计算机网络 Lab4

一、实验目的

1. 基于 UDP 进行可靠文件传输，比较 GBN 和 SR 的区别。
2. 实现基于丢包和基于延迟的拥塞控制，加强对拥塞控制算法的理解。
3. 在模拟和真实网络环境中进行测试，熟悉网络算法测试流程。
4. 进行数据可视化和结果解释，提升数据处理和分析能力

二、实验任务

测试编号	环境	可靠传输算法	拥塞控制算法	测试内容
1	模拟	GBN/SR	基于丢包	有效吞吐量 随 丢包率 变化
2	模拟	GBN/SR	基于丢包	流量利用率 随 丢包率 变化
3	模拟	SR	基于丢包/延迟	有效吞吐量 随 丢包率 变化
4	模拟	SR	基于丢包/延迟	流量利用率 随 丢包率 变化
5	模拟	SR	基于丢包/延迟	有效吞吐量 随 延迟 变化
6	模拟	SR	基于丢包/延迟	流量利用率 随 延迟 变化
7	真实	GBN/SR	基于丢包	有效吞吐量 随 上传文件大小 变化
8	真实	GBN/SR	基于丢包	流量利用率 随 上传文件大小 变化
9	真实	SR	基于丢包/延迟	有效吞吐量 随 下载文件大小 变化
10	真实	SR	基于丢包/延迟	流量利用率 随 下载文件大小 变化

三、设计思路

3.1两种可靠传输

Go-Back-N (GBN) 协议：

GBN 是一种基于窗口的滑动窗口协议，它允许发送方在未收到确认的情况下连续发送多个数据包，但一旦发生丢包或错误，发送方必须回退到出错的数据包，并重新发送从该数据包开始的所有后续数据包。

Selective Repeat (SR) 协议：

SR 协议相比 GBN 更加高效和复杂，它允许发送方仅重传那些确实丢失或出错的数据包，而不需要回退重传整个窗口。接收方也需要维护一个接收窗口，能够处理乱序到达的数据包，并及时发送对应的 ACK。

3.2 两种拥塞控制

基于丢包的拥塞控制：

设计思路：

基于丢包的拥塞控制算法通过检测数据包的丢失来调整发送窗口大小，我的实现参考了 **TCP Reno**。当发生丢包时，算法会认为网络出现拥塞，降低发送速率；当网络状况良好时，逐步增加发送速率。

基于延迟的拥塞控制：

设计思路：

基于延迟的拥塞控制算法通过监测 RTT（往返时间）的变化来推断网络拥塞状况。我的实现参考 **TCP Vegas**。当检测到 RTT 增加时，说明可能出现网络拥塞，因此需要减少发送速率；当 RTT 保持稳定或减小时，可以增加发送速率。

四、实现细节

4.1两种可靠传输

Go-Back-N (GBN) 协议

实现细节：

1. **窗口管理：**

- **发送窗口**：客户端维护一个发送窗口，窗口大小可调（初始值设为 1）。`base` 表示当前未被确认的最小序列号，`next_seq_num` 表示下一个要发送的数据包序列号。
- **接收窗口**：服务器端在 GBN 模式下只需维护一个期望接收的序列号 `expected_seq_num`。

2. **数据包发送：**
   - 客户端在 `send_packets` 线程中，不断检查 `next_seq_num` 是否小于 `base + window_size`，若满足条件则发送数据包。
   - 每发送一个数据包，客户端记录发送时间以计算 RTT，并启动一个定时器（在 GBN 中仅为窗口的第一个数据包启动一个定时器）。
3. **确认机制：**
   - 客户端在 `receive_acks` 线程中接收服务器的 ACK 包。
   - 若收到的 ACK 序号 `ack_num` 大于等于 `base`，则更新 `base` 为 `ack_num + 1`，并取消已确认数据包的定时器。
   - 若所有数据包均被确认，客户端发送一个 FIN 包表示传输完成
4. **超时与重传：**
   - 若定时器超时（即未收到 ACK），客户端将 `ssthresh` 调整为窗口大小的一半，并将窗口大小重置为 1。
   - 客户端重新发送从 `base` 开始的所有未确认的数据包，并重新启动定时器。

Selective Repeat (SR) 协议：

实现细节：

1. **窗口管理：**
   - **发送窗口**：客户端维护一个发送窗口，窗口大小可调。`base` 表示当前未被确认的最小序列号，`next_seq_num` 表示下一个要发送的数据包序列号
   - **接收窗口**：服务器端在 SR 模式下维护一个接收窗口，能够接受并缓存乱序到达的数据包。
2. **数据包发送：**
   - 客户端在 `send_packets` 线程中，不断检查 `next_seq_num` 是否小于 `base + window_size`，若满足条件则发送数据包。
   - 对于每个发送的数据包，客户端记录发送时间并启动独立的定时器
3. **确认机制：**
   - 服务器端在 `handle_sr` 方法中处理 SR 模式下的数据包接收。
   - 每收到一个数据包，无论是否按序，服务器都会发送对应的 ACK 包。
   - 客户端在 `receive_acks` 线程中处理 ACK，确认特定序列号的数据包
4. **超时与重传：**

- 若某个数据包的定时器超时，客户端仅重传该特定的数据包，而不影响其他数据包
- 客户端调整 `ssthresh` 并根据拥塞控制算法调整窗口大小。

4.2 两种拥塞控制

基于丢包的拥塞控制：

实现细节：

1. **窗口调整：**
   - 初始窗口大小设为 1。
   - 当收到一个 ACK，若当前窗口大小于 `ssthresh`，则采用加倍增长（即窗口大小翻倍）；否则，线性增长窗口大小。
   - 当检测到丢包（通过超时或三次重复 ACK），将 `ssthresh` 设为窗口大小的一半，并将调整窗口大小。
2. **实现方式：**
   - 在 `receive_acks` 方法中，根据是否收到新的 ACK 来调整窗口大小。
   - 在 `handle_timeout` 方法中，检测到超时事件后，调整 `ssthresh` 和窗口大小。

基于延迟的拥塞控制：

实现细节：

1. **RTT 估计：**
   - 通过测量数据包的发送时间和收到 ACK 的时间，估计 RTT。
   - 使用指数加权移动平均（EWMA）算法来平滑 RTT 估计值。

     $$estimated\_RTT = (1-\alpha)\times estimated\_RTT + \alpha \times sample\_RTT$$

     $$dev\_RTT = (1-\beta)\times dev\_RTT + \beta \times \mid sample\_RTT - estimated\_RTT \mid$$

     衰减因子α控制新样本对估计值的影响程度。衰减因子β控制偏差估计的更新速度。
2. **窗口调整：**
   - 维护 `estimated_RTT` 和 `dev_RTT`（RTT 的偏差）。
   - 根据 `estimated_RTT` 和 `base_RTT`（初始 RTT）计算网络延迟差异。
   - 若延迟差异小于阈值，则增加窗口大小；若延迟差异大于阈值，则减小窗口大小。
3. **实现方式：**
   - 在 `receive_acks` 方法中，更新 RTT 估计值。

- 在 `adjust_window_delay` 方法中，根据 RTT 的变化调整窗口大小。

五、输入输出示例

5.1服务器端

启动命令：

```
python server.py --protocol SR(GBN) --congestion delay(loss)
```

```
PS C:\Users\IScream\Desktop\study\计算机网络\2024实验\lab4> python server.py --protocol SR --congestion delay
Server started, waiting for data...
```

下载(download):

```
All packets ACKed by ('192.168.56.1', 53595).
Sent MD5 checksum to ('192.168.56.1', 53595)
Sent FIN to ('192.168.56.1', 53595)
Sent MD5 checksum to ('192.168.56.1', 53595)
Sent FIN to ('192.168.56.1', 53595)
File transfer to ('192.168.56.1', 53595) completed.

--- Performance Metrics ---
File size: 5242880 bytes
Transfer time: 55.20 seconds
Effective throughput: 94982.60 bytes/second
Total data sent (including retransmissions): 5339208 bytes
Flow utilization rate: 0.9820
```

上传(upload):

```
Received packet 5116 from ('192.168.56.1', 61421)
Received packet 5117 from ('192.168.56.1', 61421)
Received packet 5118 from ('192.168.56.1', 61421)
Received packet 5119 from ('192.168.56.1', 61421)
Received FIN from ('192.168.56.1', 61421), closing connection.
MD5 of received file from ('192.168.56.1', 61421): 5f363e0e58a95f06cbe9bbc662c5dfb6
Sent MD5 checksum to ('192.168.56.1', 61421)
Connection with ('192.168.56.1', 61421) closed.
```

5.2客户端

启动命令：

```
python client.py <SERVER_IP> testfile_*MB --protocol SR(GBN) --
congestion delay(loss) --operation download(upload)
```

下载(download):

```
Received packet 5114
Received packet 5115
Received packet 5116
Received packet 5117
Received packet 5118
Received packet 5119
Local MD5: 5f363e0e58a95f06cbe9bbc662c5dfb6
Received MD5: 5f363e0e58a95f06cbe9bbc662c5dfb6
MD5 checksum matches. File transfer successful.
File download completed.
hyf@ubuntu:~/ComputerNetwork$
```

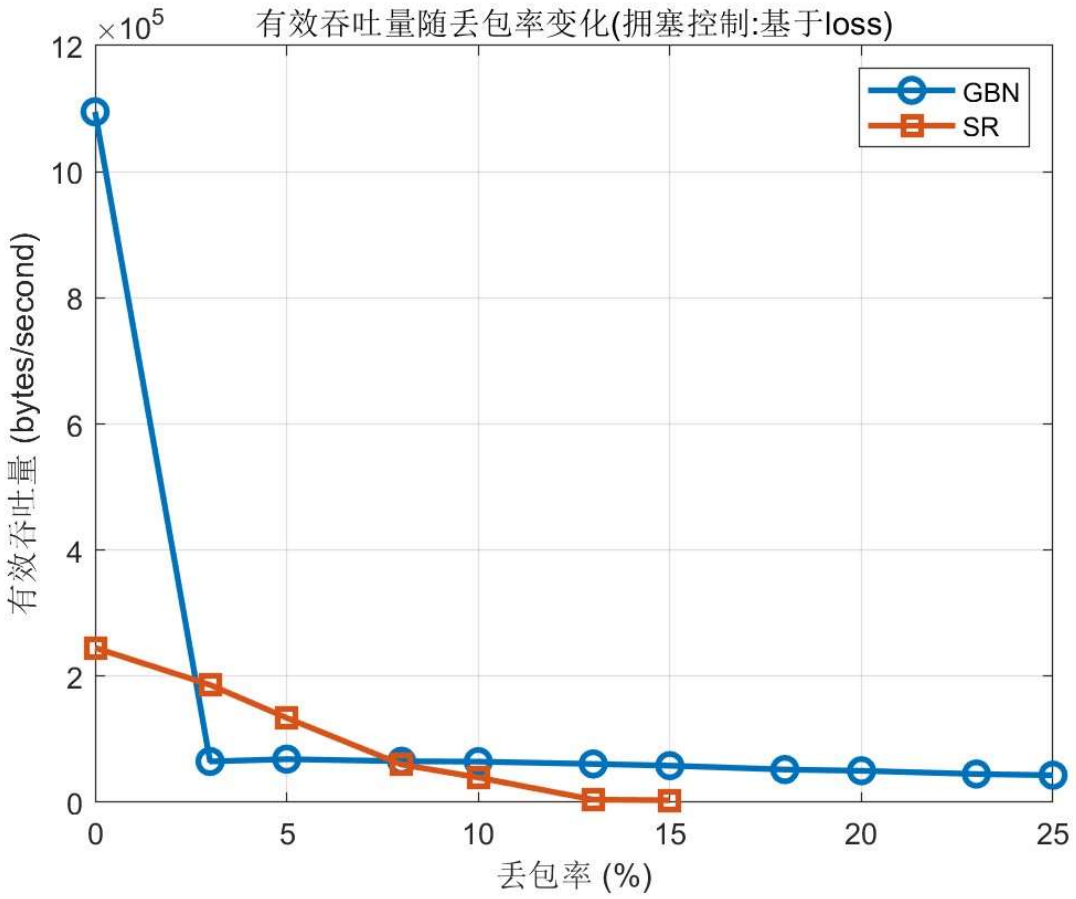
上传(upload):

```
All packets ACKed. Sending FIN.
Received ACK 0, window moves to 5120
Local MD5: 5f363e0e58a95f06cbe9bbc662c5dfb6
Received MD5: 5f363e0e58a95f06cbe9bbc662c5dfb6
MD5 checksum matches. File transfer successful.
File upload completed.

--- Performance Metrics ---
File size: 5242880 bytes
Transfer time: 62.78 seconds
Effective throughput: 83509.17 bytes/second
Total data sent (including retransmissions): 5335040 bytes
Flow utilization rate: 0.9827
```

六、测试结果及解释

1.测试1



参数设置：

测试文件大小：10MB 模式：upload 延迟：0ms

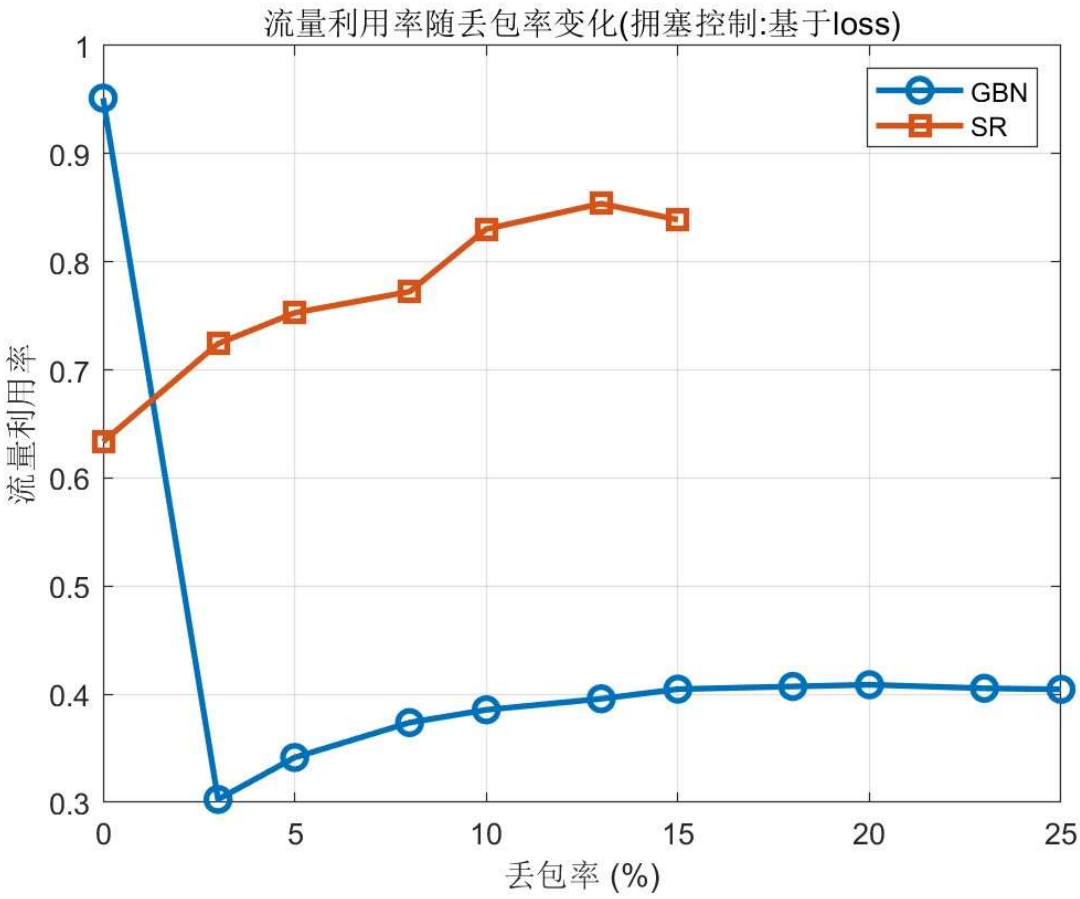
现象解释：

GBN 在丢包率增加时，吞吐量下降得非常快，在丢包率较低（例如 5%）时，已经下降到一个较低的水平。这是因为GBN 的特性是当检测到丢包时，它必须回退并重传整个窗口中的所有数据包。因此，丢包率的增加会导致大量的重传，导致吞吐量下降得更快。在丢包率较高的情况下，GBN 的重传开销过大，导致有效吞吐量显著下降。

SR 仅重传丢失的数据包，其吞吐量整体表现比 GBN 更加稳定，但从测得的数据看有效吞吐量比GBN更低。可能是因为

1. SR 协议需要接收方对每一个数据包设置单独的定时器、单独发送 ACK，且需要在接收方缓存乱序到达的数据包，会带来额外的内存和处理开销，从而影响 SR 的实际吞吐量。
2. 在 SR 中，每个超时的包会触发重传操作，而 GBN 中只需要重传第一个超时包。频繁的超时重传可能导致 SR 的吞吐量低于 GBN。
3. SR 的 ACK 包更分散，如果网络条件较差，ACK 包也容易丢失，导致发送方对个别数据包频繁重传，从而增加传输时延和资源消耗

2.测试2



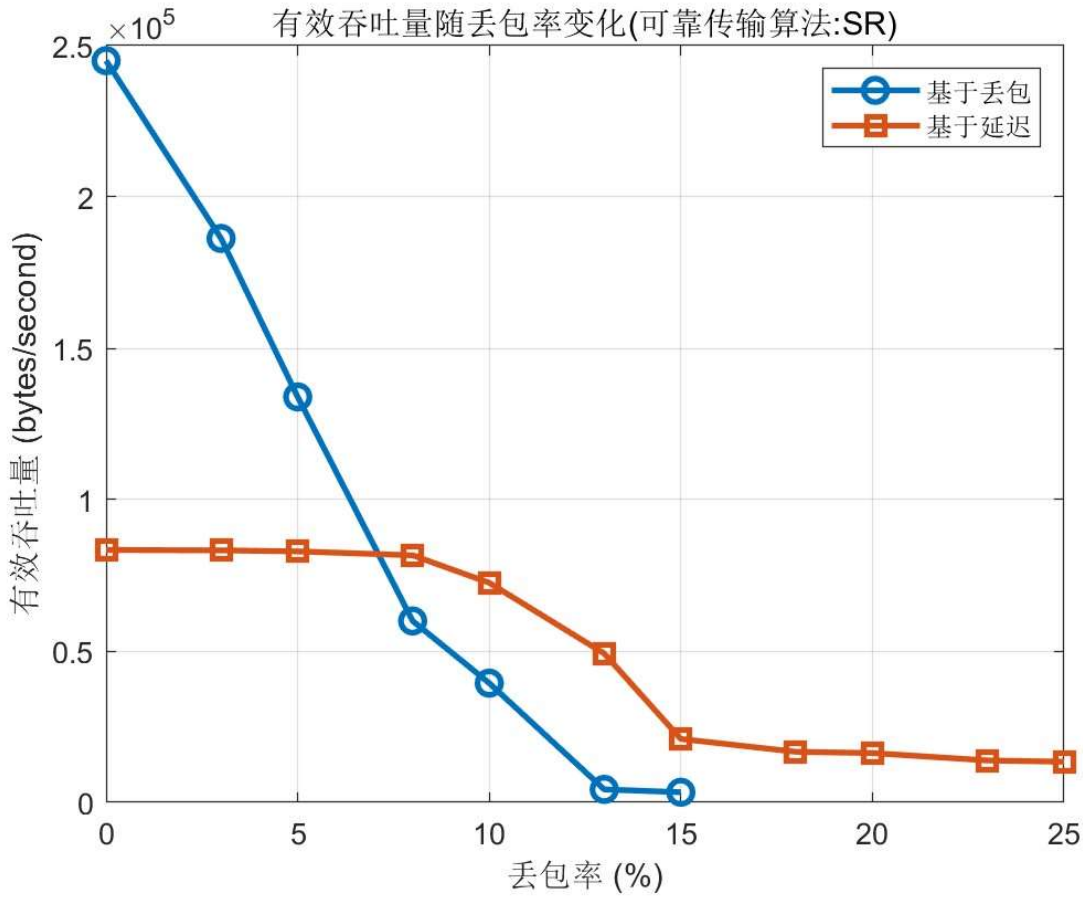
参数设置：

测试文件大小：10MB 模式：upload 延迟：0ms

现象解释：

- **GBN** 在丢包率增加时，流量利用率迅速下降。由于 GBN 的回退机制，它在丢包时需要重新发送整个窗口的数据包。这导致大量的冗余传输，尤其是在高丢包率下，流量利用率较低
- **SR** 的流量利用率随着丢包率的增加整体表现比 GBN 更加稳定，在高丢包率下依然保持较高的流量利用率。SR 仅重传丢失的数据包，从而减少了不必要的重传开销，保持了较高的流量利用率

3.测试3



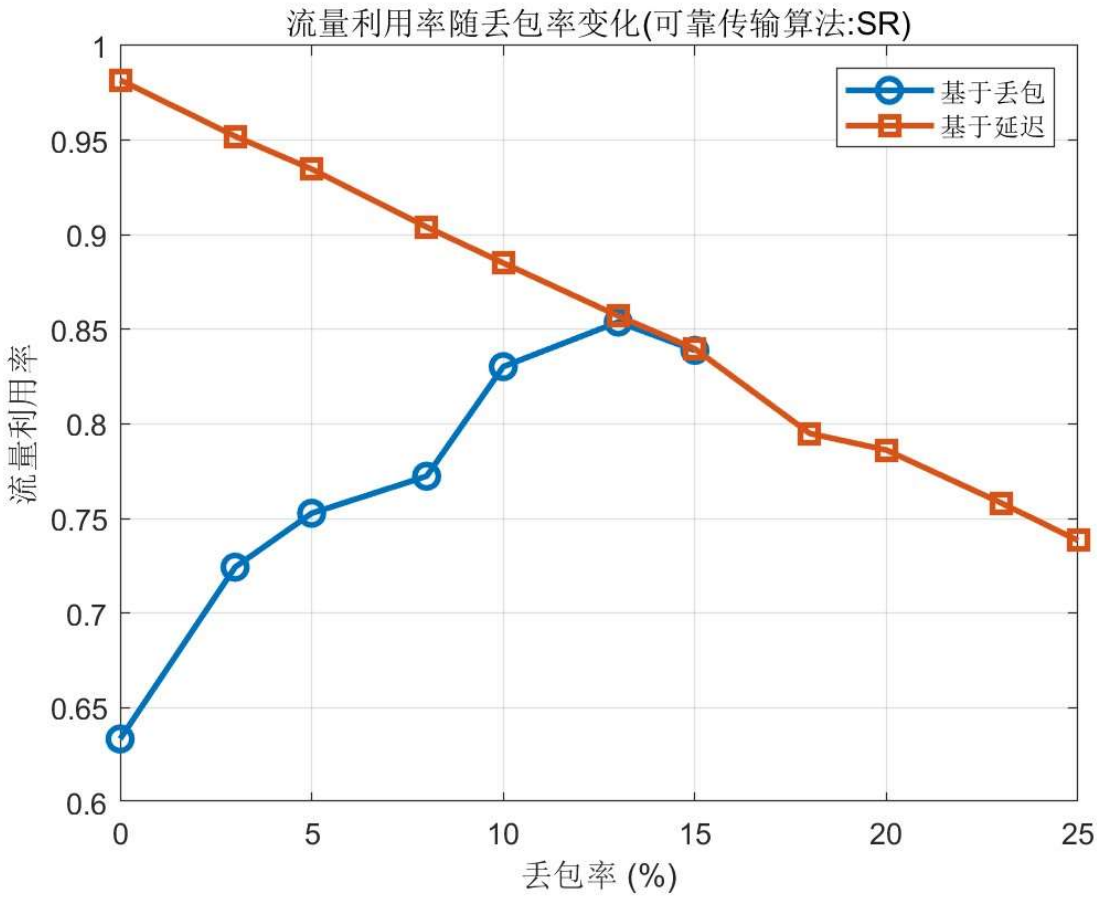
参数设置：

测试文件大小：10MB 模式：upload 延迟：0ms

现象解释：

- **基于丢包的拥塞控制**在低丢包率时的吞吐量较高，但随着丢包率增加，吞吐量下降较快。该控制方式主要依赖丢包事件来判断拥塞，一旦检测到丢包，就会减小发送窗口，从而降低有效吞吐量。在丢包率较低时，基于丢包的策略能够充分利用带宽，达到较高的吞吐量；但随着丢包率上升，窗口收缩频率增加，导致吞吐量迅速下降
- **基于延迟的拥塞控制**的吞吐量随着丢包率增加下降较慢，且在丢包率较高时（10%以上）明显优于基于丢包的策略。基于延迟的拥塞控制通过监测 RTT 的变化来调整发送窗口，在检测到 RTT 增加（潜在拥塞）时主动降低窗口大小，从而避免拥塞。因为它并不完全依赖丢包信号来调整窗口，丢包率上升时，其表现更稳定，即使在高丢包率下，也能保持一定的吞吐量。

4.测试4



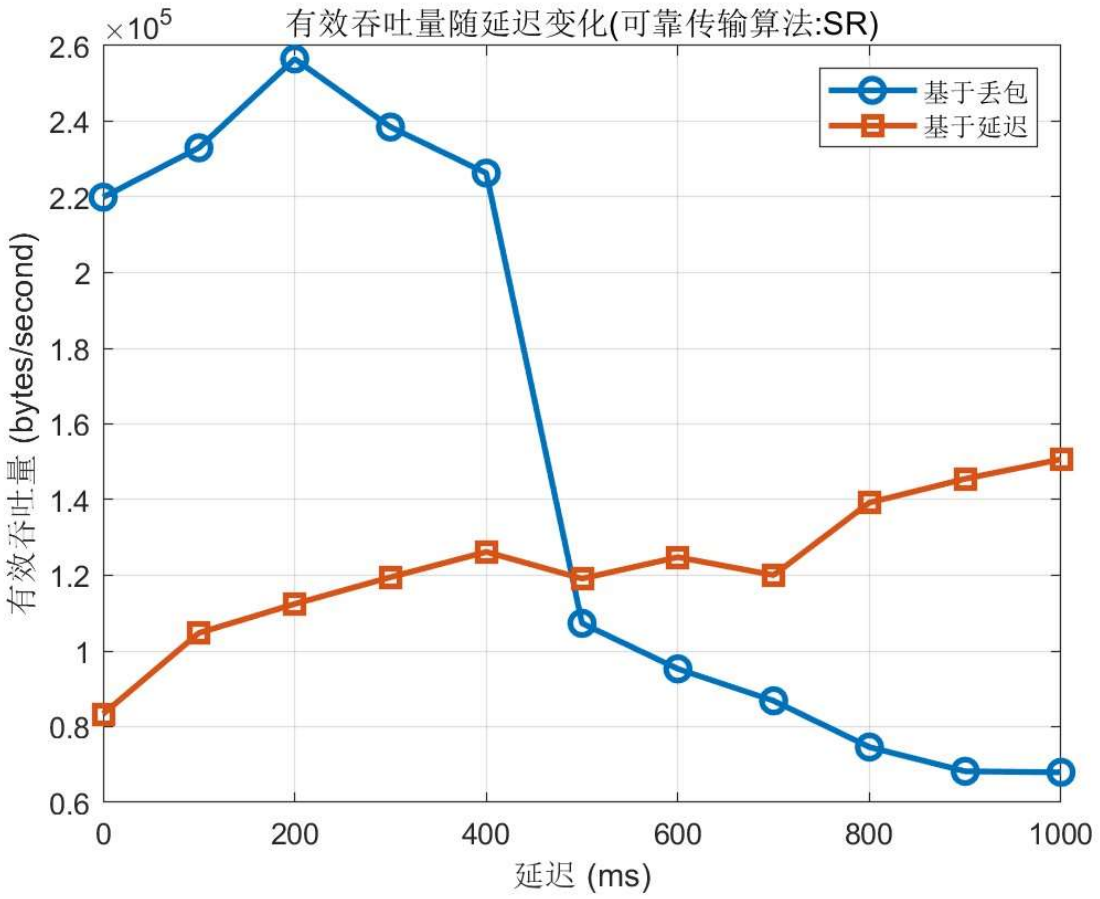
参数设置：

测试文件大小：10MB 模式：upload 延迟：0ms

现象解释：

- **基于丢包的拥塞控制**的流量利用率在丢包率增加时呈上升趋势，但整体略低于基于延迟的策略。在低丢包率情况下，基于丢包的策略因其较高的重传量，导致流量利用率相对较低。随着丢包率增加，基于丢包的拥塞控制策略在高丢包率下非常保守，倾向于减少发送速率和窗口大小，冗余数据的占比下降，流量利用率相对提高。
- **基于延迟的拥塞控制**的流量利用率在低丢包率时接近 100%，并随着丢包率增加逐渐下降。基于延迟的控制方式主动调整窗口，在低丢包率下保持高流量利用率（接近1）。随着丢包率上升，网络的负载和拥塞通常也会增加，从而导致 RTT 上升。这会被延迟控制识别为潜在的拥塞，导致传输的有效数据量逐步下降，降低流量利用率

5.测试5



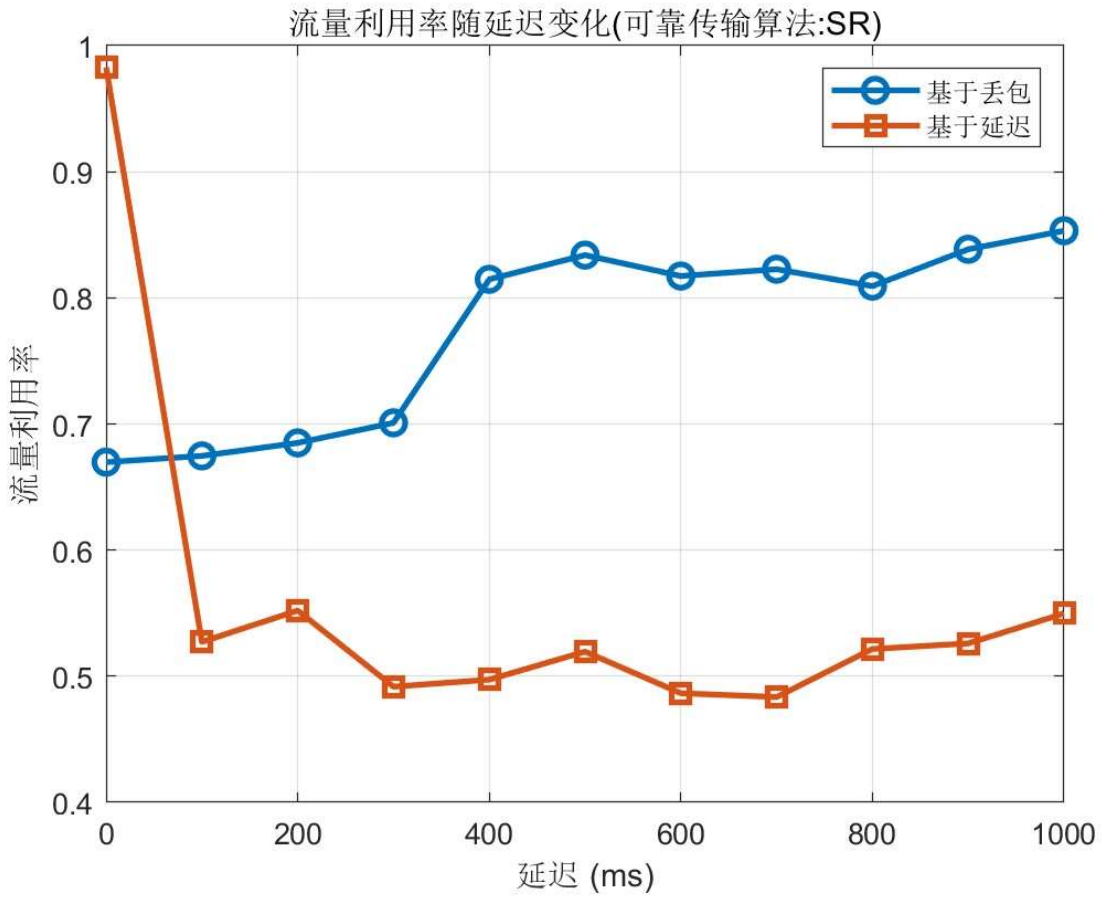
参数设置：

测试文件大小：10MB 模式：upload 丢包：0%

现象解释：

- **基于丢包的拥塞控制**在低延迟时表现出较高的吞吐量，但随着延迟增加，吞吐量出现明显下降。在低延迟环境中，丢包率较低时，基于丢包的控制策略可以充分利用带宽，实现较高的吞吐量。随着延迟增加，丢包信号的传递和确认反馈的时间也增加，导致超时和重传的概率提高，发送窗口缩小，吞吐量大幅下降。延迟较高时，重传的滞后效应变得明显，进一步限制了有效吞吐量。
- **基于延迟的拥塞控制**在延迟增加的情况下，吞吐量的下降相对缓慢，并在高延迟时的吞吐量优于基于丢包的控制策略。基于延迟的控制策略通过监测 RTT 增量来调整发送窗口，与丢包信号相比，延迟信号能更快地检测到网络的拥塞，从而提前降低发送速率，避免因过度重传而导致的吞吐量下降。尽管延迟较高时 RTT 的波动会导致窗口收缩，但由于控制策略更适应高延迟环境，吞吐量相对平稳。

6.测试6



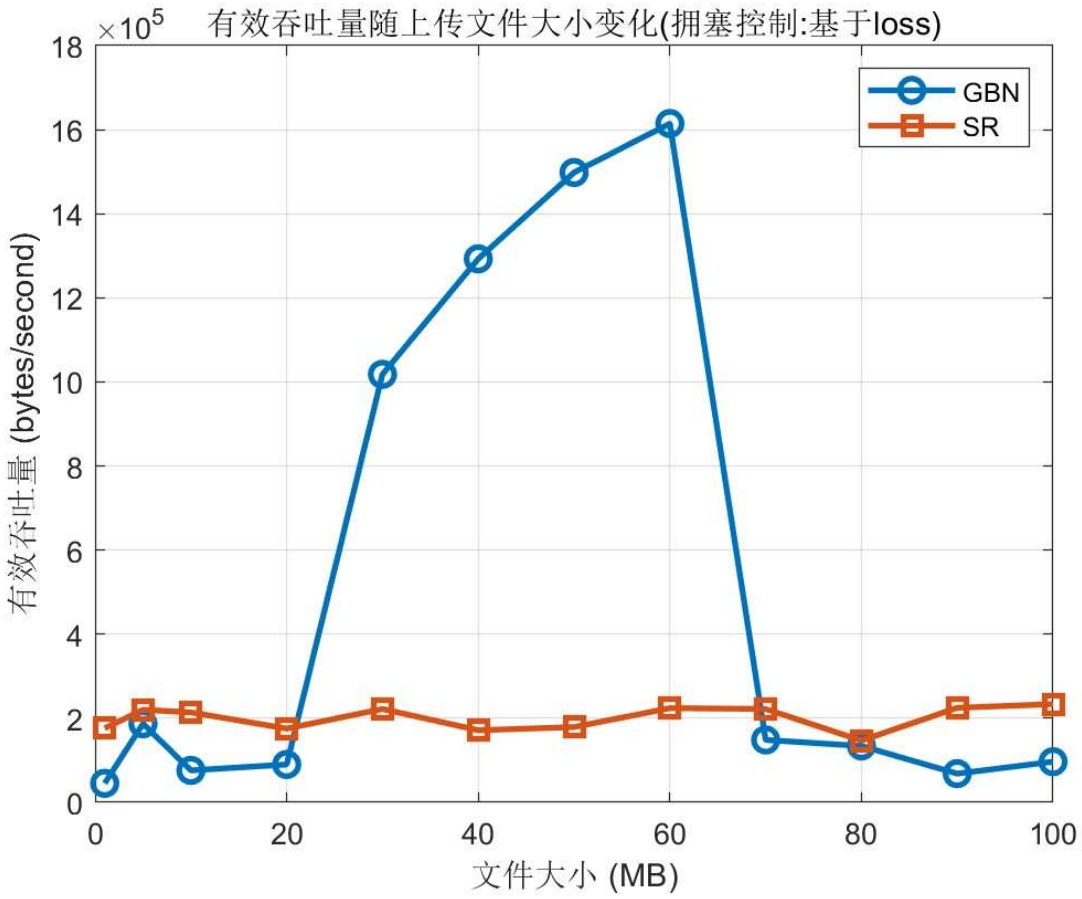
参数设置：

测试文件大小：10MB 模式：upload 丢包：0%

现象解释：

- **基于丢包的拥塞控制**在延迟较低时的流量利用率较高，但随着延迟增加，流量利用率逐渐上升并趋于稳定。在低延迟下，基于丢包的控制策略能够较好地控制重传，保持较高的流量利用率。随着延迟增加，重传的反馈变得滞后，发送窗口收缩减少了冗余数据包的数量，反而提升了有效数据的比例，使得流量利用率逐渐上升。在高延迟环境下，控制机制较保守的窗口增长策略有效减少了冗余流量，使得流量利用率趋于稳定。
- **基于延迟的拥塞控制**在延迟增加的情况下，流量利用率下降明显，但在高延迟时逐步回升。在低延迟环境中，基于延迟的控制策略能够准确控制窗口大小，保持接近 100% 的流量利用率。随着延迟增加，RTT 的波动导致发送窗口频繁收缩，出现过多的控制操作，流量利用率因此下降。但在延迟进一步增高时，策略逐步适应了延迟环境，重传量减少，流量利用率略有回升。

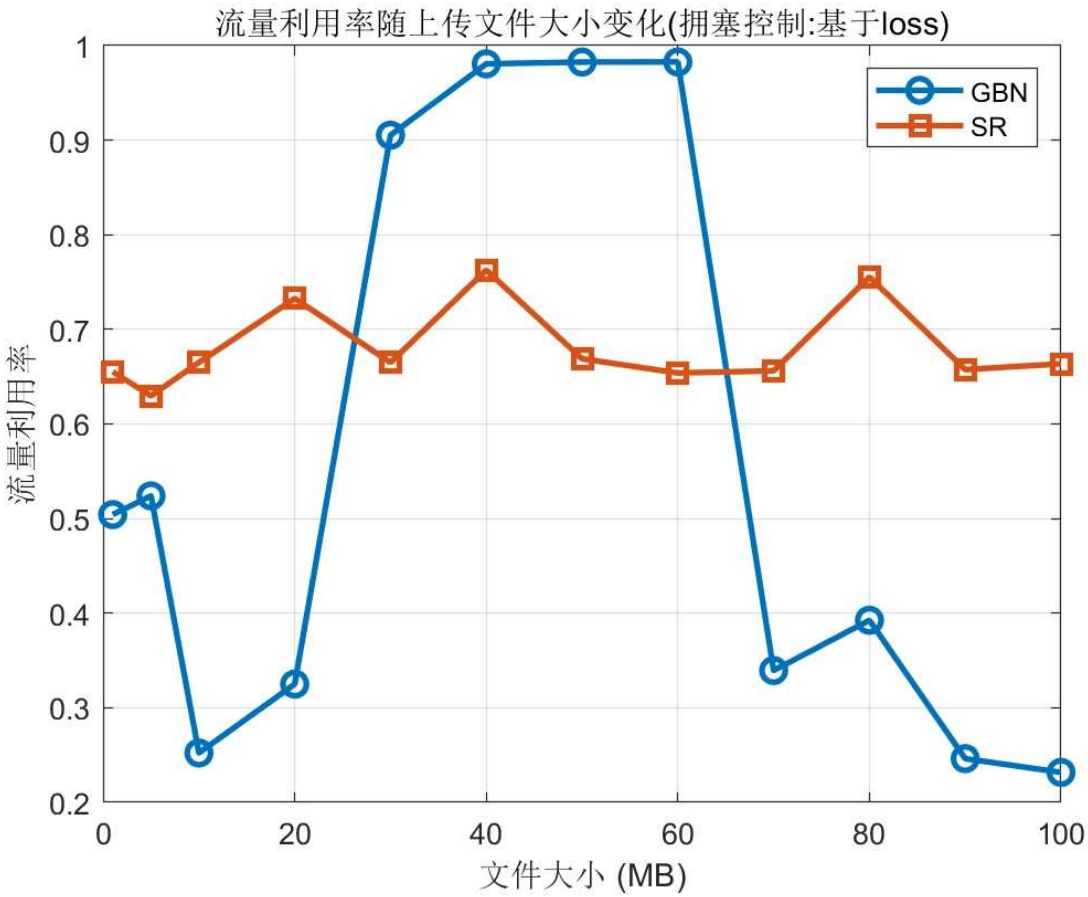
7.测试7



现象解释：

- **GBN** 在文件大小较小（10-40MB）时，吞吐量逐渐上升，并在中等文件大小（50-60MB）时达到峰值，但随着文件大小继续增加，吞吐量显著下降。在小文件大小下，GBN 的窗口增长较快，重传量较小，吞吐量逐渐增加。在文件大小增加到一定程度（约50MB）后，GBN 的发送窗口能够有效利用带宽，吞吐量达到最大值。当文件进一步增大，传输时间较长，累积的丢包和超时事件增加，使得 GBN 的回退重传开销变大，吞吐量迅速下降。
- **SR** 的吞吐量在不同文件大小下较为平稳，但整体低于 GBN。SR 协议在传输过程中能选择性地重传丢失的包，因此随着文件大小增大，其吞吐量相对平稳。由于 SR 的重传机制更细粒度，即使文件较大，也不会产生大量冗余重传，导致吞吐量变化较小。虽然 SR 的吞吐量较为稳定，但由于其额外的确认和缓存开销，整体吞吐量稍低于 GBN。

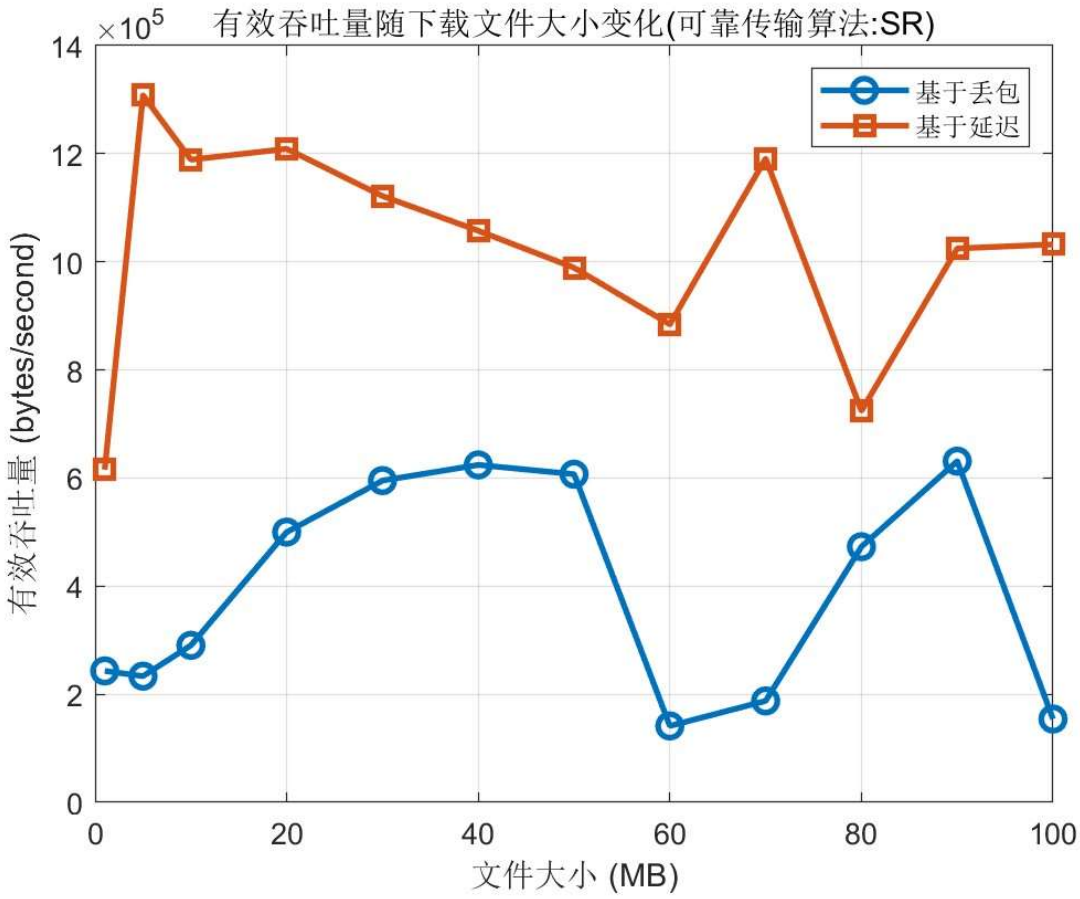
8.测试8



现象解释：

- **GBN** 在文件大小较小（20MB以下）和较大（80MB以上）时流量利用率较低，但在中等文件大小时（约50MB），流量利用率显著上升并接近 1。在文件较小时，GBN 的发送窗口增长有限，容易因为频繁的超时和回退导致重传量较大，流量利用率较低。在中等文件大小时，GBN 可以保持较大的窗口，减少回退次数，达到较高的流量利用率。对于大文件传输，随着时间推移和丢包累积，GBN 的回退重传频繁，使得流量利用率下降。
- **SR** 的流量利用率在不同文件大小下相对稳定，在 0.6-0.8 之间波动。由于 SR 协议的选择性重传机制，其重传开销相对较小，导致流量利用率在不同文件大小下相对平稳。由于 SR 需要为每个数据包单独确认，导致一些额外的流量开销，因此流量利用率略低于 GBN 的峰值，但在大文件情况下仍保持在较高水平。

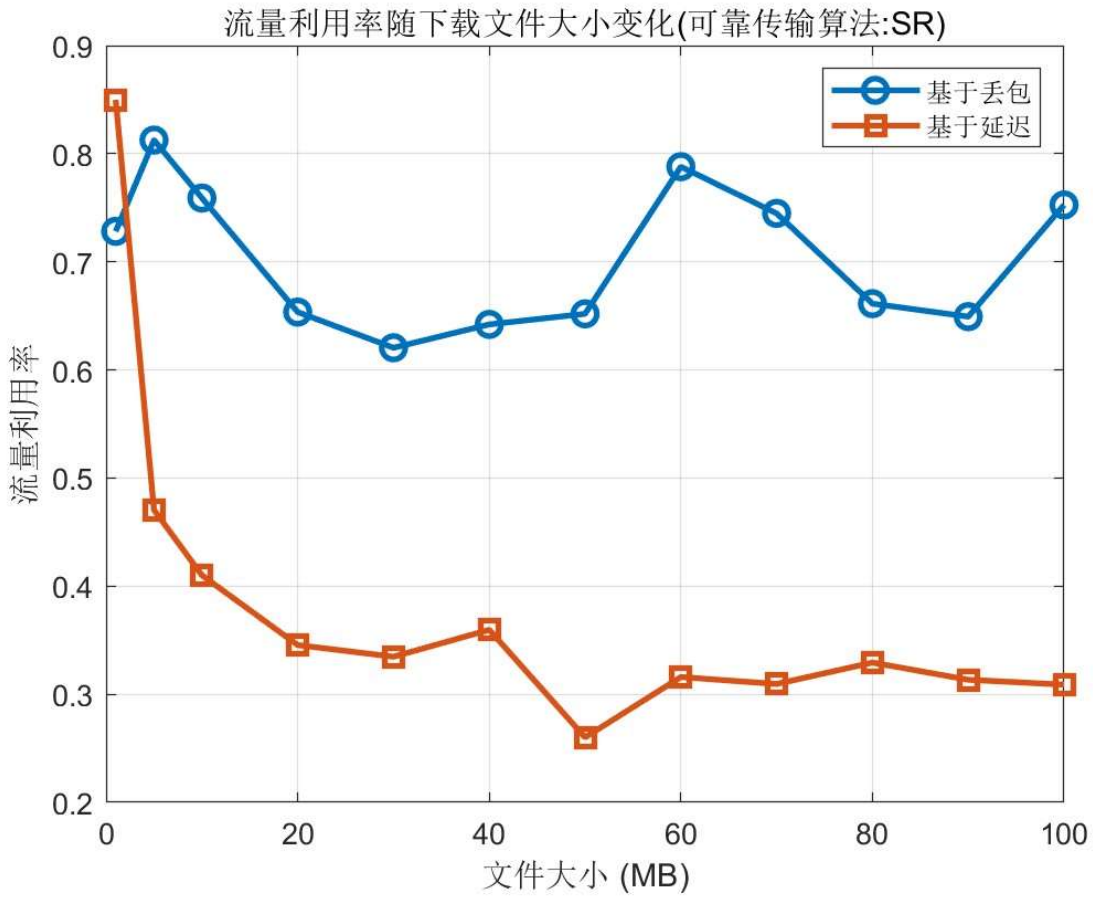
9.测试9



现象解释：

- **基于延迟的拥塞控制**在小文件下载时表现出较高的吞吐量，并随着文件大小增大而逐渐波动，但总体上较为稳定。对于小文件下载，延迟控制可以根据 RTT 快速调整窗口，从而在开始阶段迅速达到较高的吞吐量。随着文件增大，网络负载波动引起的 RTT 变化导致窗口调整频率增加，吞吐量出现波动。这种控制方式使其吞吐量在较高范围内维持，但对于更大的文件，吞吐量有略微下降的趋势。
- **基于丢包的拥塞控制**的吞吐量在文件大小增加的情况下逐渐上升，达到一定大小后维持在一个较低的波动范围内。基于丢包的控制策略在小文件下载时，由于发送窗口较小，吞吐量受限较多。随着文件大小增加，丢包事件相对分散，导致发送窗口逐渐增大，吞吐量上升。在文件较大时，传输过程中的丢包频率增加，导致吞吐量趋于稳定，且波动范围较小。

10.测试10



现象解释：

- **基于丢包的拥塞控制**的流量利用率在文件大小较小（10MB以内）时较高，随着文件大小增加逐渐下降，但在较大文件情况下趋于稳定。在小文件情况下，基于丢包的策略能够保持较高的流量利用率，因为窗口增大较快，重传较少。随着文件大小增加，重传频率增高，流量利用率下降，但在大文件传输过程中逐渐趋于稳定，维持在中等利用率水平。
- **基于延迟的拥塞控制**的流量利用率在文件大小较小时较高，但随着文件增大迅速下降，且在文件较大时保持在较低的水平。延迟控制在小文件情况下可以准确地调整窗口，保持较高的流量利用率。随着文件大小增加，延迟变化导致频繁的窗口收缩，增加了额外的控制流量开销，流量利用率迅速下降并维持在较低水平。