南開大學
Nankai University

计算机学院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议 (3-4)

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2022 年 12 月 29 日

目录

1 实验要求

基于给定的实验测试环境，通过改变延迟时间和丢包率，完成下面 3 组性能对比实验：

- 停等机制与滑动窗口机制性能对比；
- 滑动窗口机制中不同窗口大小对性能的影响；
- 有拥塞控制和无拥塞控制的性能比较。

2 协议设计

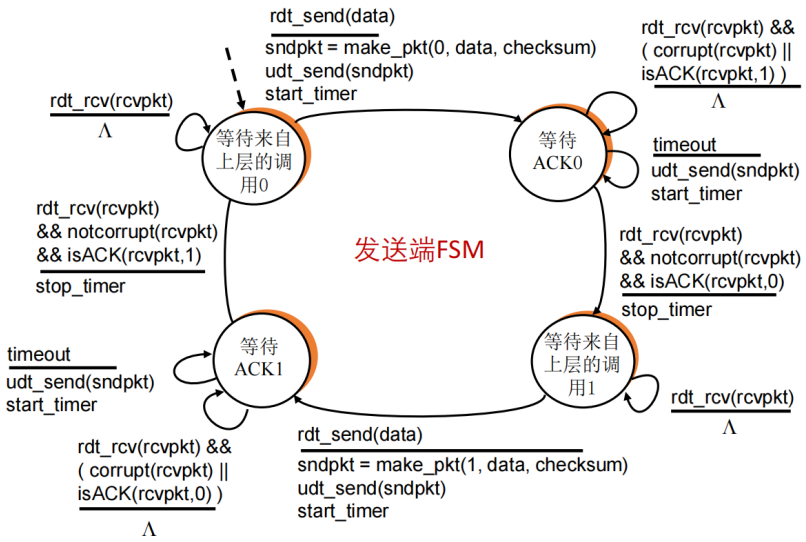
2.1 流量控制

2.1.1 rdt3.0 停等协议

在发送端实现可靠数据传输，其具体流程如下：

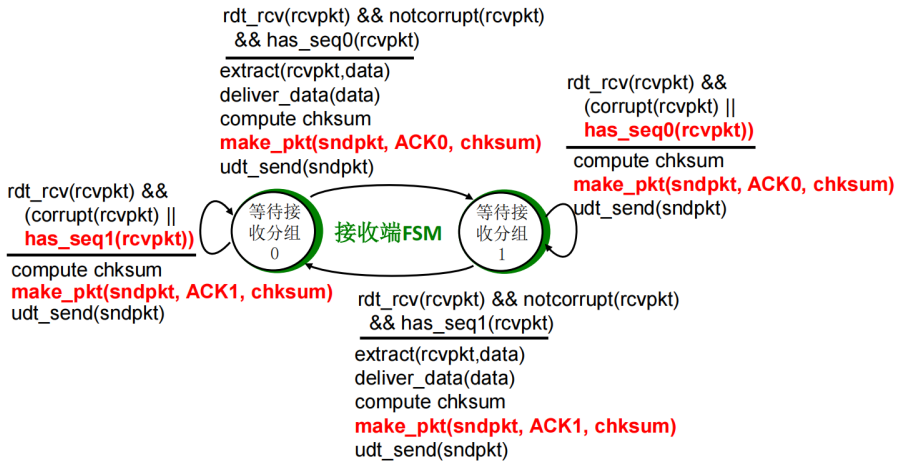
1. 初始状态的时候，发送端的序列号为 0，发送端将报文发送出去之后，就进入等待状态中，等待接收端回应的 ACK。
2. 在这个阶段中，需要开启一个超时重传计时器，根据设定的 RTO，超时未接收到 ACK 时就会重新发送数据包。
3. 在接收到接收端回应的 ACK 后，会判断 ACK 中的确认号是否和当前的序列号一致，如果一致则说明之前的数据包发送成功，可以进入下一个发送阶段，此时序列号需要 +1。
4. 如果接收到的 ACK 中确认号和当前的序列号不一致，则说明之前的数据包发送失败了，因此发送端需要重新发送数据包。
5. 进入下一状态的序列号之后，整体的流程和之前是一致的，同样是等待当前序列号的确认号，否则就触发超时重传。

■ rdt3.0：发送端状态机



在接收端实现可靠数据传输，具体流程如下：

1. 接收端初始状态的序列号为 0，等待接收 0 号数据包。在接收到数据包之后，首先要计算数据包的校验和，确保数据包在传输的过程中没有发生损坏。
2. 如果数据包的校验和不为 0，则说明在传输的过程中出现了错误，需要对当前的数据包进行重传，此时接收端回复的 ACK 中确认号为上一个序列号。
3. 而如果成功通过了校验和，要判断当前接收到的数据包中的序列号是否和自己想要确认的序列号一致，如果一致的话则说明这条报文的发送正确，确认对应的序列号。
4. 如果数据包的序列号和想要确认的序列号不一致，则同样回复上一个序列号，表示上一个这个数据包确认失败了。
5. 接收端不需要考虑超时重传的问题，因为如果接收端回复的 ACK 失败了，会触发发送端的超时重传，此时接收到的序列号和期望确认的序列号不符，而回复的确认号正好为接收到数据包的序列号。因此发送端在接收到这个 ACK 之后就会进入下一个序列号的发送状态，整个流程就恢复正常了

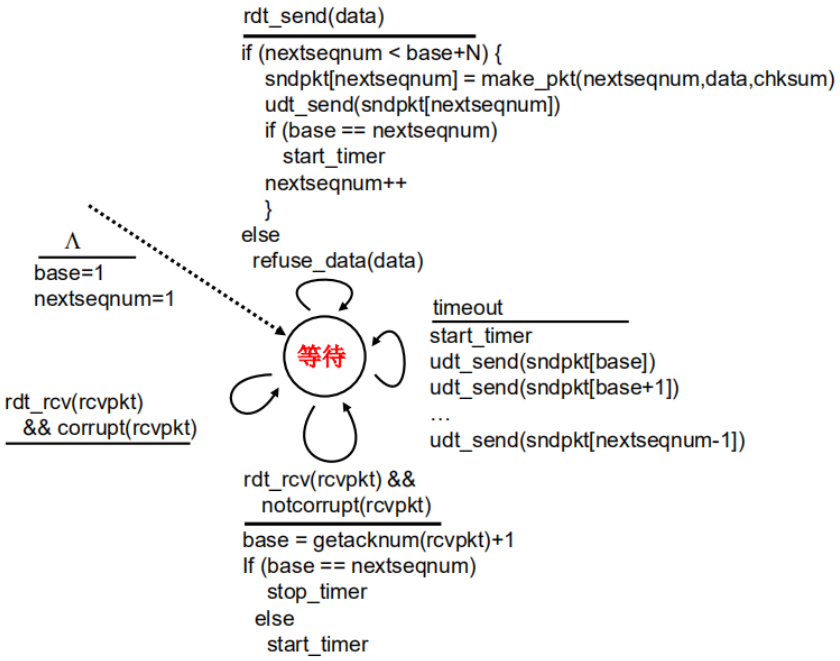


2.1.2 滑动窗口机制——后退 N 帧协议

在发送端实现 GBN 流水线协议，并且采用了固定窗口大小的发送缓冲区。通过 head 指针和 tail 指针分别来描述待确认报文序号，和当前发送过的最大报文序号。其具体流程如下：

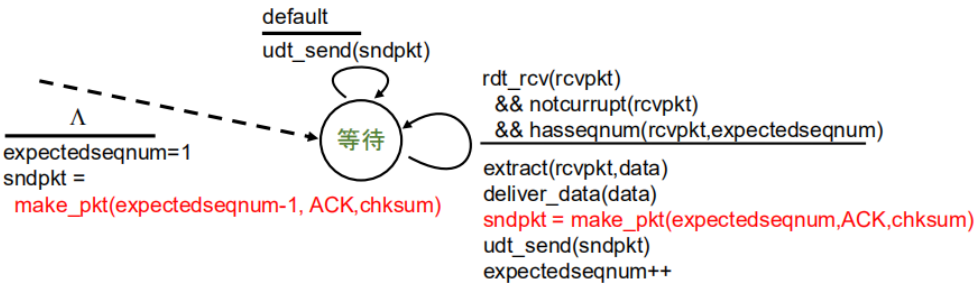
1. 发送数据的时候，首先要判断当前的发送缓冲区中是否还有空闲的位置，如果有空闲的位置，才能够进行发送。每成功发送一个新的报文之后，都要将 tail 向后进行调整，表示当前已经发送了这个报文，并等待确认中。
2. 当收到 ACK 确认报文的时候，需要判断当前确认的报文序号是否大于当前已经确认的最大报文序号，如果是，则需要将 head 指针向后移动，表示当前已经确认了这个报文。每收到一个 ACK 之后，都会重置计时器，重新开始计时，等待超时重传。
3. 当收到了一个损坏的报文的时候，就什么都不做，等待超时重传。

4. 当触发超时机制的时候，超时重传工作线程会启动，将发送缓冲区中未被确认的报文全部重新发送。



在接收端实现 GBN 流水线协议，为了简化接收端的操作，接收端并没有使用接收缓冲区。只是使用了一个 lastACK 来表示当前等待的序号，具体流程如下：

1. 当接收到一个有效的报文之后，会检查当前的报文包含的序号和自己期望等待的序号是否一致。如果一致则证明当前的报文是有效的，将其接收并传给上层应用。此时回应的确认号就是收到的报文的序号，并且将 lastACK 向后递增
2. 而如果报文无效或者序号和期待的序号不一致，则说明发生了失序，在 GBN 中需要将这个报文丢弃，并且回应确认号为 lastACK-1。
3. 累计确认机制，即接收端并不是接收到一个有效报文就回应 ACK，而是等待接收到指定 K 个有效报文之后，才会统一回应一个最大的有效报文序号。
4. 接收端不需要考虑超时重传的问题，因为如果接收端回复的 ACK 失败了，会触发发送端的超时重传，此时接收到的序列号和期望确认的序列号不符，而回复的确认号正好为接收到数据包的序列号。因此发送端在接收到这个 ACK 之后就会进入下一个序列号的发送状态，整个流程就恢复正常了。

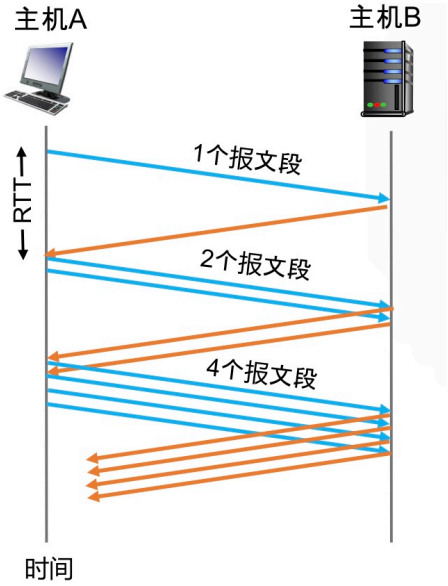


2.2 拥塞控制

2.2.1 慢启动阶段

慢启动是建立连接后，采用的第一个调整发送速率的模式。在这个阶段，cwnd 通常被初始化为 1MSS，这个值比较小，在这个时候，网络一般还有足够的富余，而慢启动的目的就是尽快找到上限。在慢启动阶段，发送方每接收到一个确认报文，就会将 cwnd 增加 1MSS 的大小，于是：

- 初始 cwnd=1MSS，所以可以发送一个最大报文段，成功确认后，cwnd = 2MSS；
- 此时可以发送两个最大报文段，成功接收后，cwnd = 4 MSS；
- 此时可以发送四个最大报文段，成功接收后，cwnd = 8 MSS.....



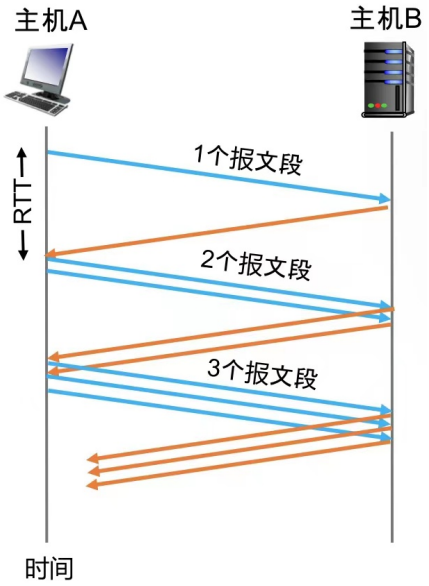
由于是一次性将窗口内的所有报文发出，所以所有报文都到达并被确认的时间，近似的等于一个 RTT。所以在这个阶段，拥塞窗口 cwnd 的长度将在每个 RTT 后翻倍，也就是发送速率将以指数级别增长。那这个过程什么时候改变呢，这又分几种情况：

- 第一种：若在慢启动的过程中，发生了数据传输超时，则此时将 ssthresh 的值设置为 cwnd / 2，然后将 cwnd 重新设置为 1MSS，重新开始慢启动过程，这个过程可以理解为试探上限；
- 第二种：第一步试探出来的上限 ssthresh 将用在此处。若 cwnd 的值增加到 >= ssthresh 时，此时若继续使用慢启动的翻倍增长方式可能有些鲁莽，所以这个时候结束慢启动，改为拥塞避免模式；
- 第三种：若发送方接收到了某个报文的三次冗余确认（即触发了快速重传的条件），则进入到快速恢复阶段；同时，ssthresh = cwnd / 2，毕竟发生快速重传也可以认为是发生拥塞导致的丢包，然后 cwnd = ssthresh + 3MSS；

以上就是慢启动的过程，下面来介绍拥塞避免。

2.2.2 拥塞避免阶段

刚进入这个模式时，cwnd 的大小近似的等于上次拥塞时的值的一半，也就是说当前的 cwnd 很接近产生拥塞的值。所以，拥塞避免是一个速率缓慢且线性增长的过程，在这个模式下，每经历一个 RTT，cwnd 的大小增加 1MSS，也就是说，假设 cwnd 包含 10 个报文的大小，则每接收到一个确认报文，cwnd 增加 1/10 MSS。



这个线性增长的过程什么时候结束，分为两种情况：

- 第一种：在这个过程中，发生了超时，则表示网络拥塞，这时候，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 1MSS，并进入慢启动阶段；
- 第二种：若发送方接收到了某个报文的三次冗余确认（即触发了快速重传的条件），此时也认为发生了拥塞，则，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 ssthresh + 3MSS，并进入快速恢复模式；

我们可以看到，慢启动和拥塞避免在接收到三个冗余的确认报文时，处理方式是一样的：判断发生了拥塞，并减小 ssthresh 的大小，但是 cwnd 的大小却不见得有减小多少，这是因为虽然发送方通过接收三次冗余确认报文，判断可能存在拥塞，但是既然可以收到冗余的确认报文，表示拥塞不会太严重，甚至已经不再拥塞，所以对 cwnd 的减小不是这么剧烈。

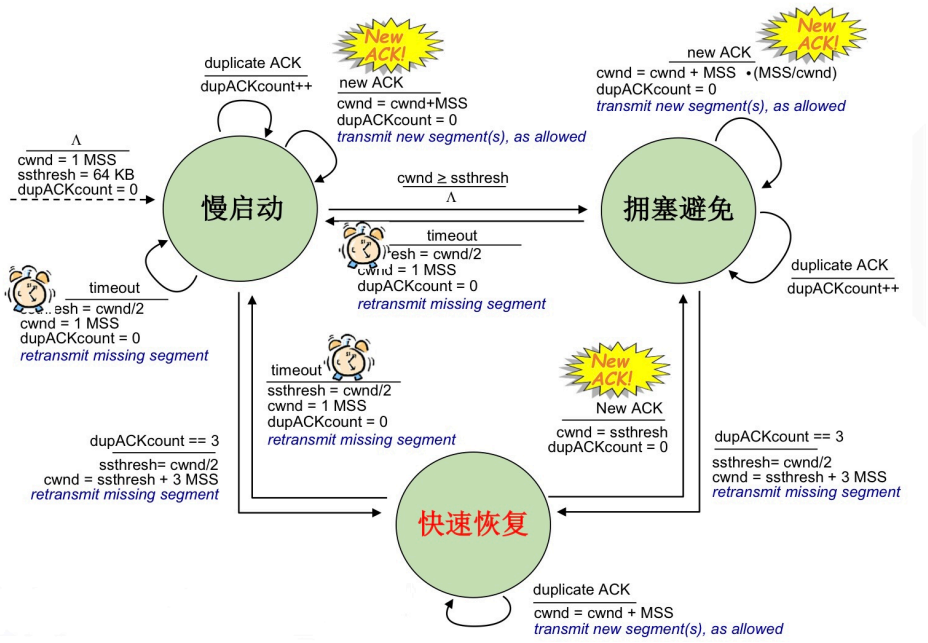
2.2.3 快速恢复阶段

在快速恢复阶段，每接收到一个冗余的确认报文，cwnd 就增加 1MSS，其余不变，而当发生以下两种情况时，将退出快速恢复模式：

- 第一种：在快速恢复过程中，计时器超时，这时候，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 1MSS，并进入慢启动阶段；
- 第二种：若发送方接收到一条新的确认报文（不是冗余确认），则 cwnd 被置为 ssthresh，然后进入到拥塞避免模式；

这里有一个疑问，进入到此模式的条件就是接收到三次冗余的确认报文，判断报文丢失，那为什么再次接收到冗余确认报文时，cwnd 还是要增长呢？此时再次收到一条冗余的确认报文，表示发送端发出的报文又有一条离开网络到达了接收端（虽然不是接收端当前想要的一条），这说明网络中腾出了一条报文的空间，所以允许发送端再向网络中发送一条报文。但是由于当前序号最小的报文丢失，导致拥塞窗口 cwnd 无法向前移动，于是只好将 cwnd 增加 1MSS，于是发送端又可以发送一条数据段，提高了网络的利用率。

2.2.4 三种模式相互转换状态图



3 实验对比流程

本次实验以测试样例中的图片 1 为例，采用控制变量法，通过分别控制延迟时间和丢包率，以传输时间和吞吐率为指标得出实验结果，做三组对比实验：

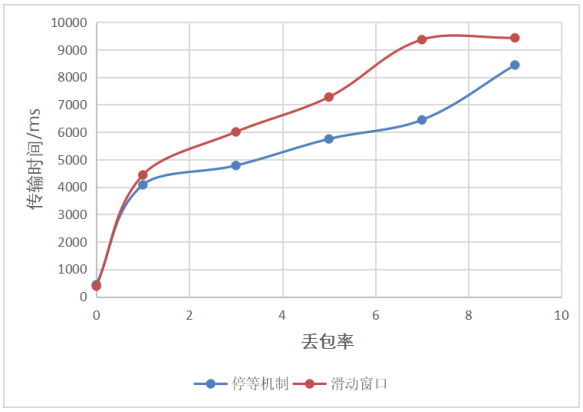
- 停等机制与滑动窗口机制性能对比
- 滑动窗口机制中不同窗口大小对性能的影响
- 有拥塞控制和无拥塞控制的性能比较

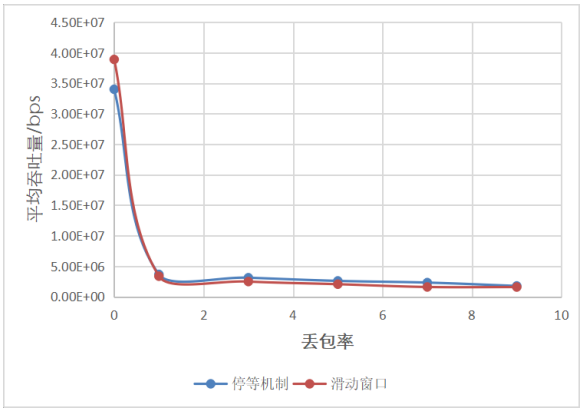
3.1 实验环境

X86 平台	Windows 系统	C++ 语言
Visual Studio 2022 集成开发环境		

3.2 停等机制与滑动窗口机制性能对比

使用路由器设置延迟时间为 0，改变丢包率分别为 0%、1%、3%、5%、7%、9%；在使用滑动窗口机制时设置窗口大小为 8，观察两种机制在不同丢包率下的传输时间与吞吐率，实验结果如下图所示：



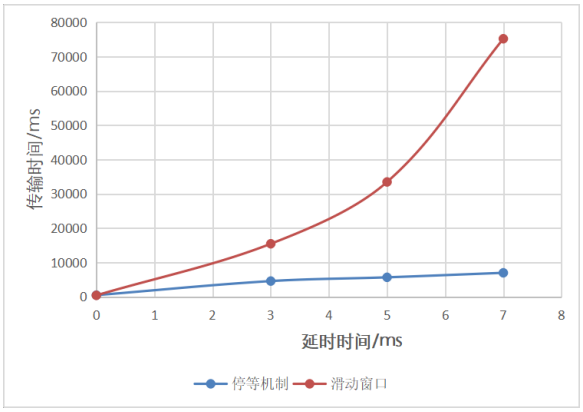


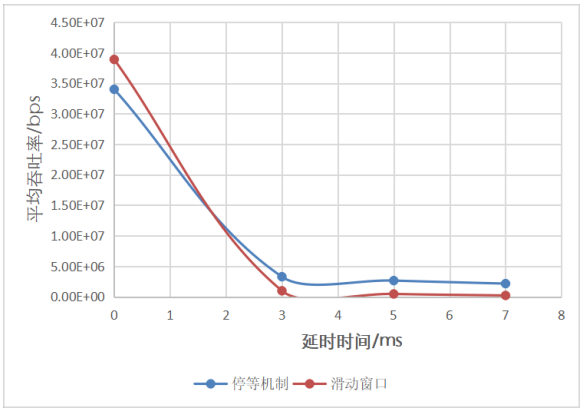
(a) 停等机制与滑动窗口机制在不同丢包率下的传输时间对比（时延 =0)(b) 停等机制与滑动窗口机制在不同丢包率下的吞吐率对比（时延 =0）

图 3.1: 停等机制与滑动窗口机制在不同丢包率下的性能对比

观察实验结果可以发现：当丢包率较小时，滑动窗口比停等机制传输时间更短，平均吞吐率更高，性能更优。随着丢包率变大，滑动窗口和停等机制效率下降，滑动窗口机制也逐渐比停等机制所需的传输时间要更长，平均吞吐率也更低，其性能降低显著。究其原因，是因为滑动窗口机制采用流水线的模式，可以连续发送多个报文，不需要像停等机制一样等待上一个报文的 ACK 收到后才能传输下一个报文，所以在丢包率较小时，这大大提高了数据的传输速率，使得其性能更高。但随着丢包率逐渐增大，如果采用滑动窗口机制，当服务器端没有收到预期的数据报文，会一直回复同一个 ACK，导致发送端重发窗口中全部内容，使代价大大提高，严重影响数据传输效率。

使用路由器设置丢包率为 0，改变延迟时间分别为 0ms、3ms、5ms、7ms，设置超时时间为 100ms；在使用滑动窗口机制时设置窗口大小为 8。观察两种机制在不同时延下的传输时间与吞吐率，实验结果如下图所示：





(a) 停等机制与滑动窗口机制在不同时延下的传输时间对比（丢包率 =0)(b) 停等机制与滑动窗口机制在不同时延下的吞吐率对比（丢包率 =0）

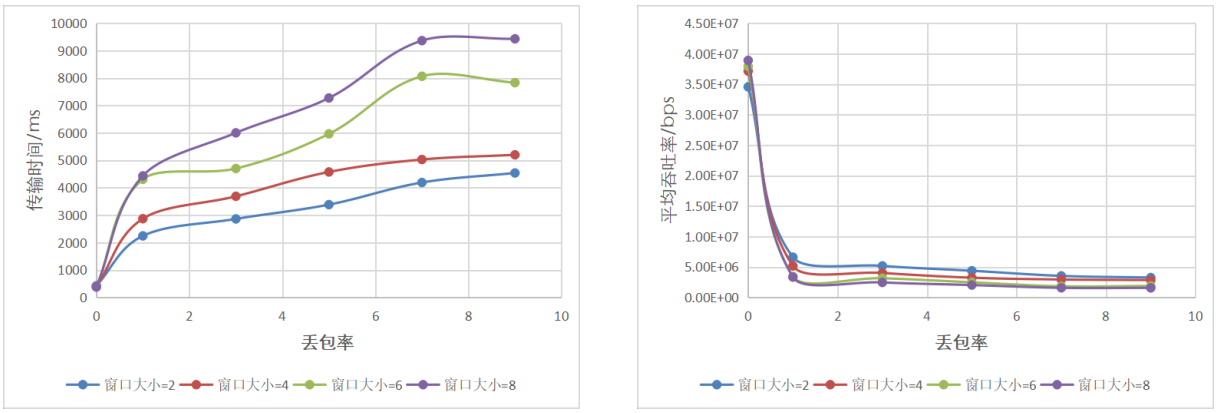
图 3.2: 停等机制与滑动窗口机制在不同时延下的性能对比

观察实验结果可以发现：当时延较小时，滑动窗口比停等机制传输时间更短，平均吞吐率更高，性能更优。随着时延增大，滑动窗口和停等机制效率下降，但滑动窗口机制下降更快。究其原因，是因为滑动窗口机制采用流水线的模式，可以连续发送多个报文，不需要像停等机制一样等待上一个报文的 ACK 收到后才能传输下一个报文，所以在时延较小时，这大大提高了数据的传输速率，使得其性能更

高。但时延对滑动窗口机制影响较大，时延较大时，窗口内报文重发的几率更大，导致效率下降快。

3.3 滑动窗口机制中不同窗口大小对性能的影响

使用路由器设置延迟时间为 0，改变丢包率分别为 0%、1%、3%、5%、7%、9%。观察滑动窗口大小分别为 2、4、6、8 时，在不同丢包率下的传输时间与吞吐率，实验结果如下图所示：

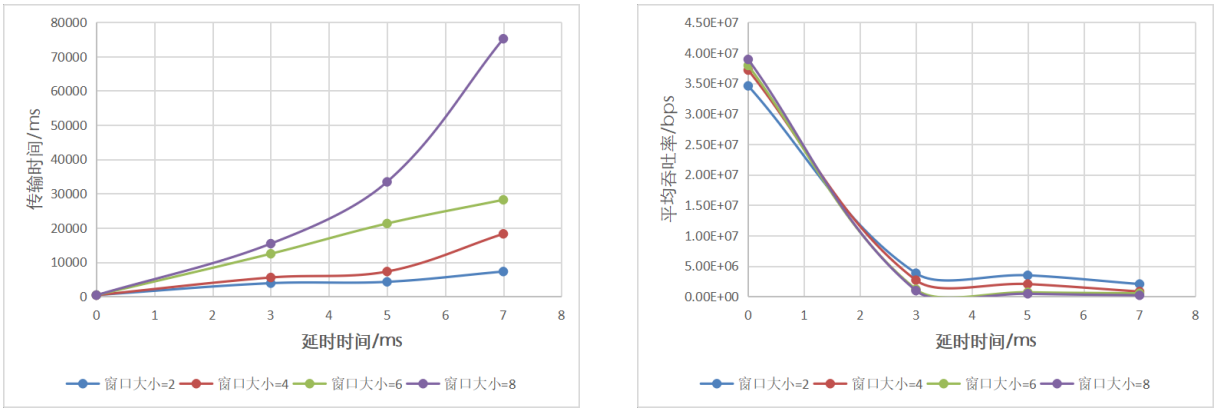


(a) 滑动窗口机制中不同窗口大小在不同丢包率下的传输时间对比（时延 =0）(b) 停等机制与滑动窗口机制在不同丢包率下的吞吐率对比（时延 =0）

图 3.3: 滑动窗口机制中不同窗口大小在不同丢包率下的性能对比

观察实验结果可以发现：当丢包率较小时，滑动窗口越大，传输时间越短，平均吞吐率越高，性能越优。随着丢包率变大，不论窗口多大，效率都会下降，但窗口越大下降幅度越大。究其原因，是因为滑动窗口越大，允许连续发送报文个数越多，所以在丢包率较小时，窗口越大效率越高。当丢包率变大时，由于丢包过多，接收方接不到报文，一直回复同一个 ack 报文，窗口无法滑动，只能等到超时重传。此时窗口越大，重发窗口内报文越多，所以效率会变低。

使用路由器设置丢包率为 0，改变延迟时间分别为 0ms、3ms、5ms、7ms，设置超时时间为 100ms。观察滑动窗口大小分别为 2、4、6、8 时，在不同时延下的传输时间与吞吐率，实验结果如下图所示：



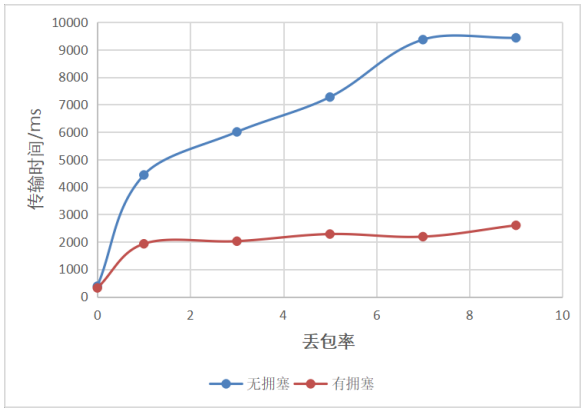
(a) 滑动窗口机制中不同窗口大小在不同时延下的传输时间对比（丢包率 =0）(b) 停等机制与滑动窗口机制在不同时延下的吞吐率对比（丢包率 =0）

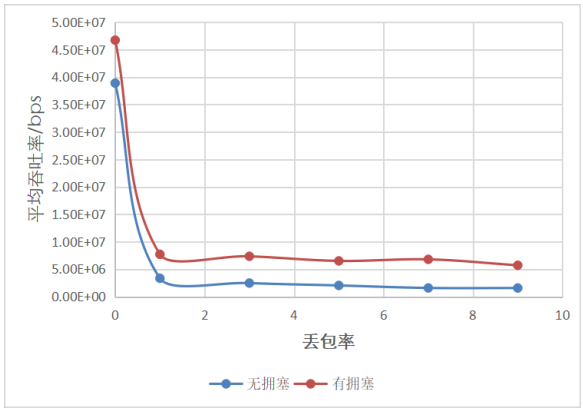
图 3.4: 滑动窗口机制中不同窗口大小在不同时延下的性能对比

观察实验结果可以发现：时延较小时，滑动窗口越大，传输时间越短，平均吞吐率越高，性能越优。但随着时延增多，不论窗口多大，效率都会降低，而窗口越大的效率降低幅度越大。究其原因，是因为当时延变大时，由于时延过大，接收方接到报文时间长，可能导致超时重发。一旦超时，窗口内所有报文都要重发，此时窗口越大，重发窗口内报文越多，所以效率会变低。

3.4 有拥塞控制和无拥塞控制的性能比较

使用路由器设置延迟时间为 0，改变丢包率分别为 0%、1%、3%、5%、7%、9%。观察有拥塞控制和无拥塞控制在不同丢包率下的传输时间与吞吐率，实验结果如下图所示：



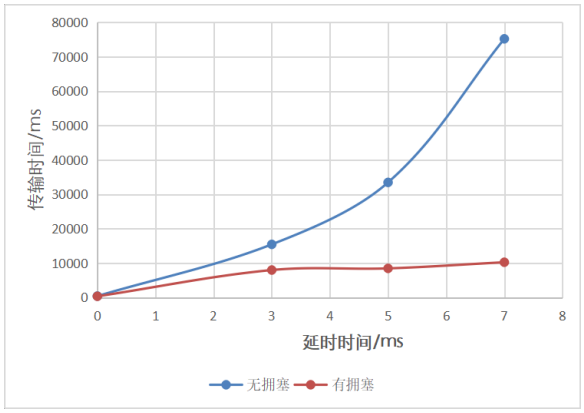


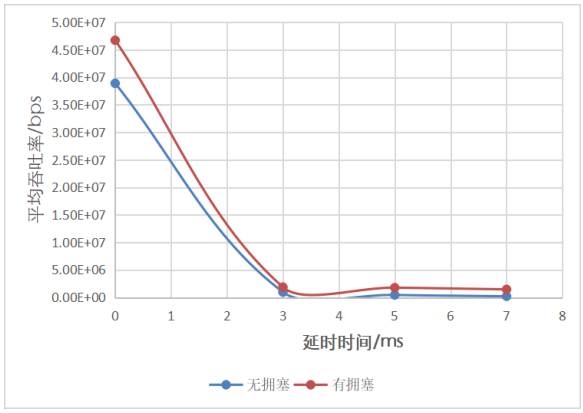
(a) 有拥塞控制和无拥塞控制在不同丢包率下的传输时间对比（时延 =0)(b) 有拥塞控制和无拥塞控制在不同丢包率下的吞吐率对比（时延 =0）

图 3.5: 有拥塞控制和无拥塞控制在不同丢包率下的性能对比

观察实验结果可以发现：随着丢包率增大，有拥塞控制与无拥塞控制性能都降低，但有拥塞控制的性能始终优于无拥塞控制的情况。因为无拥塞控制的只能重发窗口内的报文，而有拥塞控制能够随时改变窗口大小，当丢包率小时，它可以调整窗口大小大于无拥塞控制中的固定窗口大小，提高传输速率；在丢包率大的时候，降低窗口大小，有利于后续重发报文的数量降低，从而提高传输性能。

使用路由器设置丢包率为 0，改变延迟时间分别为 0ms、3ms、5ms、7ms，设置超时时间为 100ms。观察有拥塞控制和无拥塞控制在不同时延下的传输时间与吞吐率，实验结果如下图所示：





(a) 有拥塞控制和无拥塞控制在不同延时下的传输时间对比（丢包率 =0)(b) 有拥塞控制和无拥塞控制在不同延时下的吞吐率对比（丢包率 =0）

图 3.6: 有拥塞控制和无拥塞控制在不同延时下的性能对比

观察实验结果可以发现：随着时延增大，有拥塞控制与无拥塞控制性能都降低，但有拥塞控制的性能始终优于无拥塞控制的情况。因为无拥塞控制的只能重发窗口内的报文，而有拥塞控制能够随时改变窗口大小，当时延小时，它可以调整窗口大小大于无拥塞控制中的固定窗口大小，提高传输速率；在时延较大时，降低窗口大小，有利于后续重发报文的数量降低，从而提高传输性能。