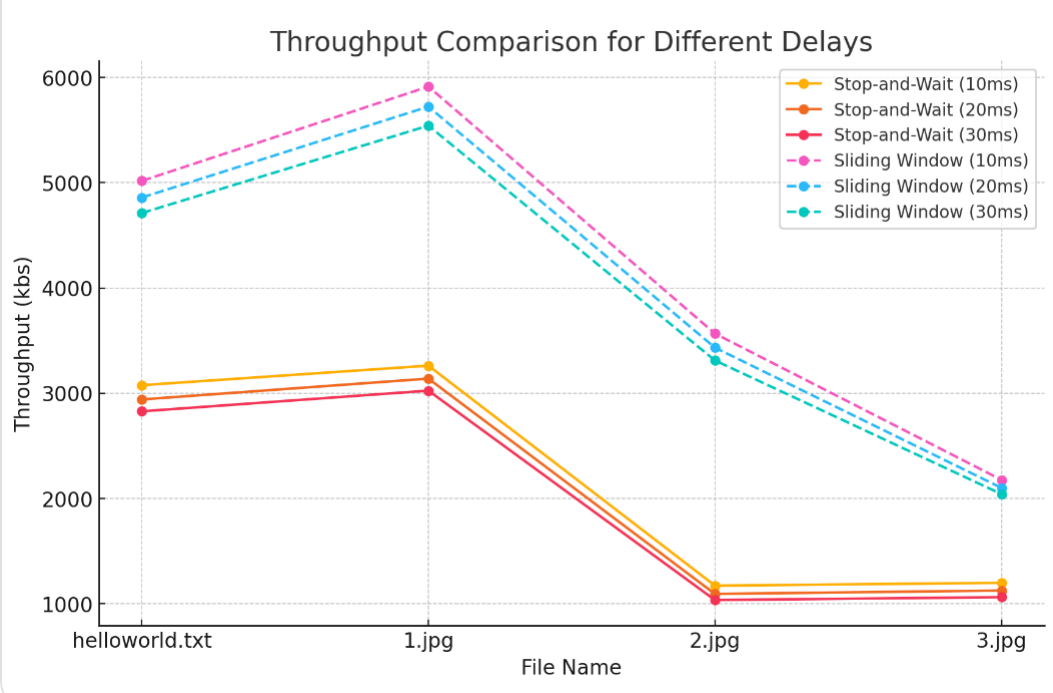
机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	2940.60	4.33
停等机制	1.jpg(1.77MB)	3138.55	4.88
停等机制	2.jpg(5.62MB)	1093.43	41.82
停等机制	3.jpg(11.4MB)	1125.70	81.10
滑动窗口	helloworld.txt	4858.91	2.45
滑动窗口	1.jpg	5721.94	2.81
滑动窗口	2.jpg	3432.73	14.47
滑动窗口	3.jpg	2100.98	46.70

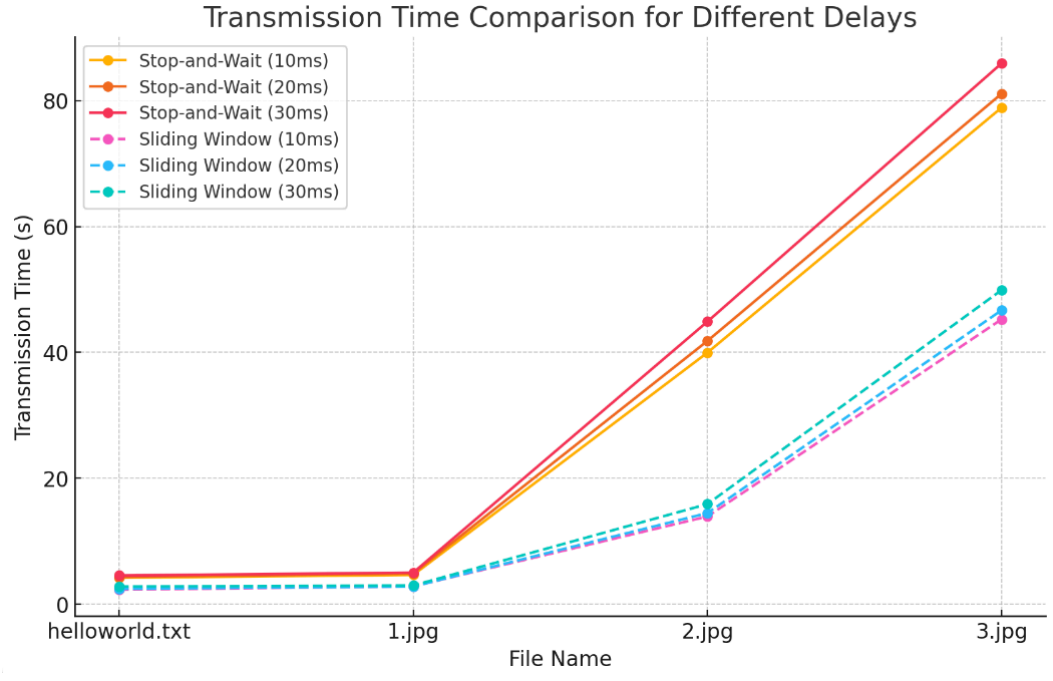
30ms延时：

机制	文件	吞吐率/kbs	时间/s
停等机制	helloworld.txt(1.57MB)	2827.81	4.56
停等机制	1.jpg(1.77MB)	3024.18	5.02

机制	文件	吞吐率/kbs	时间/s
停等机制	2.jpg(5.62MB)	1035.34	44.89
停等机制	3.jpg(11.4MB)	1062.49	85.92
滑动窗口	helloworld.txt	4711.42	2.81
滑动窗口	1.jpg	5541.38	2.97
滑动窗口	2.jpg	3311.09	15.90
滑动窗口	3.jpg	2042.04	49.91

结果可视化如下所示：





延时增加时，停等机制下发送方等待接收方确认的时间变长，整个传输过程时间增加；滑动窗口机制下虽然可等待接收方确认多个包，但延时增加仍会导致**吞吐率下降和传输时间增加。**

停等机制每次发送数据包后需等待接收方确认，随着延时增加，该等待时间变长，致使整个传输过程变慢。同时，由于每发一个包都要等待确认，在这个等待过程中网络带宽处于空闲状态，无法有效利用，导致吞吐率降低。而滑动窗口机制虽可等待多个包确认，但延时使数据包在网络中传输和确认的往返时间变长，这影响了窗口状态更新以及新数据包的发送，尽管其能利用窗口内数据包并行传输减少部分影响，但在大文件传输时，延时累积依然会使传输时间增加。并且，随着延时增加，确认包返回慢，窗口滑动变慢，进而影响数据包的持续发送，使得单位时间内传输的数据量减少，最终导致吞吐率下降。不过，相比停等机制，滑动窗口机制在应对延时方面仍具有一定优势，其在传输时间和吞吐率上的表现相对更好。

（二）滑动窗口机制中不同窗口大小对性能的影响

由于实验可调整变量过多，我采取了**低延时低丢包环境**和**高延时高丢包环境**下的不同窗口大小对性能影响的测试。

低延时低丢包环境（延时为5ms，丢包为3%）：

窗口大小为10：

文件	吞吐率/kbs	时间/s
helloworld.txt	4083.56	2.58
1.jpg	5661.29	2.41
2.jpg	3230.58	13.25
3.jpg	2355.37	43.82

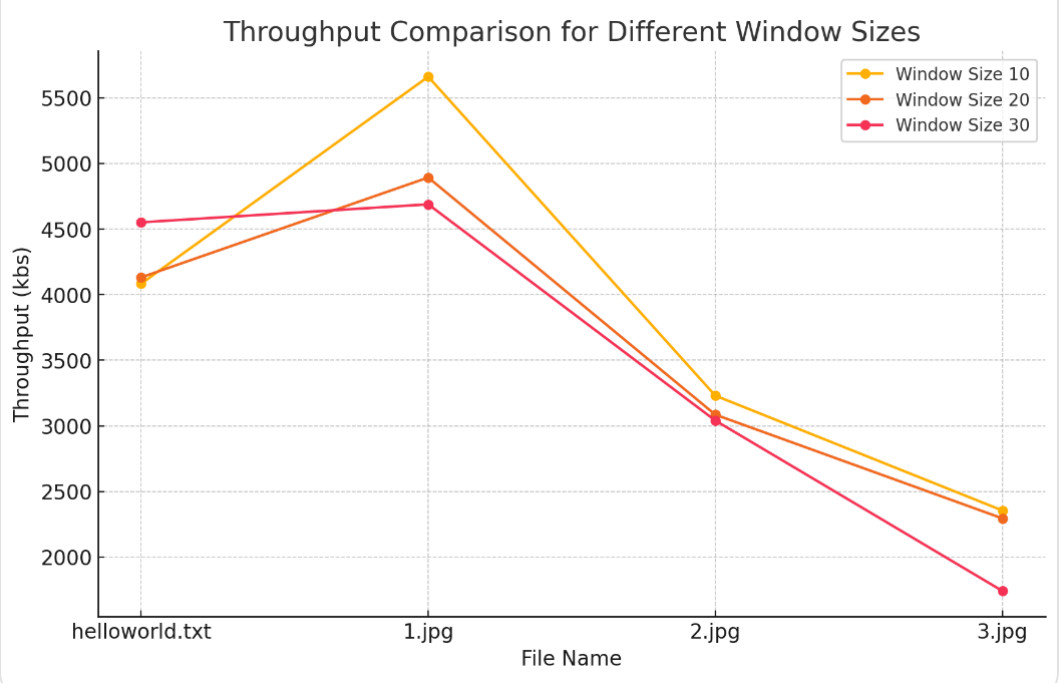
窗口大小为20：

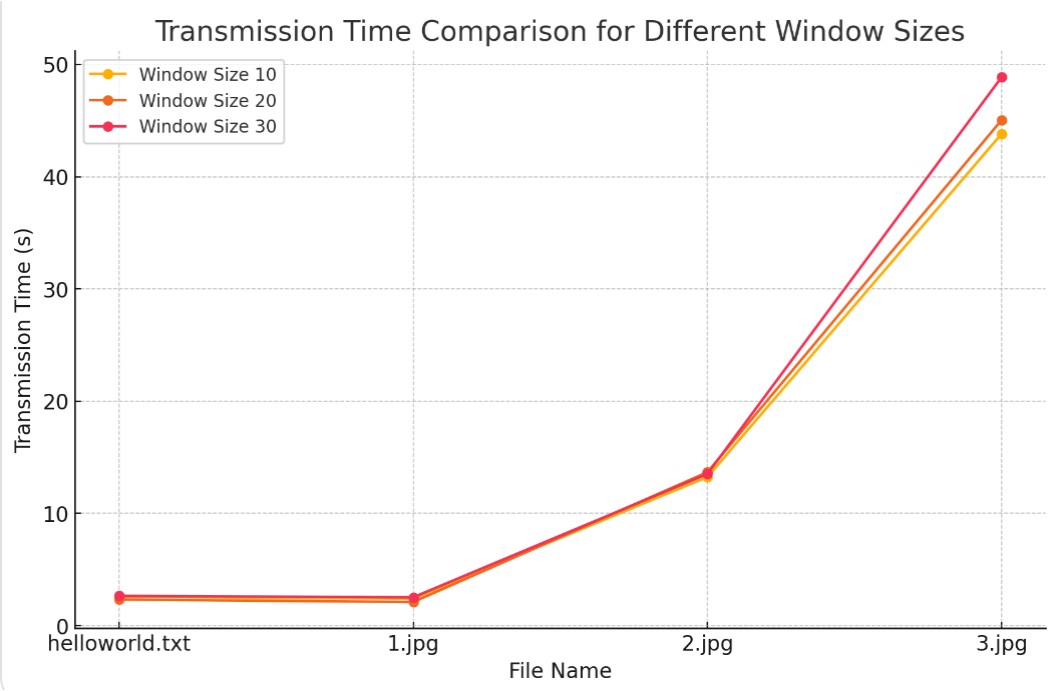
文件	吞吐率/kbs	时间/s
helloworld.txt	4131.85	2.33
1.jpg	4892.82	2.11
2.jpg	3085.59	13.68
3.jpg	2295.07	45.05

窗口大小为30：

文件	吞吐率/kbs	时间/s
helloworld.txt	4551.25	2.65
1.jpg	4688.58	2.53
2.jpg	3040.85	13.52
3.jpg	1742.23	48.87

结果可视化如下：





高延时高丢包环境（延时为30ms，丢包为30%）：

窗口大小为10：

文件	吞吐率/kbs	时间/s
helloworld.txt	1876.34	6.81
1.jpg	2201.42	5.92
2.jpg	1420.69	36.99
3.jpg	1134.67	113.81

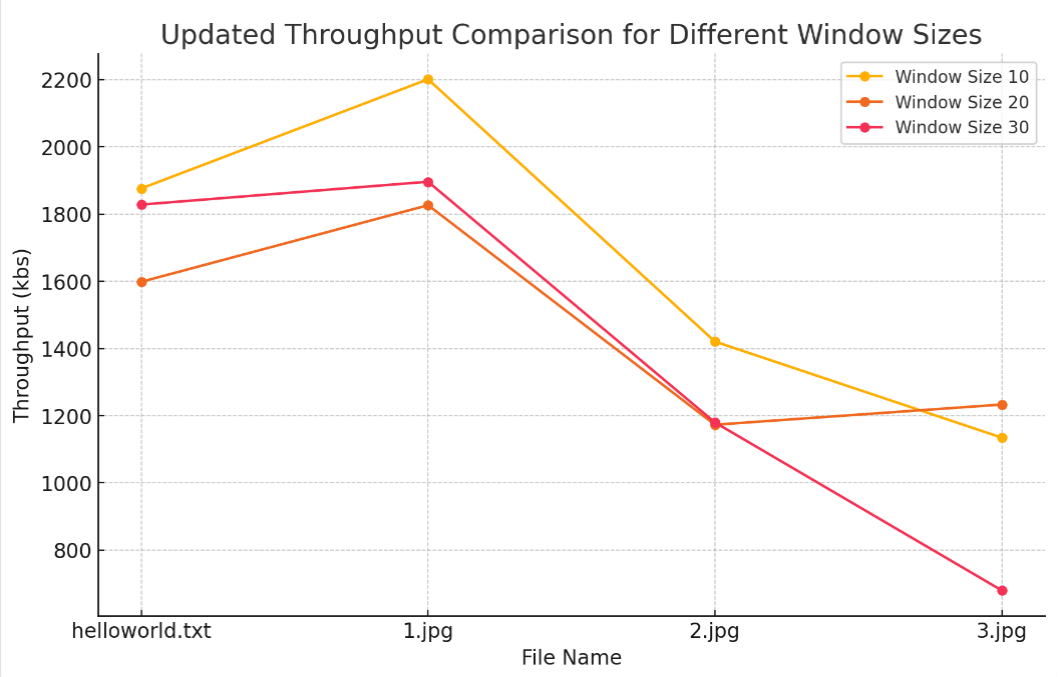
窗口大小为20：

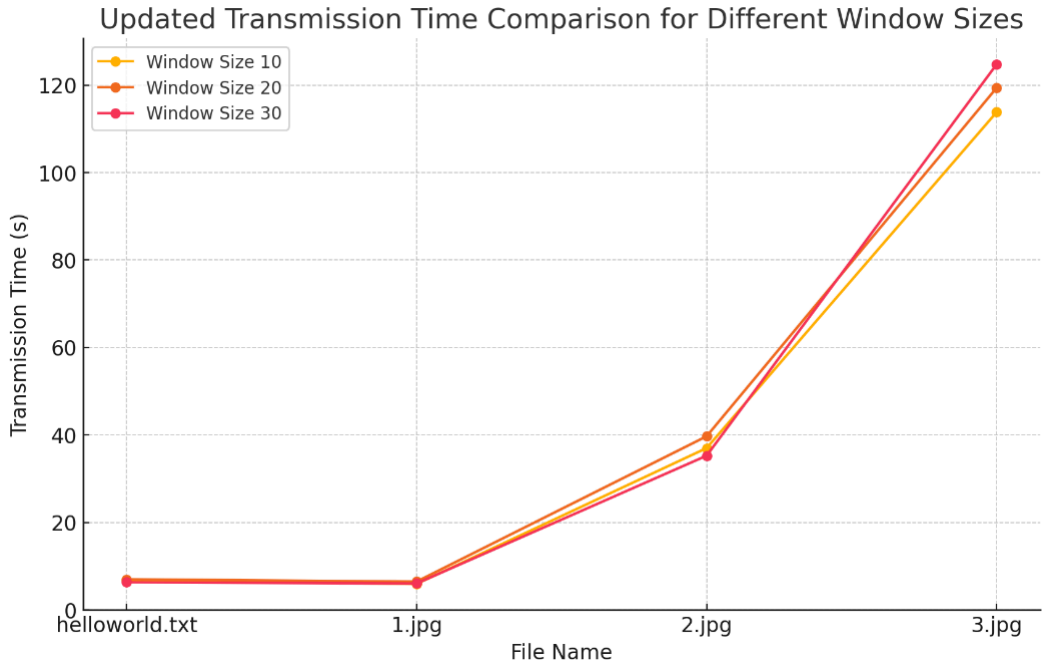
文件	吞吐率/kbs	时间/s
helloworld.txt	1598.47	6.99
1.jpg	1826.55	6.46
2.jpg	1173.13	39.69
3.jpg	1233.25	119.37

窗口大小为30：

文件	吞吐率/kbs	时间/s
helloworld.txt	1828.34	6.31
1.jpg	1896.46	6.02
2.jpg	1179.95	35.25
3.jpg	679.93	124.74

结果可视化如下：





1. **低延时低丢包环境**

   （1）网络特性及小窗口瓶颈

   在低延时环境下，网络传播延迟极小，数据包的传输时间和往返时间短暂，这使得每个数据包在发送与接收之间的时间损耗极低，进而为频繁的数据交换创造了良好条件。然而，即便处于如此优越的低延时环境中，较小的窗口仍会面临诸多潜在问题。由于数据包传输速度快，往返时间（RTT）短，网络能够在短时间内积累大量传输机会。此时，若窗口过小，每次仅能发送一个数据包，那么发送端必然会频繁陷入等待接收端确认（ACK）的状态。例如，当发送端发送一个数据包后，尽管该数据包可能已迅速抵达接收端，但发送端必须等待接收端返回确认信息才能继续发送下一个数据包。这种等待过程会造成网络资源的严重浪费，因为在等待期间，网络带宽处于闲置状态，无法得到充分利用。同时，窗口较小意味着发送端的等待时间在整个传输过程中所占比重较大，这直接限制了吞吐率的提升，导致数据传输效率低下。

   （2）大窗口提升吞吐率的原理

   与之相反，当使用更大的窗口时，情况则截然不同。大窗口允许发送端一次性发送多个数据包而无需逐个等待确认。在低延时环境下，网络延迟小的特性使得多个数据包能够同时在网络中传输，且不会引发严重的冲突或拥塞问题。这就好比拓宽了数据传输的“管道”，增加了单位时间内能够传输的数据量。发送端可以提前准备更多数据包，并利用窗口的并行传输能力同时将它们发送出去。例如，窗口大小为 30 时，发送端可以一次性将 30 个数据包发送至网络中，在接收端处理这些数据包并返回确认信息的过程中，发送端无需闲置等待，而是可以继续准备后续数据包，从而充分利用了网络带宽，有效提高了吞吐率。

   （3）对传输时间的影响

   在低延时环境下，大窗口对传输时间也产生积极影响。以文件传输为例，对于较小文件，大窗口可一次性发送更多数据，减少了因等待确认而产生的时间间隔，从而加快了传输速度，缩短传输时间。对于较大文件，虽然整体传输数据量较大，但大窗口能够保持数据的持续发送，避免了小窗口因频繁等待确认而导致的传输中断，使得传输过程更加流畅，同样有助于缩短传输时间。不过，在某些情况下，过大的窗口可能会因网络中数据包数量过多，在接收端处理环节产生一定的延迟，导致传输时间略有增加，但总体而言，大窗口在低延时环境下对传输时间的影响利大于弊。

2. **高延时高丢包环境**

（1）小窗口导致低吞吐率的原因

在高延时环境下，网络传播延迟较长，数据包的往返时间（RTT）显著增加。这种情况下，如果采用较小的窗口，每次发送一个数据包后都必须等待长时间的确认，这将导致极低的吞吐率。例如，发送一个数据包后，发送端可能需要等待几十毫秒甚至几百毫秒才能收到确认信息，在此期间，网络带宽无法得到有效利用，处于空闲状态。由于窗口小，发送端在等待确认期间无法发送新的数据包，导致大量时间被浪费在等待上，严重影响了数据传输的效率。而且，随着网络延时的增加，小窗口的这种劣势会愈发明显，因为等待确认的时间在整个传输过程中所占比例越来越大，使得单位时间内能够传输的数据量极少，吞吐率急剧下降。

（2）大窗口减少等待时间的机制

大窗口在高延时环境中的优势主要体现在能够有效减少发送端的等待时间。当窗口较大时，发送端在发送一个数据包后，可以继续发送多个数据包，而无需等待前一个数据包的确认。例如，窗口大小为 30 时，发送端发送第一个数据包后，即使尚未收到确认，仍可继续发送后续 29 个数据包。这样一来，发送端的空闲时间大大减少，网络带宽能够得到更充分的利用。在等待确认的过程中，网络中始终有多个数据包在传输，避免了因等待单个数据包确认而造成的带宽闲置。这种方式提高了数据的并发传输能力，使得传输效率得到显著提升，从而有效提高了吞吐率。

（3）提高带宽利用率及整体传输效率

在高延时网络中，大窗口还能够提高带宽利用率，进而提升整体传输效率。由于 RTT 较大，数据包往返时间长，若采用小窗口，网络带宽在等待确认期间极易被浪费。而大窗口可以同时发送多个数据包，在等待确认的同时继续发送新数据，确保了网络带宽始终处于忙碌状态，避免了空闲带宽的浪费。例如，在发送端等待第一个数据包确认的过程中，后续数据包可以持续占用网络带宽进行传输，充分利用了这段原本被闲置的时间。此外，大窗口增加了并发传输的数据量，减少了每个数据包的等待时间在整个传输过程中所占的比重。对于单个数据包而言，其等待确认的时间相对固定，但在大窗口下，由于有更多数据包同时在传输，整体的传输效率得以提高，使得单位时间内能够传输的数据量增加，吞吐率得到提升。同时，对于大文件传输，大窗口能够更好地利用网络带宽，减少因高延时导致的传输中断，使传输过程更加连续，有助于缩短传输时间。

（三）有拥塞控制和无拥塞控制的性能比较

在引入阻塞控制算法后，丢包率和延时均会对网络性能产生影响，但**丢包率的影响通常远大于延时**。阻塞控制算法通过让发送方等待接收方确认来保障数据传输可靠性具体来说，当延时增加时，数据包等待确认时间变长，网络带宽无法充分利用，吞吐率随之降低，不过**数据传输本身仍可继续，只是效率较低**。与之相比，丢包带来的后果更为严重。一旦发生丢包，无论是确认包还是数据包丢失，都会触发重传机制。每次重传不仅会占用额外网络资源、增加网络负载，还可能导致发送方误以为数据未成功送达而多次重传，进而造成吞吐率大幅下降。在引入滑动窗口机制的情况下，**丢包还会致使窗口停滞，无法发送新数据包，**进一步拖慢整个传输过程。尤其是在高丢包率环境下，重传开销巨大，会使吞吐率急剧恶化，传输时间显著延长。综合来看，尽管延时增加也会对吞吐率产生负面影响，但相比丢包的严重后果，其影响相对较小。所以在优化网络性能时，减少丢包率通常比降低延时更为关键，特别是在采用阻塞控制机制的环境中。所以在接下来的实验中我将**以丢包率为重点**进行实验。

丢包10%：

文件	无阻塞控制吞吐率/kbs	无阻塞控制时间/s	有阻塞控制吞吐率/kbs	有阻塞控制时间/s
helloworld.txt	3806.24	12.20	2705.65	4.71
1.jpg	4041.82	13.32	2842.76	4.97
2.jpg	1381.86	79.72	927.66	42.87

文件	无阻塞控制吞吐率/kbs	无阻塞控制时间/s	有阻塞控制吞吐率/kbs	有阻塞控制时间/s
3.jpg	1426.17	165.58	963.11	80.69

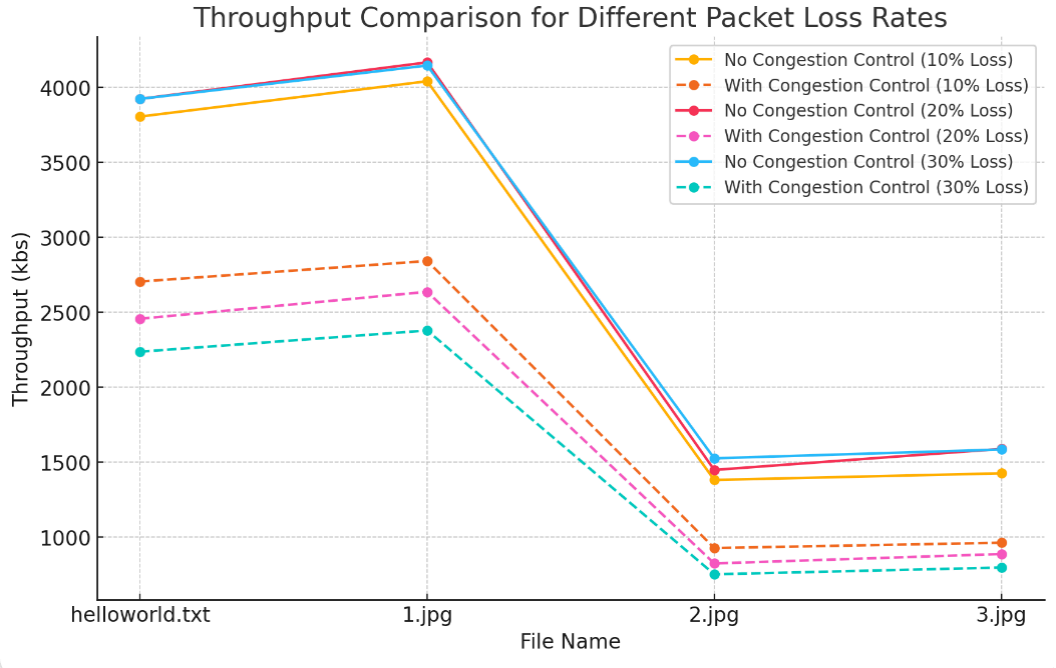
丢包20%：

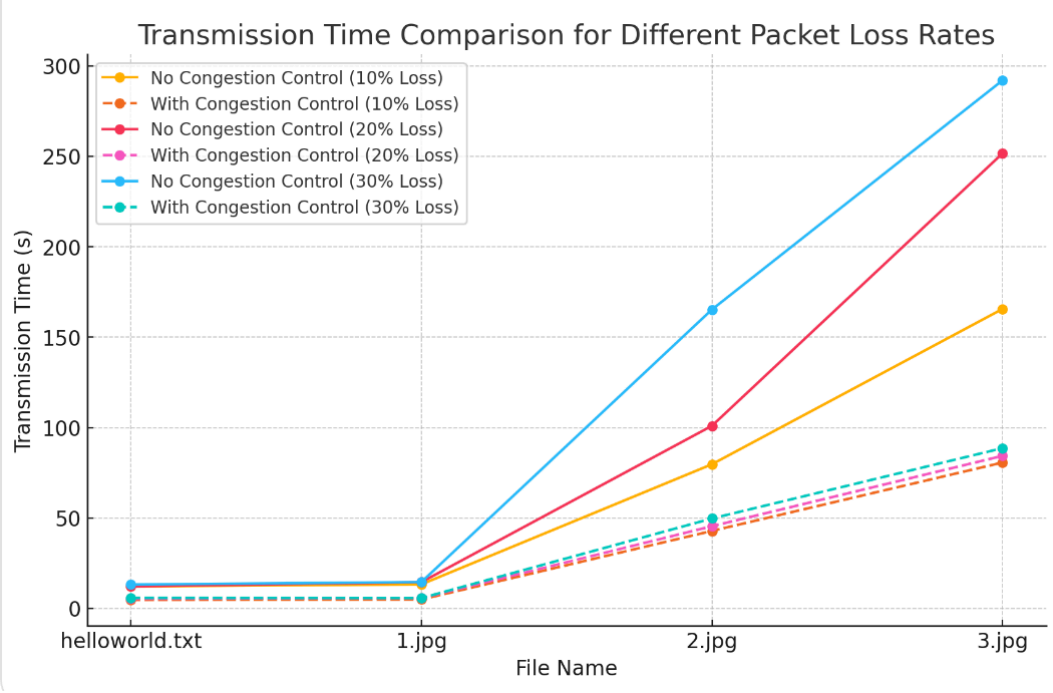
文件	无阻塞控制吞吐率/kbs	无阻塞控制时间/s	有阻塞控制吞吐率/kbs	有阻塞控制时间/s
helloworld.txt	3924.24	12.24	2457.53	5.34
1.jpg	4167.82	14.54	2637.14	5.68
2.jpg	1448.86	100.97	824.83	45.47
3.jpg	1589.17	251.58	887.86	84.36

丢包30%：

文件	无阻塞控制吞吐率/kbs	无阻塞控制时间/s	有阻塞控制吞吐率/kbs	有阻塞控制时间/s
helloworld.txt	3924.24	13.25	2237.48	5.84
1.jpg	4146.82	14.59	2379.26	5.74
2.jpg	1526.23	165.26	752.58	49.74
3.jpg	1584.98	291.95	797.58	88.63

结果可视化如下：





1. 低丢包率（10%）
    - 此时无拥塞控制能快速发送大量数据，利用网络带宽优势明显，吞吐率较高。但因不考虑网络拥塞，丢包后无序重传，导致传输时间长。有拥塞控制虽限制了发送速率，但保证了数据包的有序性和可靠性，重传次数少，传输时间短，不过也牺牲了一定的吞吐率。
2. 中丢包率（20%）
    - 无拥塞控制吞吐率仍高于有拥塞控制，但随着丢包率增加，其性能受影响更大。因为更多的丢包需要更多重传，无序重传导致网络资源浪费和冲突增加，传输时间进一步延长。有拥塞控制通过更谨慎地调整发送速率，能更好地适应丢包情况，保持相对稳定的性能，传输时间虽有所增加但增幅小于无拥塞控制。
3. 高丢包率（30%）
    - 无拥塞控制在高丢包率环境下，其吞吐率方面的优势依旧得以体现，然而，由于丢包现象严重，致使大量的重传操作频繁发生。这些重传多为无效重传，它们使得网络资源被极大地浪费在重复且无意义的数据传输上，进而导致传输时间急剧增加。与之相反，有拥塞控制在高丢包率情形中，通过严谨地把控发送速率，有效地避免了过多数据包的发送以及不必要的重传情况。这样的做法防止了因过多数据包在网络中无序堆积和频繁重传而引发的网络性能急剧恶化问题，尽管其吞吐率相对较低，但传输时间的增长幅度得到了一定程度的限制，整体性能表现较为稳定。

在网络环境中，若丢包率较低且对实时性要求不高，无拥塞控制可能在一定程度上获得较高吞吐率。但在大多数实际情况中，尤其是网络状况不稳定、丢包率较高时，有拥塞控制机制能更好地保证数据传输的可靠性和稳定性，减少传输时间，虽然吞吐率可能有所降低，但整体网络性能更优。因此，在网络协议设计和优化时，应根据具体的应用场景和网络条件，合理选择是否采用拥塞控制以及采用何种拥塞控制算法，以平衡吞吐率和传输时间等性能指标，提高网络的整体性能和用户体验。

总结：这次实验，我深刻体会到网络协议设计与优化的复杂性。在不同机制的对比中，明白了滑动窗口机制在高效利用带宽方面的优势，以及拥塞控制在应对网络不稳定时的重要性。实验过程中遇到的各种问题促使我深入思考网络传输原理，也让我认识到实际应用中需根据具体场景权衡性能指标。这次实验测试变量之多、操作量之大以及期末周门门紧逼的ddl曾一度让我处于崩溃边缘，但计网实验的结束标志着所有事情一步一步朝着更好的方向发展，我相信这段时间的经历会对今后的学习生活大有裨益。最后，感谢助教老师的包涵和悉心指导！