

**《计算机网络实践》课程报告**



**基于UDP的TCP协议设计与实现**

**学 号 3022206045 3022232100**

**姓 名 陆子毅 李国鸿**

**学 院 智能与计算学部**

**专 业 计算机科学与技术**

**年 级 2022级**

**任课教师 石高涛**

**2024年08月28日**

# 一、报告摘要

基于系统已有的UDP协议和接口，实现自定义TCP协议。协议设计主要包含连接管理、可靠数据传输、流量控制和拥塞管理四个部分。实验得到一个功能较为完善的TCP协议，可以用构建的TCP接口编写应用程序。

# 二、任务要求

## **2.1 连接管理**

需要实现TCP连接中建立连接的三次握手和TCP关闭连接时的四次挥手。根据TCP的状态转换图正确实现TCP连接建立过程，和TCP连接的关闭过程。

## **2.2 可靠数据传输**

TCP是在IP层提供的不可靠、尽力而为服务基础上建立的一种可靠数据传输服务。实验中需要实现超时重传，报文序列管理等功能。达到简易的可靠数据传输功能。

## **2.3 流量控制**

TCP的流量控制是指在已经建立好连接的双方之间，为了避免发送速率过快和过慢锁带来的过高丢包率，高延迟，网络拥塞和网络链路利用率不高的情况。主要利用滑动收发窗口来实现。

## **2.4 拥塞管理**

TCP的拥塞控制下，连接有四种状态，分别是慢启动，拥塞避免，快速重传和快速回复。

根据TCP的RFC和实践要求，详细描述实践任务需要解决的具体问题。主要解决在陌生网络环境中提高链路利用率，使TCP能够动态调节网络性能。

# 三、协议设计

**注意：协议设计的内容要涵盖“计算机网络课程实践说明书”中“二、TCP功能需求”所要求的所有内容。这是评分的依据。**

下面给出主要的章节。

## **3.1 总体设计**

分为四个板块，分别是，连接管理模块，可靠传输模块，流量控制模块，拥塞管理模块。

## **3.2 连接建立的设计**

TCP在建立连接过程中采用三次握手的方式，参考RFC 793文档中TCP建立连接的描述。“三次握手”是用于建立连接的过程。此过程通常由一个 TCP 启动，并由另一个 TCP 响应。如果两个 TCP 同时启动该过程，则该过程也有效。有丢包和延迟的情况下建立连接的功能，将会在可靠数据传输部分实现。

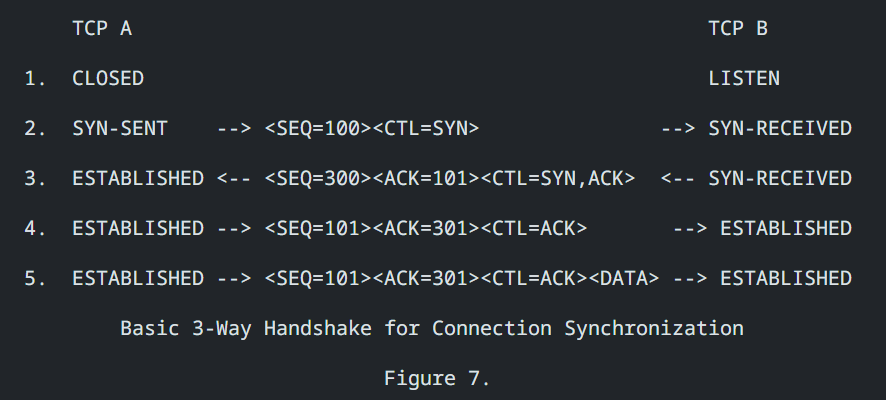


图3-2-1 RFC 793对于“三次握手”的图解

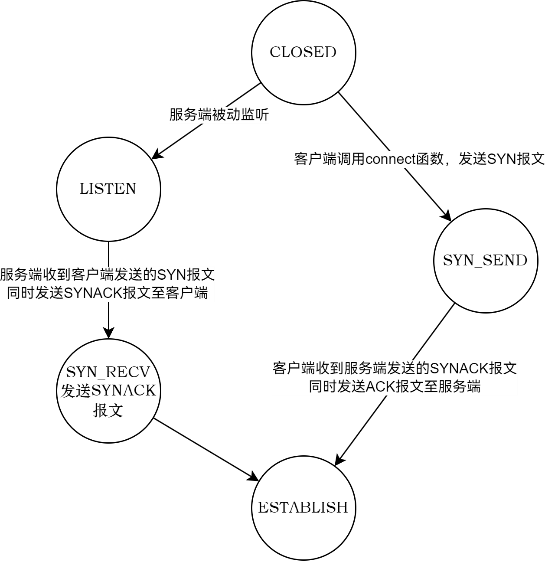


图3-2-2 三次握手的FSM图解

控制三次握手最主要的数据就是客户端和服务端的状态（state），需要根据客户端和服务端收到的不同的报文flag来进行不同的状态转换。同时在建立连接的过程中需要维护一个全连接队列，将已经完成三次握手的sock放入队列中，等待tju\_accept()函数的调用。建立完成时，需要服务端新建一个sock来建立与客户端的连接而不是直接使用listen sock作为建立的连接进行通信。

根据设计的“三次握手”协议，双方在建立连接的通信过程如下：

1. client调用tju\_connect函数，向server发送SYN报文请求连接。client状态转变为SYN\_SENT。server收到client发送的SYN报文，将server的连接状态转变为SYN\_RECV。
2. server发送SYNACK报文给client，client收到server发送的SYN\_ACK报文之后，将连接状态转变为ESTABLISH。
3. client向server发送ACK报文，确认连接建立，server收到ACK报文之后，新建一个sock并将连接状态变为ESTABLISH，并将新建立的sock放入全连接队列中

## **3.3 可靠数据传输的设计**

当发送方数据发送速度过快，接收方无法及时处理时，可能会导致数据包丢失。流量控制的目的就是调节发送方的发送速度，确保接收方能够跟上接收进度，避免丢包。

接收方在返回的确认消息（ACK）中包含了自己的接收窗口大小。发送方会依据这一窗口大小来调整其发送窗口，确保发送数据量不超过接收方的接收能力。此外，如果发送方收到接收窗口大小为 0 的消息，它将停止数据传输，防止发生大量数据包丢失的情况。

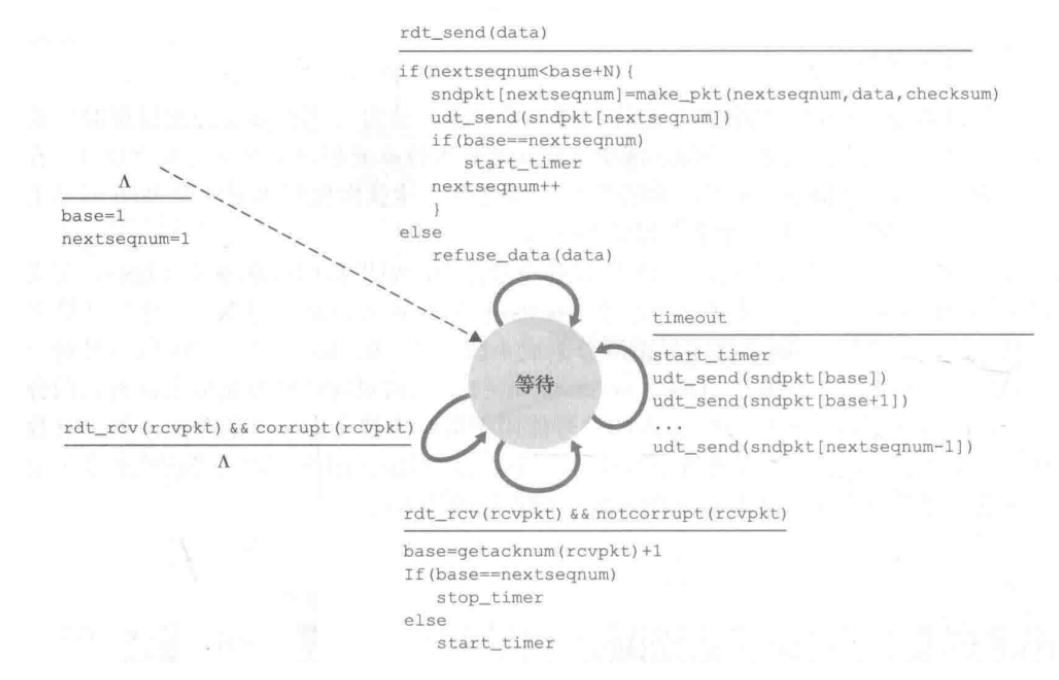


图3-3-1 RDT的FSM图解

校验

1. 序号机制

TCP 协议通过序号字段确保数据能够有序地传递给应用程序。在 TCP 的视角中，数据被看作是一系列有序但无结构的字节，每个字节都被赋予一个序号。序号字段在 TCP 报文段中指明了该段数据的第一个字节的序号。发送方使用 nextseq 来追踪下一个即将发送的序列号，而 base 则追踪最老的未被确认的序列号。接收方则用 expect\_seq 来追踪它期望接收的下一个字节的序列号。 当数据传输处于空闲状态且所有数据都已被确认时，这三个序列号是相等的。每当发送方发送一个报文段，nextseq 就会增加；接收方在接收到报文段后，会增加 expect\_seq 并发送确认；发送方在收到确认后，会更新 base。这三个序列号之间的差异反映了网络通信的延迟，而它们的增量则依赖于报文段中数据的长度。

1. 确认机制

TCP 协议中的确认号指的是接收方期望接收的下一个字节的序列号。TCP 默认采用累积确认机制，这意味着接收方仅确认它已经接收到的第一个丢失字节之前的所有字节。

**重传机制**

1. 重传定时器

每个 TCP 报文段都关联有一个定时器，但为了简化管理，通常使用一个统一的重传定时器来监控多个未被确认的报文段。 发送方定时器：在发送方，主要涉及三个与发送和重传相关的事件：接收到来自应用程序的数据（导致发送队列从空变为非空）、定时器超时以及在收到 ACK 后调整发送窗口。接收方定时器：通常只在 TCP 三次握手阶段被启用。

1. 超时处理

TCP 为每个发送的报文段设置一个计时器。如果在计时器超时之前未收到对应的确认，该报文段将被重传。当接收方检测到数据错误时，它会直接丢弃错误的数据，而发送方则会在超时后重传相应的报文段。

**RTT 和 RTO 的计算：**

通过测量从发送某一序列号的数据包到接收方发送确认之间的时间，可以获取往返时间（RTT）。

RTT 的估计值可以通过以下公式计算：estimated\_rtt=(1−α)×estimated\_rtt+α×sample\_rttestimated\_rtt=(1−α)×estimated\_rtt+α×sample\_rtt，其中 α 通常设为 0.125。

计算 RTT 的偏差：deviation\_rtt=(1−β)×deviation\_rtt+β×∣sampleRTT−estimated\_rtt∣deviation\_rtt=(1−β)×deviation\_rtt+β×∣sampleRTT−estimated\_rtt∣，β 通常设为 0.25。

根据估计的 RTT 和偏差来计算重传超时（RTO）：RTO=estimated\_rtt+4×deviation\_rttRTO=estimated\_rtt+4×deviation\_rtt。 重传超时间隔：每次重传后，TCP 会将下一次的超时时间设为上一次的两倍。

冗余 ACK 在超时发生之前，如果发送方连续收到三个相同的确认（ACK），它会推测可能发生了丢包，并会提前重传相应的报文段，而不需要等待定时器超时。接收方在接收到数据分组后，会稍作等待以积累确认，当检测到丢包时，它会发送期望的报文段和乱序数据包边界的确认。

## **3.4 流量控制的设计**

**滑动窗口机制原理**

滑动窗口协议是一种允许发送方在未收到确认的情况下连续发送多个数据包的技术，这有助于提高数据传输的效率和网络吞吐量，实现 TCP 的并行数据传输。滑动窗口的大小反映了接收方 TCP 缓冲区的容量，发送方根据接收方通告的窗口大小来确定可以发送的数据量。当接收方的窗口大小为零时，发送方会暂停发送数据，直到接收方通知有可用的窗口空间。

流量控制通过接收方通告其窗口大小给发送方来实现，确保发送方不会发送超出接收方处理能力的数据量。接收方在 ACK 包中包含其窗口大小信息，发送方据此调整其发送速率。

**窗口结构细节**

发送方的滑动窗口由以下四个部分组成：

已发送且已确认的字节：这部分数据已经被发送方发送出去，并且接收方已经确认收到。

已发送但未确认的字节：发送方已经发送这部分数据，但还未收到接收方的确认。

未发送但可发送的字节（可用窗口）：这部分数据接收方已经准备好接收，发送方可以立即发送。

未发送且接收方尚未准备接收的字节：这部分数据当前接收方无法接收。

发送窗口主要由第二和第三部分构成，表示发送方可以发送或已经发送但尚未确认的数据。

接收端的滑动窗口由以下三个部分组成：

已接收并发送了 ACK 的数据：这部分数据已经被接收方接收并确认。

接收窗口（RWND）：这部分数据是接收方允许发送方发送但尚未实际接收的。

尚未接收且不允许发送的数据：这部分数据是接收方当前无法接收的。

接收窗口的大小（RWND）是接收方通告给发送方的关键信息，它决定了发送方可以发送多少数据而不会导致接收方的缓冲区溢出。

**零窗口探测**

当接收方缓存不足时，会发送零窗口大小的报文，通知发送方停止发送数据。若之后接收方有足够缓存，但其发送的非零窗口报文丢失，发送方将一直认为接收窗口为 0，导致死锁。为解决此问题，TCP 引入了零窗口探测定时器。当发送方收到零窗口通知后，定时器启动，定时发送 1 字节的探测报文，接收方回应其最新的接收窗口大小。如果仍为 0，则重设计时器，继续等待。首次超时设为 1.5 秒，之后每次超时时间加倍，总时长控制在 5 到 60 秒之间。

## **3.5 连接关闭的设计**

TCP连接的关闭采用“四次挥手”协议。TCP 是全双工通信，可以双向传输数据。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了 TCP 连接。

（1）第一次挥手：客户端发送一个 FIN（SEQ=x） 标志的数据包->服务端，用来关闭客户端到服务端的数据传送。然后客户端进入 FIN-WAIT-1 状态。

（2）第二次挥手：服务端收到这个 FIN（SEQ=X） 标志的数据包，它发送一个 ACK （ACK=x+1）标志的数据包->客户端 。然后服务端进入 CLOSE-WAIT 状态，客户端进入 FIN-WAIT-2 状态。

（3）第三次挥手：服务端发送一个 FIN (SEQ=y)标志的数据包->客户端，请求关闭连接，然后服务端进入 LAST-ACK 状态。

（4）第四次挥手：客户端发送 ACK (ACK=y+1)标志的数据包->服务端，然后客户端进入TIME-WAIT状态，服务端在收到 ACK (ACK=y+1)标志的数据包后进入 CLOSE 状态。此时如果客户端等待 2MSL 后依然没有收到回复，就证明服务端已正常关闭，随后客户端也可以关闭连接了。

## **3.6拥塞控制的设计**

为了控制网络中的拥塞，发送方会使用一个名为拥塞窗口（cwnd，congestion window）的状态变量来调整数据的发送速率。拥塞窗口的大小直接影响到发送方可以发送到网络中的数据量。

拥塞控制的过程通常涉及三个主要算法：慢启动（slow start）、拥塞避免（congestion avoidance）和快速恢复（fast recovery）。下面是一个简化的拥塞控制算法流程图：

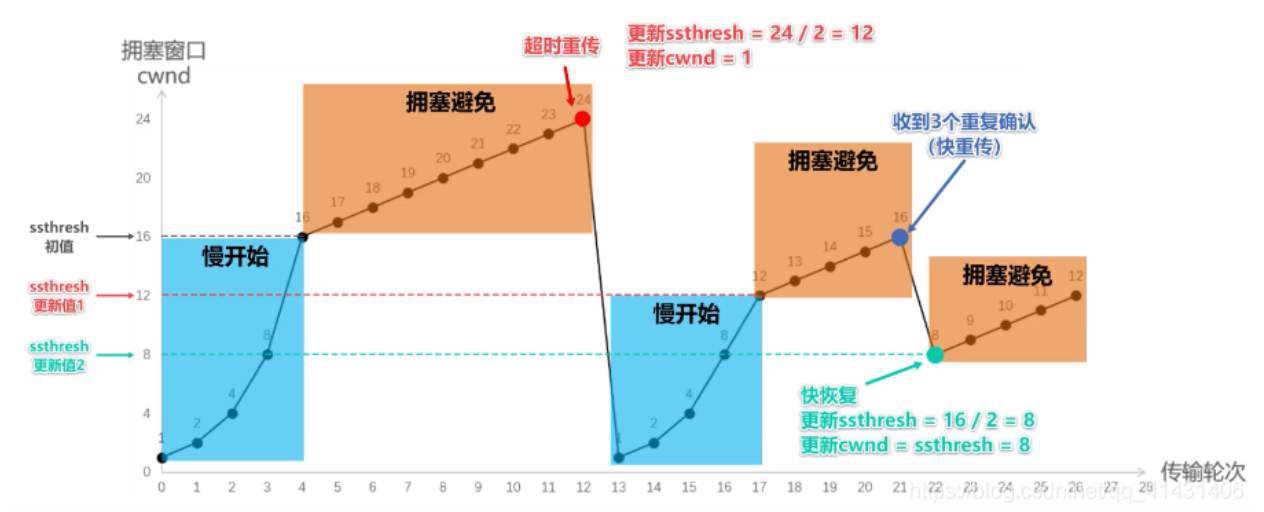


图 3-6-1 拥塞窗口随传输轮次变化

定义两个新的变量来管理连接状态的变动：cwnd（拥塞窗口）和ssthresh（慢启动阈值）。cwnd限制了发送方在接收到确认信号（ACK）前能够发送到网络中的数据量，其初始值设定为3倍的最大报文段大小（MSS）；ssthresh则用于判定是采用慢启动还是拥塞避免算法来调节数据流，其初始值应设置为通告窗口大小（Advertised Window）。发送窗口的尺寸（swnd）应取cwnd和ssthresh两者中的较小值。

在TCP协议中，SYN和SYN/ACK包的确认不会增加拥塞窗口（cwnd）的大小。此外，如果SYN或SYN/ACK包在传输过程中丢失，那么在成功传输SYN和SYN/ACK包之后，发送方使用的初始窗口大小不应超过单个最大报文段（MSS）的大小。

**3.6.1 慢启动**

慢启动阶段通常在TCP连接建立之初或在检测到数据丢失并重置传输计时器之后发生。在这个阶段，TCP通过逐步增加发送到网络的数据量来探测网络容量，目的是防止突然大量数据传输导致网络拥塞。

在慢启动期间，每当发送端收到接收端的一个确认（ACK），其拥塞窗口（cwnd）就会增加一个最大报文段大小（MSS）。这种机制使得cwnd以指数级增长，从而快速地增加数据传输量，直到达到慢启动阈值（ssthresh）或发生数据丢失。

慢启动状态的转变可以通过以下三种情况来描述：

**超时导致的慢启动重启：**

当TCP检测到超时（即重传计时器到期），它会将拥塞窗口（cwnd）重置为1个最大报文段大小（MSS），并重新开始慢启动过程。同时，慢启动阈值（ssthresh）会被设置为当前拥塞窗口大小（cwnd）的一半。

**达到慢启动阈值：**

当拥塞窗口（cwnd）增长至慢启动阈值（ssthresh）时，TCP将停止使用慢启动算法，转而进入拥塞避免状态。在拥塞避免状态下，cwnd的增长速度会减缓，通常每收到一个ACK，cwnd增加1/MSS。

**快速重传和快速恢复：**

当TCP检测到3个冗余的ACK（即接收到重复的ACK，表明有数据包丢失），它会执行快速重传机制，立即重传丢失的数据包，而不是等待重传计时器超时。同时，TCP进入快速恢复状态，将ssthresh设置为cwnd的一半，并将cwnd设置为新的ssthresh值加上3倍MSS（因为收到了3个冗余ACK）。在快速恢复状态下，每当收到一个重复ACK，cwnd就增加1/MSS，直到收到新的ACK为止，之后TCP会回到拥塞避免状态。

**3.6.2 拥塞避免**

拥塞避免算法的设计目的是在TCP连接中控制数据传输的增长速度，以防止网络拥塞。在拥塞避免阶段，cwnd（拥塞窗口）的增长方式与慢启动阶段不同，它采取的是线性增长策略，而不是指数增长。具体来说：

拥塞避免阶段的cwnd增长：

在拥塞避免阶段，每当一个往返时间（RTT）内收到一个新的确认（ACK），cwnd就会增加1个MSS（最大报文段大小）。这种增长方式比慢启动阶段的指数增长要慢，但更加稳定。

**拥塞避免算法与慢启动状态的转变：**

当TCP在拥塞避免阶段检测到拥塞（例如，通过超时或收到3个冗余ACK），它会采取与慢启动状态相同的措施：

超时：如果发生超时，TCP将cwnd重置为1个MSS，并重新开始慢启动过程。同时，将ssthresh设置为cwnd的一半。

快速重传和快速恢复：如果检测到3个冗余ACK，TCP会执行快速重传，立即重传丢失的数据包，并进入快速恢复状态。在快速恢复状态下，cwnd的增长方式会有所调整，以快速恢复丢失的数据并逐渐增加cwnd。

**超时后的慢启动状态：**

当发生超时事件时，TCP会回到慢启动状态，重新开始慢启动过程。这是因为超时通常意味着网络中可能存在拥塞，需要TCP重新评估网络的容量。

拥塞避免算法通过这种线性增长方式，试图在不引起网络拥塞的同时，最大化网络的吞吐量。这种策略有助于TCP在保持网络稳定性的同时，逐步增加数据传输量。

**3.6.3 快速恢复**

快速恢复是TCP丢失恢复机制的一种优化。在进入快速恢复状态时，会将慢启动阈值（ssthresh）设置为当前拥塞窗口（cwnd）的一半，并将拥塞窗口（cwnd）设置为慢启动阈值加上3倍的最大报文段大小（3MSS）。对于每一个额外收到的冗余ACK，cwnd会增加1个MSS。

状态转换规则如下：

当收到一个丢失报文段的ACK时，TCP将cwnd设置为ssthresh，并切换到拥塞避免状态。

如果发生超时，TCP会采取与慢启动和拥塞避免相同的措施，将cwnd重置为1个MSS，并进入慢启动状态。

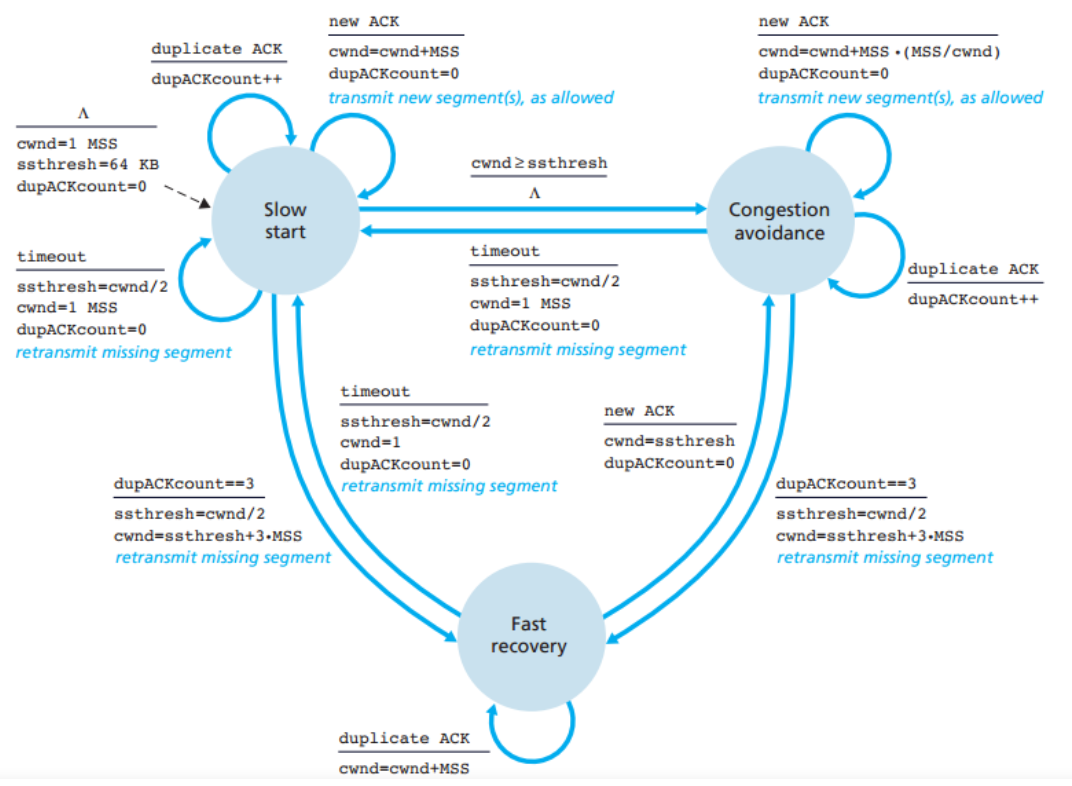


图3-6-2 拥塞控制状态转换图

# 四、协议实现

详细描述功能实现的细节。主要功能模块使用流程图或者伪代码来辅助说明。禁止贴源码。

**注意：协议实现的内容要和“协议设计”部分相对应。每项功能设计都要有相应的实现。**

## **4.1 连接建立的实现**

（一）全连接队列

使用数据结构实现了全连接队列，用于存储已经可以用于通信的连接。具体实现参照C++数据结构队列的实现。在tju\_socket函数中完成队列的初始化。

在tju\_handle\_packet中，在server成功建立连接之后，将新建立的连接放入队列中等待tju\_accept函数调用取出。

同时tju\_accept函数中添加了阻塞，当队列中没有新建立的连接的时候会陷入等待，直到连接建立完成，返回可以直接用于通信的sock。

（二）三次握手建立连接

主要在tju\_handle\_packet函数和tju\_connect函数中实现。

首先，在tju\_connect函数中，当函数被调用，就向服务端发送一个SYN报文，请求建立连接。建立连接过程中，需要提前建立sock，并写入established\_socks哈希表中，避免之后因不存在能够处理服务端发送的SYNACK报文的sock而将包丢弃。随机进入阻塞状态直到sock的状态变为ESTABLISH。

然后，需要修改tju\_handle\_packet函数，该函数用于处理收到的报文。通过预值的提取函数，将报文的flag，seq，ack，src，des等字段提取出来，然后根据sock目前的状态，结合tcp连接的状态转换图进行相应的报文发送。

下面是tju\_handle\_packet函数的伪代码：

根据 sock 的状态进行处理

如果状态是 LISTEN

如果 flag 是 SYN\_FLAG\_MASK

更新状态为 SYN\_RECV

创建并发送 SYN-ACK 包

如果状态是 SYN\_SENT

如果 flag 是 SYN\_FLAG\_MASK | ACK\_FLAG\_MASK

创建并发送 ACK 包

更新状态为 ESTABLISHED

如果状态是 SYN\_RECV

如果 flag 是 ACK\_FLAG\_MASK

分配并初始化新连接 new\_conn

设置新连接的本地和远程地址

更新新连接状态为 ESTABLISHED

计算哈希值并存储新连接

将新连接加入接受队列

更新 sock 状态为 LISTEN

## **4.2 可靠传输的实现**

**重传功能实现**

在 TCP 协议的三次握手过程之后，重传机制是确保数据完整性的关键。当系统不在三次握手状态（即不处于 SYN\_SENT 或 SYN\_RECV）时，首先检查 sended\_pointer 是否为零。如果为零，表示没有需要重传的数据包；如果非零，则需要通过循环重新发送那些已经发送但未收到确认（ACK）的数据包，范围从 base 到 nextseq。在此过程中，首先更新接收窗口 advertised\_window，其计算方式为 sock->received\_capacity - sock->received\_len。然后，使用 sprintf() 将 sended\_queue[i] 格式化为字符型，并调用 sendToLayer3() 发送数据包，最后通过 trace\_write() 记录日志信息。

**发送端定时器实现**

在客户端建立连接时（tju\_connect()），首次启动定时器。设置定时器状态为开启，调用 getCurrentTime() 获取当前时间并存储在 start\_time 中。使用 pthread\_create() 创建定时器线程，并检查定时器是否成功建立。在发送数据前，检查发送缓冲区长度 sending\_len。如果 sending\_len 小于等于零，则通过循环等待；如果 sending\_len 大于零且未超时，则发送数据包并检查定时器状态。如果定时器关闭，则重新启动并更新 start\_time。定时器线程定期检查状态，如果开启，则计算当前时间与 start\_time 的差值。如果超过 timeout\_interval，则更新 start\_time，将 timeout\_interval 翻倍，并调用 resent 函数进行重传。收到 ACK 包时，关闭定时器并移动发送窗口。如果 base 小于 nextseq，则重新启动定时器。

**RTO 计算实现**

通过 handle\_RTO 函数实现 RTO 计算。sampleRTT 计算为当前时间减去发送时间 send\_time。如果 sampleRTT 小于 6000 毫秒，则直接返回。如果 timeout\_interval 是 15000 毫秒的倍数，则将 estmated\_rtt 设置为 sampleRTT，并将 deviation\_rtt 设为 sampleRTT/2。否则，使用公式 deviation\_rtt = 3/4 \* deviation\_rtt + |sampleRTT - estmated\_rtt| / 4 更新 deviation\_rtt，并将 estmated\_rtt 更新为 7/8 \* estmated\_rtt + 1/8 \* sampleRTT。然后计算新的 timeout\_interval。如果 4 \* deviation\_rtt 大于 1000 毫秒，则 timeout\_interval = estmated\_rtt + 4 \* deviation\_rtt，否则 timeout\_interval = estmated\_rtt + 1000。RTO 的上限设为 120 秒，下限为 1 秒，并记录 RTT 日志。

## **4.3 流量控制的实现**

**窗口管理**

在发送线程中，首先判断发送数据缓存长度。如果大于零，则更新发送窗口。检查 rwnd（接收窗口）是否为零。如果不为零，则加锁以避免线程冲突。获取以下参数：

发送窗口大小 window\_size：sock->window.wnd\_send->window\_size

可用窗口大小 window\_left：window\_size + base - nextseq

实际可用窗口大小 max\_dlen：min(window\_left, rwnd)

未发送完的数据大小 data\_left：sock->sending\_len - max\_dlen

确定本次发送的数据长度 dlen，如果 data\_left 大于零，则 dlen 取 max\_dlen，否则取 sock->sending\_len。然后创建缓冲区，复制数据到 sending\_buf 中，更新 sending\_len 和 sending\_capacity，并移动缓冲区内的数据。最后解锁。

数据包发送流程，获取以下参数：

seq：sock->window.wnd\_send->nextseq

ack：sock->window.wnd\_recv->expect\_seq

plen：DEFAULT\_HEADER\_LEN + dlen

更新 wnd\_send->nextseq 表示已发送但未确认的数据。如果 seq 小于 base + window\_size，则可以发送。更新接收窗口 advertised\_window，创建数据包并放入 sended\_queue 队列中，更新 sended\_pointer 并自增。记录发送时间并启动定时器（如果尚未启动），最后发送数据包并记录日志。

**零窗口探测**

零窗口探测超时线程的工作方式类似于普通超时线程，但不会调用 resent 函数进行数据重传。而是获取接收窗口大小并发送零窗口探测包 zero\_window\_probe 进行探测。

## **4.4 连接关闭的实现**

与连接关闭相关的函数主要是tju\_close和tju\_handle\_packet。

1. tju\_close

由主动提出关闭连接的一方调用，调用时发送FINACK报文，告知对方想要关闭连接。

同时进入阻塞等待状态，等待自己sock的状态变为CLOSED完成连接关闭。

1. tju\_handle\_packet

与建立连接过程的逻辑相同，也是通过flag来控制状态的转变以及做出相应的反应。下面是具体实现的伪代码：

如果状态是 ESTABLISHED

如果 flag 是 FIN\_FLAG\_MASK | ACK\_FLAG\_MASK

创建并发送 ACK 包

更新状态为 CLOSE\_WAIT

创建并发送 FIN-ACK 包

调试信息 "FIN sent! sock state -> LAST\_ACK"

更新状态为 LAST\_ACK

如果状态是 FIN\_WAIT\_1

如果 flag 是 ACK\_FLAG\_MASK

更新状态为 FIN\_WAIT\_2

否则如果 flag 是 FIN\_FLAG\_MASK | ACK\_FLAG\_MASK

调试信息 "FIN received! sock state -> CLOSING"

创建并发送 ACK 包

更新状态为 CLOSING

如果状态是 FIN\_WAIT\_2

如果 flag 是 FIN\_FLAG\_MASK | ACK\_FLAG\_MASK

更新状态为 TIME\_WAIT

创建并发送 ACK 包

更新状态为 CLOSED

如果状态是 LAST\_ACK

如果 flag 是 ACK\_FLAG\_MASK

更新状态为 CLOSED

如果状态是 CLOSING

如果 flag 是 ACK\_FLAG\_MASK

更新状态为 TIME\_WAIT

更新状态为 CLOSED

## **4.5 拥塞控制的实现**

**4.4.1 初始化变量**

在socket函数中，初始化拥塞窗口（cwnd）、慢启动阈值（ssthresh）和拥塞状态（congestion\_status）分别为IW（32倍的最大数据长度，MAX\_DLEN）、MSS（最大报文段大小，MAX\_DLEN）和SLOW\_START（慢开始阶段）。

**4.4.2 超时重传与慢开始阶段**

定义handle\_loss\_ack函数来处理超时重传后的拥塞控制状态变化，并在retransmit\_packet重传函数中调用此函数。首先，更新ssthresh为（cwnd+1）/2，cwnd为1个MSS。根据状态转换逻辑：

如果当前拥塞状态为SLOW\_START，且cwnd大于ssthresh，则状态变为CONGESTION\_AVOIDANCE（拥塞避免）。

如果当前拥塞状态为CONGESTION\_AVOIDANCE或FAST\_RECOVERY，超时后状态应重置为SLOW\_START（慢开始）。 记录状态变化的原因为timeout（超时）。

**4.4.3 处理正常到达的确认**

创建handle\_valid\_ack函数来处理收到正常到达的确认后的拥塞控制状态变化，并在handle函数中按序接收到ACK时调用。根据状态转换逻辑：

如果当前拥塞状态为SLOW\_START，cwnd增加1个MSS，并检查cwnd与ssthresh的关系，如果cwnd超过ssthresh，则状态变为CONGESTION\_AVOIDANCE，记录变化原因为SLOW\_START。

如果当前拥塞状态为CONGESTION\_AVOIDANCE，cwnd增加MSS乘以MSS除以cwnd的值，记录变化原因为CONGESTION\_AVOIDANCE。

如果当前拥塞状态为FAST\_RECOVERY，cwnd设置为ssthresh，状态变为CONGESTION\_AVOIDANCE，记录变化原因为FAST\_RECOVERY。

**4.4.4 处理重复的确认**

创建handle\_dup\_ack函数来处理收到重复ACK后的拥塞控制状态变化，并在handle函数中接收到重复ACK时调用。根据状态转换逻辑：

如果当前拥塞状态为FAST\_RECOVERY，cwnd增加1个MSS。

当收到的重复ACK数量（ack\_cnt）达到3时，将拥塞状态转变为FAST\_RECOVERY，ssthresh设置为cwnd的一半，cwnd设置为ssthresh加上3个MSS，并重置ack\_cnt为0。

如果ack\_cnt小于3，ack\_cnt自增一，状态保持不变。

# 五、实验结果及分析

测试所实现协议的功能和性能，并对性能结果进行分析。需要针对考察点逐一展开。

## **5.1 连接建立的功能测试与结果分析**

线上测试结果和本地测试相同，客户端和服务器能够正常建立连接。连接建立功能基本实现。

图形用户界面, 网站

描述已自动生成

表格

低可信度描述已自动生成

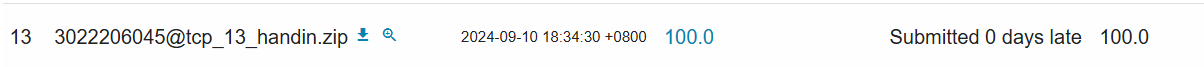
表格

描述已自动生成

## **5.2 可靠传输的功能测试与结果分析**

线上测试结果和本地测试相同，客户端和服务器能够正常传输数据。

可以在本地通过命令调整带宽、延迟、丢包率等信息。



## **5.3 流量控制的功能测试与结果分析**

在本次实验中，服务器和客户端之间的通信是单向的，因此在窗口大小的图表中，服务器仅显示接收窗口（rwnd），客户端仅显示发送窗口（swnd）。

接收窗口（rwnd）的变化分析：

初始化阶段：sock->received\_capacity 被设置为 128 倍的 MAX\_DLEN，这意味着一开始，接收窗口可以容纳 128 个数据包。

窗口动态调整：接收窗口的大小由 rwnd = advertised\_wnd = received\_capacity - received\_len 决定。其中，received\_len 表示已经接收但尚未处理的数据包长度。

日志记录机制：关于 RWND 的日志仅在数据包被放入接收缓冲区时记录。随着数据包的接收，received\_len 增加，导致 rwnd 减少。相对地，当数据包被处理并移出接收缓冲区时，received\_len 减少，rwnd 增加。

窗口大小变化的曲线趋势：

快速下降期：在数据传输初期，客户端可能同时发送多个数据包，导致服务器在短时间内接收大量数据，received\_len 迅速增加，使得 rwnd 快速下降。

稳定期：随着时间的推移，服务器端开始流水线式地处理接收到的数据包，即一边接收新数据包，一边将旧数据包处理并移至应用层 buffer。这种处理方式使得接收缓冲区的填充速率与数据包处理速率趋于一致，rwnd 的变化变得平稳。

这种窗口大小的变化模式展示了 TCP 协议在流量控制方面的自适应能力，确保了数据传输的连续性和稳定性，同时防止了接收方因处理速度跟不上而导致的缓冲区溢出。通过动态调整发送窗口大小，TCP 协议能够根据网络状况和接收方的处理能力，优化数据传输过程，提高网络的利用效率。

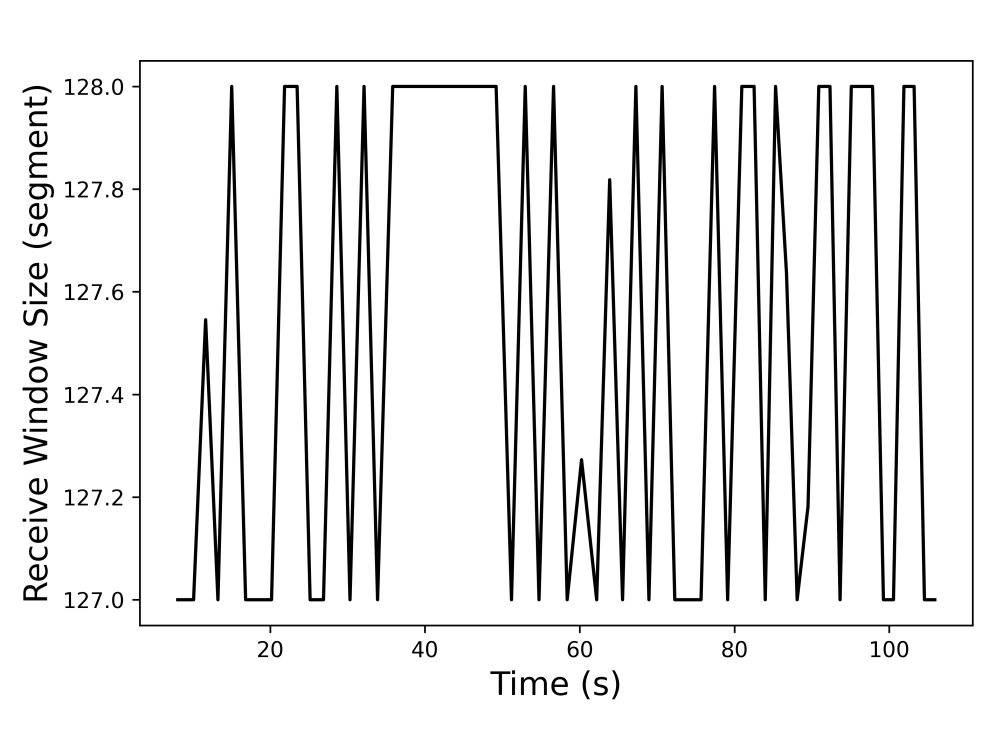


图 RWND变化

**估算的往返时间**（estmated\_rtt）：这个值是通过平滑处理样本往返时间（sample\_rtt）来计算的。计算公式是：

estmated\_rtt = (1 - ALPHA) \* estmated\_rtt + ALPHA \* sample\_rtt

其中，ALPHA 通常取值为 0.125。这意味着 estmated\_rtt 的更新会部分地反映最新的 sample\_rtt，同时保留之前估算值的一部分，以平滑短期波动。

**往返时间偏差**（deviation\_rtt）：这个值衡量了 sample\_rtt 与 estmated\_rtt 之间的偏差，计算公式是：

deviation\_rtt = (1 - BETA) \* deviation\_rtt + BETA \* |sampleRTT - estmated\_rtt|

BETA 通常取值为 0.25。这个计算方式同样采用了平滑技术，使得 deviation\_rtt 能够反映出 sample\_rtt 与 estmated\_rtt 之间的变化趋势，但变化幅度不会太大。

由图可见，由于 ALPHA 和 BETA 的取值较小，estmated\_rtt 和 deviation\_rtt 的变化不会对单个样本的极端值过于敏感，从而在动态网络环境中提供了稳定性。estmated\_rtt 的曲线与 sample\_rtt 的曲线变化相近，但更加平滑，而 deviation\_rtt 的变化幅度相对较小，这反映了它在捕捉 sample\_rtt 与 estmated\_rtt 之间短期波动的同时，保持了一定的稳定性。

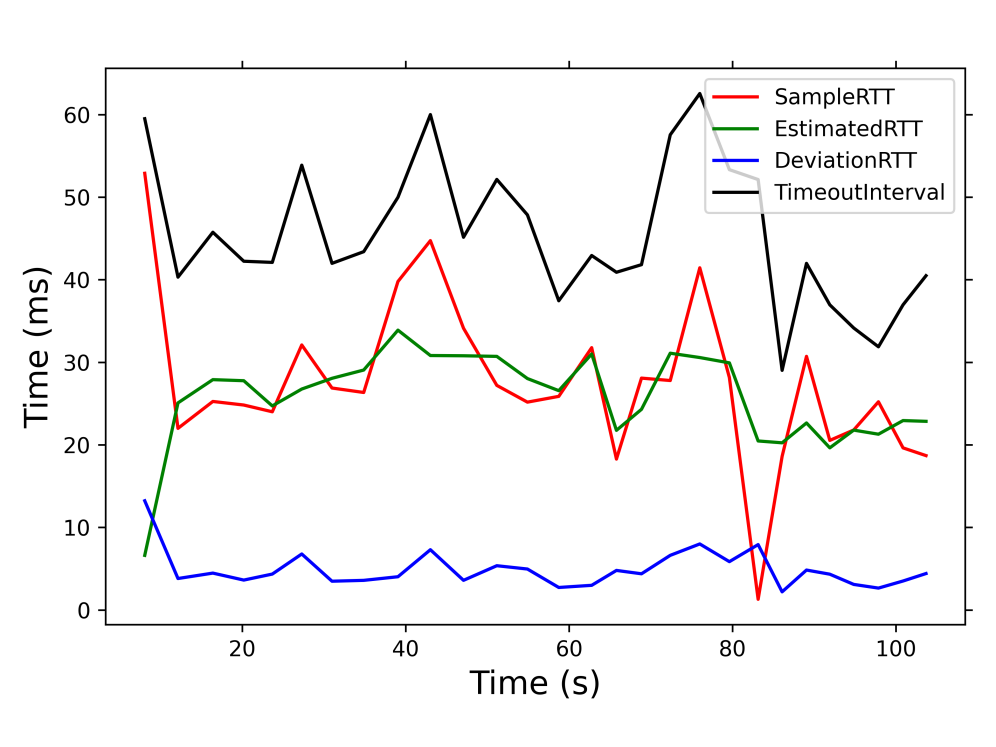


图 RTT的估算

观察吞吐量，反映数据传输效率的变化：

初始阶段：在 RDT 流程刚开始时，网络通常处于最佳状态，没有发生丢包和超时现象。此时，TCP 窗口可以充分利用可用的网络带宽，吞吐量因此较大。这是因为没有重传的需要，所有数据包都能够一次性成功传输，从而最大化了数据的传输效率。

中间阶段：随着数据传输的持续，可能会遇到一些网络波动，导致偶尔的丢包和超时。然而，由于 TCP 的流量控制和拥塞控制机制，吞吐量虽然略有降低，但仍然能够保持相对稳定。在这个阶段，RTT 的估算逐渐变得准确，发送方能够更好地预测网络的延迟情况，并通过调整发送窗口的大小来优化数据传输。同时，利用 advertised\_window 对 SND（发送窗口）进行准确限制，确保发送的数据量与接收方的处理能力相匹配，避免数据包丢失。

稳定上升期：随着时间的推移，TCP 对网络状况的适应性增强，RTT 的估算越来越准确，发送窗口的调整也越来越精确。这使得吞吐量整体上稳步上升，网络带宽得到更高效的利用。在这个阶段，TCP 能够根据网络的实际状况动态调整发送速率，避免因发送过快而导致的丢包和拥塞。

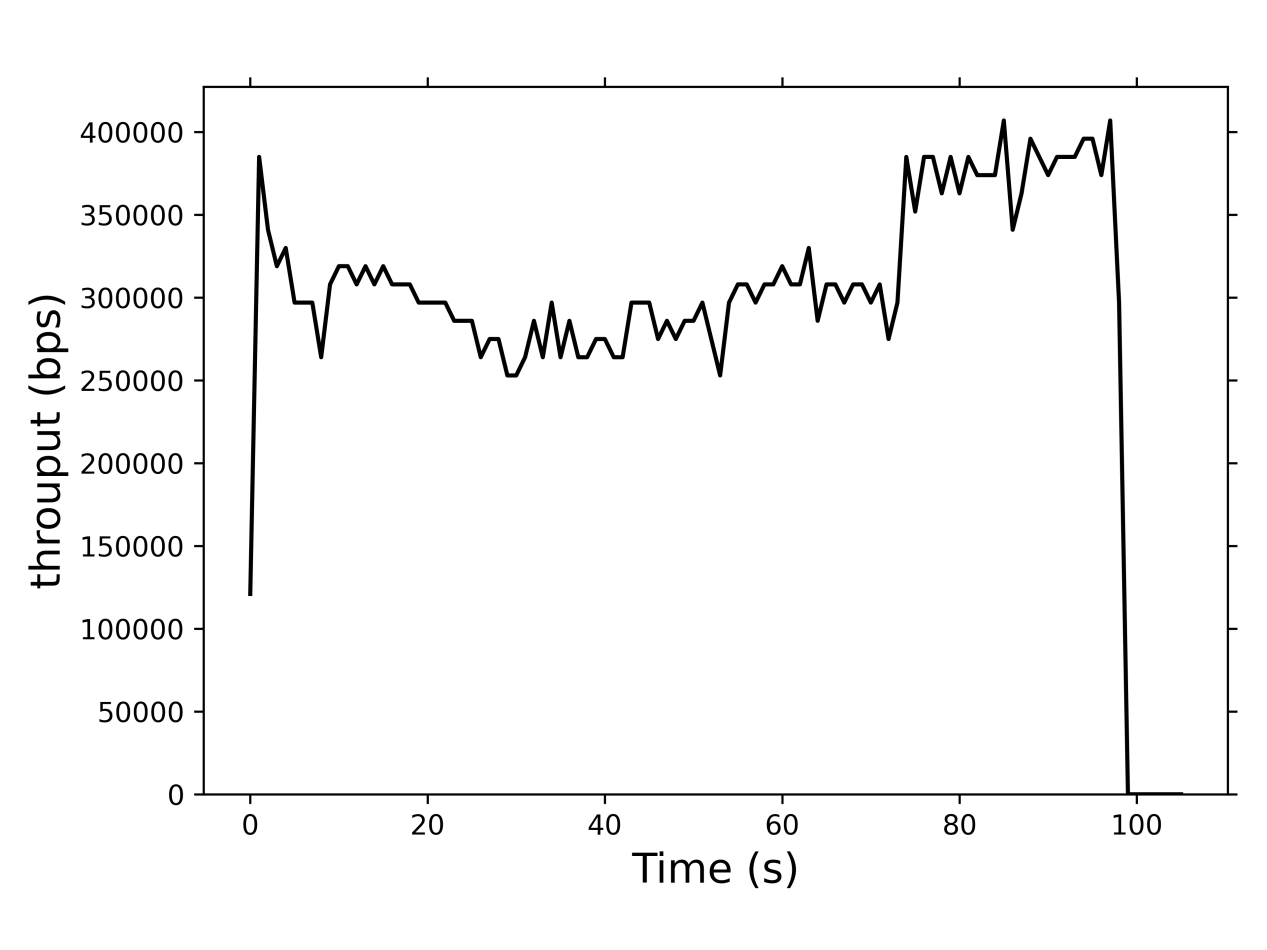
结束阶段：当 RDT 流程接近结束并开始关闭连接时，新的数据传输减少，吞吐量随之下降。这是因为发送方开始减少数据的发送，同时等待并确认所有已发送数据包的接收情况。在这个阶段，TCP 通过四次挥手过程来关闭连接，确保数据的完整性和可靠性。

图 吞吐量变化

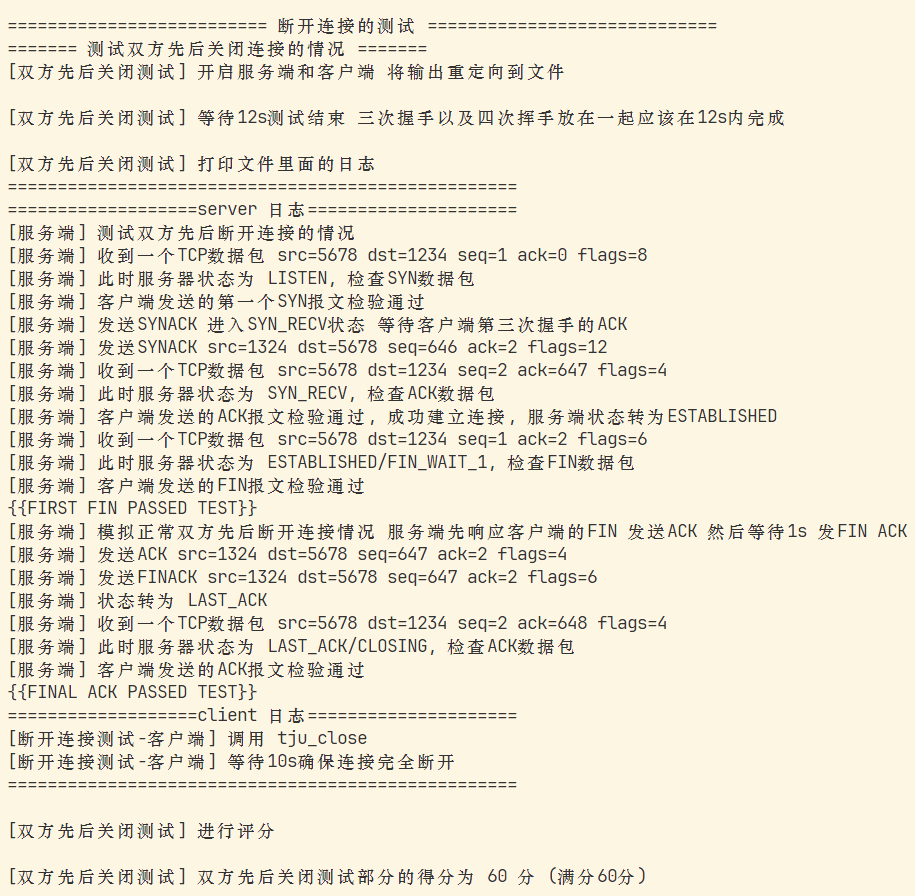
## **5.4 连接关闭的功能测试与结果分析**

客户端与服务端先后关闭连接和同时关闭连接的测试正常。连接关闭功能基本实现。

线上测试的日志基本与本地日志一致，只给出测试结果

图形用户界面, 文本, 应用程序, 电子邮件, 网站

描述已自动生成



文本

描述已自动生成

## **5.5 拥塞控制的功能测试与结果分析**

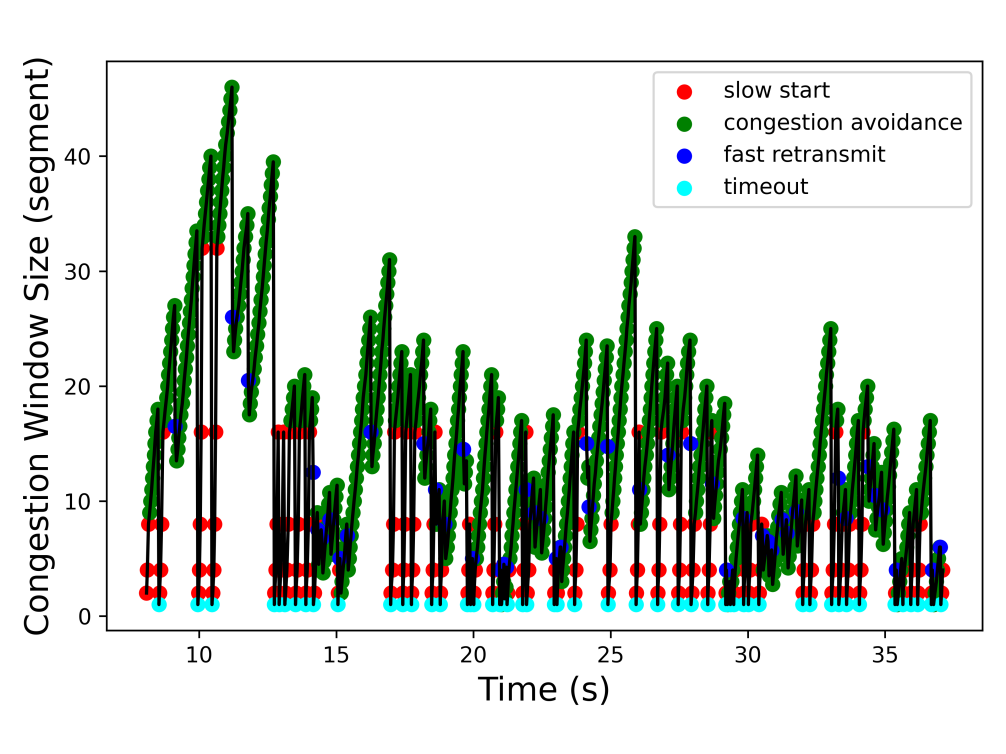


图 5-5-1 congestion窗口变化图

在传输过程的初始阶段，拥塞窗口的增长图示如下。观察到拥塞窗口从2个最大报文段（MSS）开始逐步扩大。一旦达到慢启动阈值（ssthresh），窗口增长转入拥塞避免阶段，此时增长速率减缓。只有在超时事件发生时，拥塞窗口才会被重置至1个MSS，并重新进入慢开始阶段。值得注意的是，在随后的增长周期中，ssthresh的值会调整为超时时拥塞窗口大小的一半。随着拥塞窗口的持续增长，一旦再次触及阈值，拥塞控制将再次切换到拥塞避免阶段。

此外，从图中还可以看到，当传输过程中收到三次重复的 ACK 包时，拥塞状态会变为快速重传阶段。此时，拥塞窗口调整为原来的拥塞窗口的一半加 3 MSS。在收到正确的确认后，退出快速重传阶段，拥塞状态再次变为拥塞避免阶段。

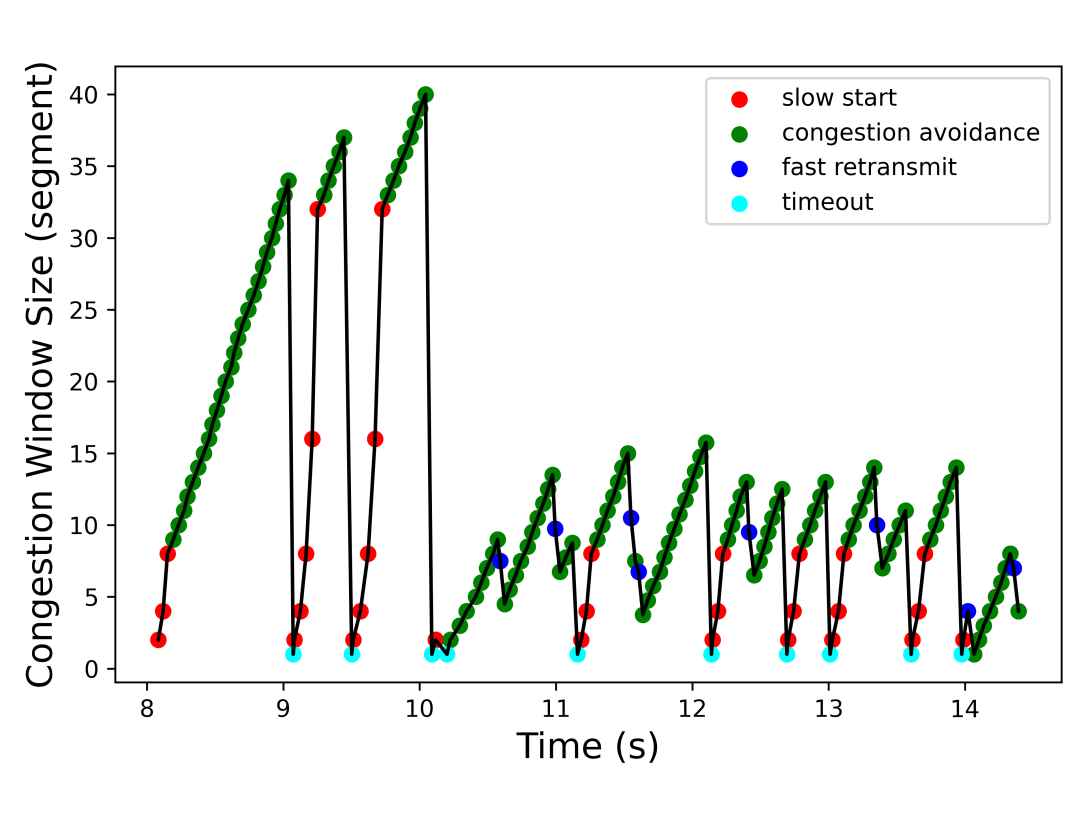


图 5-5-2 congestion窗口局部变化图

## **5.6 TCP协议性能测试与结果分析**

# 六、总结

总结在实践过程中遇到的各类问题、困难以及解决过程中的收获，对实践内容等方面的体会与建议。