天津大学

《计算机网络实践》课程报告

TCP 的设计与实现

学　　号 3019244266　3019244258

姓　　名 李润泽　　甘子奇

学　　院 智能与计算学部

专　　业 计算机科学与技术

年　　级 2019 级

任课教师 赵增华

2021 年　9 月　11 日

一、 报告摘要

本实验需要根据 TCP 的标准文档，在应用层实现 TCP 协议，称为 TJU_TCP。

协议中需要实现连接管理，通过“三次握手与四次挥手”实现建立连接与关闭连接的功能；对于流量控制，协议需要建立发送以及接收窗口，通过窗口滑动的原理实现，已达到消除发送方使接收方缓存溢出的可能性的目的；此外，协议需要实现可靠数据传输，在网络出现不可靠的状态时，也能可靠传输指定的文件并得以按序接收；最后，协议需要实现拥塞控制功能，可以实时根据网络中的业务状况动态调整拥塞窗口的大小。

协议中的功能全部得以实现，并在本地与自动验收网站全部通过测试，已成功验证了四个基本功能。

二、 任务分析

1）理解传输层协议的基本功能以及架构，使用 UDP 协议传输数据包和控制包，并掌握 TCP 报文段和 TCP 的几个主要功能等所设计协议的数据结构并进行实现。

2）理解并掌握面向连接的基本原理，需要对收发数据之前三次握手建立连接，结束后四次挥手断开连接等功能设计协议并进行实现。

3）理解并掌握流量控制的基本原理，设计 TCP 的流量控制协议并进行实现，通过流量控制来解决端到端的发送速率问题，以消除发送方使接收方缓存溢出的可能性。此模块发送方和接收方需要分别跟踪几个变量。

4）理解并掌握可靠数据传输的基本原理，确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流，设计相应的协议并进行实现。

5）理解并掌握拥塞控制的基本原理，解决网络引起的拥塞问题，设计出提高传输效率但又避免给网络造成太大压力的协议并进行实现。

三、 协议设计

3.1 总体设计

主要为四个部分：连接管理、流量控制、可靠数据传输、拥塞控制。

连接管理：TCP 发送方与接收方在数据传输前需要进行三次握手建立连接，在数据传输完之后需要进行四次挥手断开连接，需要利用 SYN,FIN,ACK 等报文中的字段。

流量控制：要实现流量控制功能，消除发送方使接收方缓存溢出的可能性，具体途径是设计滑动窗口，并利用 SWS 更新算法等，发送方维护 rwnd。

可靠数据传输：可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流；即该字节流与连接的另一方端系统发送出的字节流是完全相同。TCP 利用报文段头部结构中的检验和，序号，确认号来实现这一功能，并通过设重传计时器，利用快速重传以及选择确认的方式来提高数据传的输性能和数据传输的可靠性。

拥塞控制需要解决网络引起的拥塞问题，具体要实现的是慢启动、拥塞避免、快速恢复算法。

3.2 数据结构设计

TCP 协议的头部进行如下设计：

<table>
<tr><td colspan="2">source_port</td><td colspan="2">destination_port</td></tr>
<tr><td colspan="4">seq_num</td></tr>
<tr><td colspan="4">ack_num</td></tr>
<tr><td colspan="2">hlen</td><td colspan="2">plen</td></tr>
<tr><td>flags</td><td colspan="2">advertised_window</td><td>ext</td></tr>
</table>

下列是参数代表的意义：

Source_port：源端口

Destination_port：目标端口

Seq_num：表示本报文段所发送数据的第一个字节的编号。在 TCP 连接中所传送的字节流的每一个字节都会按顺序编号，若出现序列号回绕，再次从 0 开始

Ack_num：表示接收方期望收到发送方下一个报文段的第一个字节数据的编号。

Hlen： 表示首部长度

Plen： 协议地址长度

Flags： TCP 的编码位，包括 ACK,SYN,FIN 等

Advertised_window： 定义窗口大小，可作为滑动窗口解决流量控制

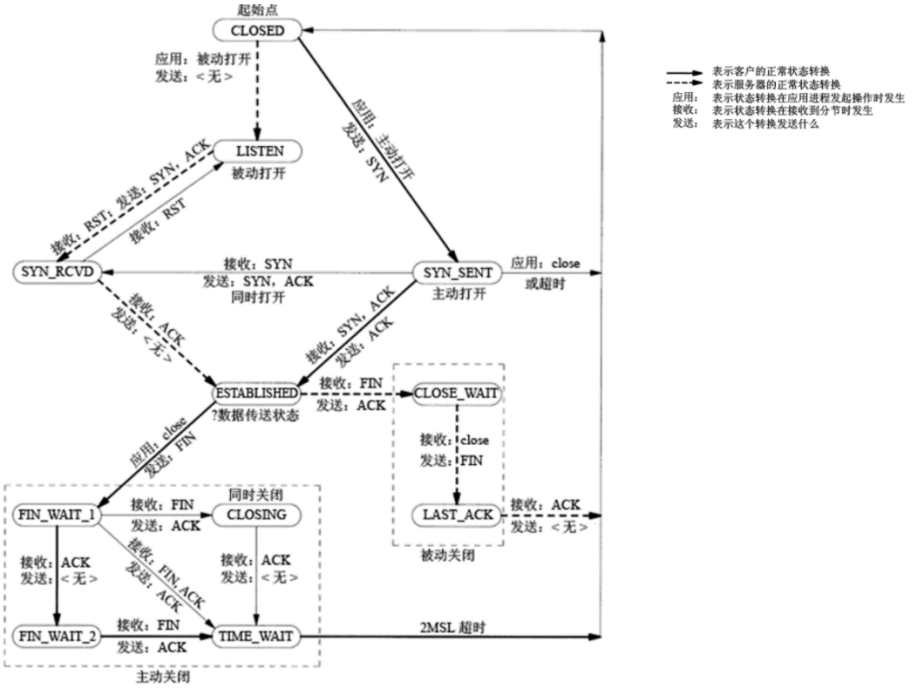
Ext： 额外数据

3.3 协议规则设计

3.4.1 连接控制

TCP 四个基本元素即发送方、接收方的 IP 地址，发送方、接收方的端口号，每方的 IP 和端口号可以看作是一个套接字。对于 TCP 的连接建立，分为连接建立，数据传输和断开连接三个部分。

该部分的 FSM 图表示为：



1.TCP 建立连接，三次握手

服务端进程准备好接收来自外部的 TCP 连接，一般情况下是调用 bind、listen、socket 三个函数完成。这种打开方式被认为是被动打开。然后服务端进程处于 LISTEN 状态，等待客户端连接请求。

客户端通过 CONNECT 发起主动打开，向服务器发出连接请求，请求中首部同步位 SYN = 1，同时选择一个初始序号 sequence ，简写

seq = x。SYN 报文段不允许携带数据，只消耗一个序号。此时，客户端进入 SYN-SEND 状态。

服务器收到客户端连接后,，需要确认客户端的报文段。在确认报文段中，把 SYN 和 ACK 位都置为 1 。确认号是 ack = x + 1，同时也为自己选择一个初始序号 seq = y。这个报文段也不能携带数据，但同样要消耗掉一个序号。此时，TCP 服务器进入 SYN-RECEIVED 状态。

客户端在收到服务器发出的响应后，还需要给出确认连接。确认连接中的 ACK 置为 1 ，序号为 seq = x + 1，确认号为 ack = y + 1。TCP 规定，这个报文段可以携带数据也可以不携带数据，如果不携带数据，那么下一个数据报文段的序号仍是 seq = x + 1。这时，客户端进入 ESTABLISHED 状态

服务器收到客户的确认后，也进入 ESTABLISHED 状态。

2.TCP 断开连接，四次挥手

客户端应用程序发出释放连接的报文段，并停止发送数据，主动关闭 TCP 连接。客户端主机 发送释放连接的报文段，报文段中首部 FIN 位置为 1 ，不包含数据，序列号位 seq = u，此时 客户端主机进入 FIN-WAIT-1 阶段

服务器主机接受到客户端发出的报文段后，即发出确认应答报文，确认应答报文中 ACK = 1，生成自己的序号位 seq = v，ack = u + 1，然后服务器主机就进入 CLOSE-WAIT 状态。

客户端主机收到服务端主机的确认应答后，即进入 FIN-WAIT-2 的状态。等待客户端发出连接释放的报文段。

这时服务端主机会发出断开连接的报文段，报文段中 ACK = 1，序列号 seq = v，ack = u + 1，在发送完断开请求的报文后，服务端主机就进入了 LAST-ACK 的阶段。

客户端收到服务端的断开连接请求后，客户端需要作出响应，客户端发出断开连接的报文段，在报文段中，ACK = 1, 序列号 seq = u + 1，因为客户端从连接开始断开后就没有再发送数据，ack = v + 1，然后进入到 TIME-WAIT 状态，请注意，这个时候 TCP 连接还没有释放。必须经过时间等待的设置，也就是 2MSL 后，客户端才会进入 CLOSED 状态，时间 MSL 叫做最长报文寿命。

服务端主要收到了客户端的断开连接确认后，就会进入 CLOSED 状态。因为服务端结束 TCP 连接时间要比客户端早，而整个连接断开过程需要发送四个报文段，因此释放连接的过程也被称为四次挥手。

3.4.2 流量控制

一条 TCP 连接的每一侧主机都为该连接设置了接收缓存，当该 TCP 连接收到正确、按序的字节后，它就会将数据放入接收缓存。相关的应用进程会从该缓存中读取数据，但不一定是数据刚一到达就立即读取。

事实上，接收方也许正忙于其他任务，甚至要过很长时间后才去读取该数据。如果接收方读取数据相对缓慢，而发送方发送得太多、太快，发送的数据就会很容易地使该连接的接收缓存溢出。

因此，我们需要建立流量控制功能，已达到消除发送方使接收方缓存溢出的可能性。

TCP 需要维护一个称为接收窗口的变量来提供流量控制。接收窗口用于给发送方一个指示：该接收方还有多少可用的缓存空间。窗口的大小必须被视为无符号数，否则过大的窗口大小将显示为负数，TCP 将无法工作，我们可以将连接记录中的发送和接收窗口大小保留 32 位字段，并使用 32 位进行所有窗口计算。

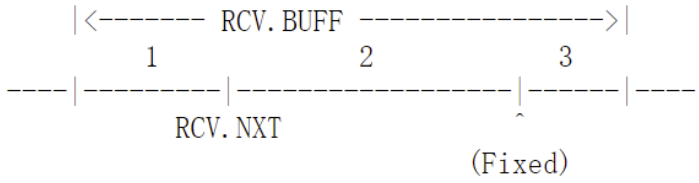
TCP 必须在接收器中包含 SWS 避免算法。接收器的 SWS 避免算法确定何时可以改变右侧窗口边缘，通常被称为“更新窗口”。

假设总接收缓冲区的空间为 RCV.BUFF，在给定任何时刻，缓冲区中的 TCP 数据 RCV.USER 与已接收且已确认但用户进程尚未使用的数据有关。

在连接处于静止状态时，RCV.WND = RCV.BUFF 且 RCV.USER = 0。

在数据到达并被确认时，为保持右侧窗口边缘处于固定状态，我们需要接收方提供小于完整地缓冲区空间，即当 RCV.NXT 增加时，接收方必须使得 RCV.WND 保持 RCV.NXT + RCV.WND 不变。

因此，缓冲区空间 RCV.BUFF 一般分为三个部分：



其中，RCV.USER 表示已接收但未读取的数据；RCV.WND 表示告知发送方空闲的空间，剩余的部分表示可用但未告知发送方的空闲空间。

为接收方提供的 SWS 避免算法是用于保持 RCV.NXT+RCV.WND 值不变，直到剩余空间满足：RCV.BUFF - RCV.USER - RCV.WND ≥ min{Fr * RCV.BUFF, Eff.snd.MSS}。

其中，Fr 是一个分数（推荐为 1/2），而 Eff.snd.MSS 表示用于连接的有效发送最大报文段长度；

当满足上述不等式时，RCV.WND 设置为 RCV.BUFF － RCV.USER。

这种算法是用于以 Eff.snd.MSS 为增量推进 RCV.WND，对于实际情况下的接收缓冲区，Eff.snd.MSS ＜ RCV.BUFF ／ 2。另外，假设接收方的 Eff.snd.MSS 与发送方相同，接收方必须使用自己的 Eff.snd.MSS。

3.4.3 可靠数据传输

因特网的网络层服务（IP 服务）是不可靠的。IP 不保证数据报的交付,不保证数据报的按序交付，也不保证数据报中数据的完整性。对于 IP 服务，数据报能够溢出路由器缓存而永远不能到达目的地，数据报也可能是乱序到达，而且数据报中的比特可能损坏（由 0 变为 1 或者相反）。由于运输层报文段是被 IP 数据报携带着在网络中传输 的，所以运输层的报文段也会遇到这些问题。

TCP 在 IP 不可靠的尽力而为服务之上创建了一种可靠数据传输服务。

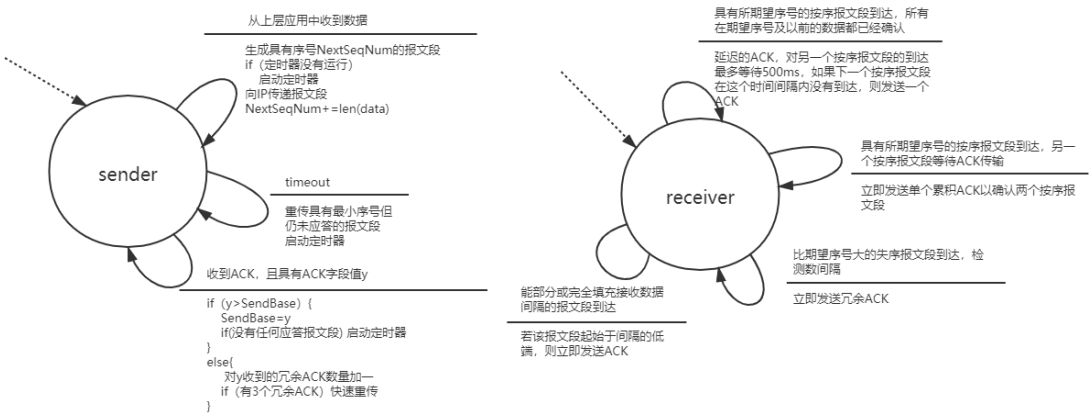
TCP 的可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流；即该字节流与连接的另一方端系统发送出的字节流是完全相同。

IP 提供的服务是尽力交付的服务，也是不可靠的服务。但是 TCP 在 IP 之上提供了可靠度传输服务。TCP 采用了流水线下的可靠数据传输协议,但是在差错恢复时,并没有简单的采取 GBN 协议或者选择重传协议，而是将二者结合了起来。

TCP 采用了累积确认的方式，这类似于 GBN，即如果 TCP 发送了对某个序号 N 的确认，则表明在 N 之前的所有字节流都已经被正确接收。但是另一方面，TCP 又不会像 GBN 协议那样简单丢弃失序到达的报文段，而是会将它们缓存起来，但是这些被缓存的报文段不会逐个被确认。当发生超时时，TCP 只会重传发生超时的那一个报文段。

TCP 还允许接收方选择性的确认失序到达的分组，而不是累积的对最后一个确认最后一个正确到达的分组，将它与 TCP 所采取的选择重传结合起来看就很想选择重传协议的工作机制。因此说 TCP 的差错恢复结合了 GBN 和选择重传。

可靠数据传输的 FSM 图表示为：



1.确认应答机制

发送方与接收方传输数据时，为了保证数据一定会发送给接受方，TCP 协议规定接受方接收到数据后需要给发送方一个接受到数据的通知，若是发送方没有接收到接受方的确认报文，发送方认为数据没有发送到，就会重新发送数据。

网络传输数据是不能保证传送顺序的，即使是客户端有序的发送数据，服务器端接受到的数据也有可能无序，因此为了保证有序，TCP 的包头中保存发送数据的序列号，告诉接受端读取数据的顺序。

确认序列号是发送方发送数据最高序号加 1，作用是对已经接收到的数据进行确认，并告知对方下次发送数据的位置。

2.超时重传问题

（1）RTO（重传时间）计算

1)在对发送方和接收方之间发送的段进行往返时间(RTT)测定之前，发送方设置 RTO 为 1 秒；

2）当获取第一个 RTT（测定为 R）时，

SRTT <- R

RTTVAR <- R/2

RTO <- SRTT + max{G, K * RTTVAR}

在这里 G 表示系统时钟粒度，一般很小；K = 4.

3）当后续的 RTT 被获取（测定为 R’）时，

RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R’|

SRTT <- (1 - alpha) * SRTT + alpha * R’

RTO <- SRTT + max{G, K * RTTVAR}

在这里 alpha = 1/8, beta = 1/4, K =4

无论何时，如果 RTO 小于 1 秒，那么 RTO 应当四舍五入到 1 秒。

（2）如果出现超时问题，

timer 设置如下：

1）每次数据报发送是，如果计时器没有启动，则启动计时器，并将超时时间设置为 RTO；

2）当所有数据确认时，关闭 timer；

3）当接收到新数据 ack，重启 timer。

发送方应当进行以下措施：

1）发送最早为确认的数据；

2）设置 RTO = 2 * RTO，若 RTO 过高会重置；

3）开启 timer；

4）如果 timer 等待 SYN/ACK 且 RTO 不足 3s，则 RTO 设置为 3s；

5）此外，接收到 3 个冗余 ack 也会触发快速重传。

3.检验和

用于检测在一个传输分组中的比特错误。发送方的 UDP 对报文段中的所有 16 比特字的和进行反码运算,求和时遇到的任何溢出都被回卷，得到的结果被放在报文段中的检验和字段。在接收方，全部的 4 个 16 比特字（包括检验和）加在一起。如果该分组中没有引入差错，则显然在接收方处该和将是 1111111111111111。如果这些比特之一是 0，那么我们就知道该分组中已经出现了差错。

4.快速重传

如果在超时重传定时器溢出之前，接收到连续的三个重复冗余 ACK（其实是收到 4 个同样的 ACK，第一个是正常的，后三个才是冗余的），发送端便知晓哪个报文段在传输过程中丢失了，于是重发该报文段，不需要等待超时重传定时器溢出。

5.选择确认

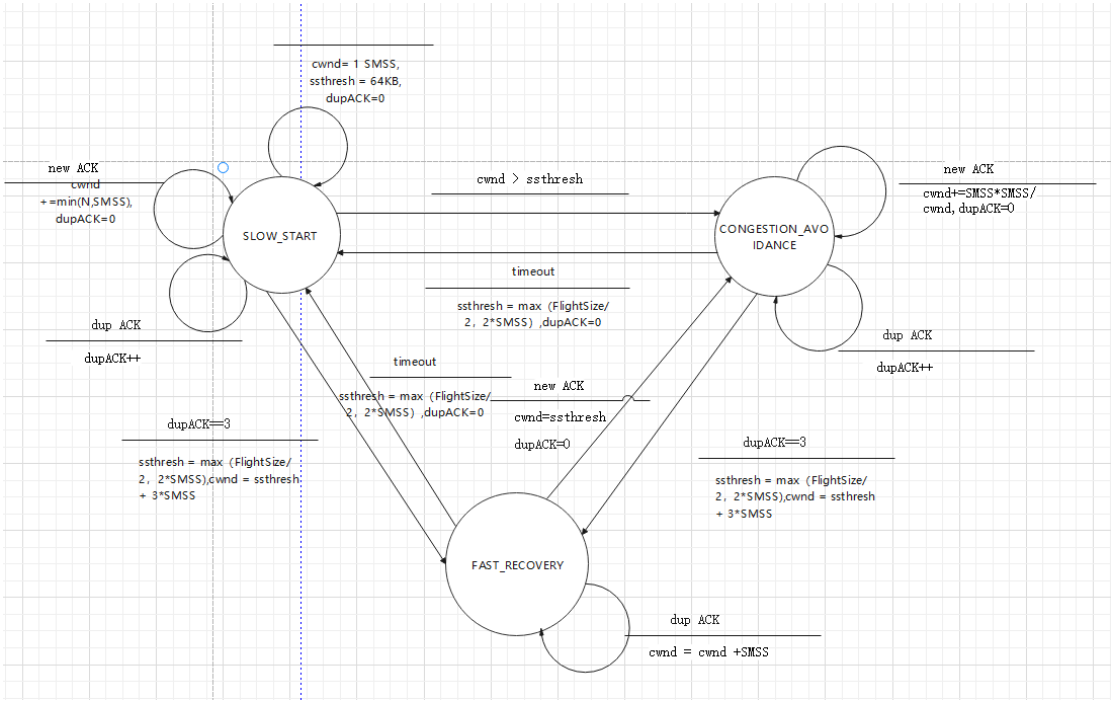
允许 TCP 接收方有选择地确认失序报文段，而不是累积地确认最后一个正确接收的有序报文段，即跳过重传那些已被接收方选择性地确认过的报文段。TCP 发送方仅需维持已发送过但未被确认的字节的最小序号（SendBase）和下一个要发送的字节的序号（NextSeqNum），而 TCP 接收方会将正确接收但失序的报文段缓存起来。

3.4.4 拥塞控制

TCP 有四种相互交织的拥塞控制算法：慢启动、拥塞避免、快速重传和快速恢复。

TCP 为了实现拥塞的控制，接收方，发送方各自维持一个窗口变量，拥塞窗口 (cwnd) 是对发送方在收到 ACK 之前可以传输到网络的数据量的发送方限制，而接收方的通告窗口 (rwnd) 是对接收方发送量的限制的数据。 cwnd 和 rwnd 的最小值控制数据传输。INITIAL WINDOW (IW)：初始窗口是三次握手完成后发送方拥塞窗口的大小。

FSM:



1.慢启动、拥塞避免

IW，cwnd 的初始值，必须使用以下准则作为上限进行设置。

SMSS 是发送方可以传输的最大段的大小。该值可以基于网络的最大传输单元，该大小不包括 TCP/IP 头部和选项。如果 SMSS > 2190 字节：IW = 2 * SMSS 字节且不得超过 2 个段；如果（SMSS > 1095 字节）和（SMSS <= 2190 字节）：IW = 3 * SMSS 字节且不得超过 3 个段；如果 SMSS <= 1095 字节： IW = 4 * SMSS 字节且不得超过 4 个段。

另外 SYN/ACK 和 SYN/ACK 的确认不得增加拥塞窗口的大小。此外，如果 SYN 或 SYN/ACK 丢失，则在正确传输 SYN 之后发送方使用的初始窗口必须是最多由 SMSS 字节组成的一个段。

当 cwnd < ssthresh 时使用慢启动算法，而当 cwnd > ssthresh 时使用拥塞避免算法。当 cwnd 和 ssthresh 相等时，发送方可以使用慢

启动或拥塞避免。

在慢启动期间，对于累积确认新数据的每个收到的 ACK，TCP 最多将 cwnd 增加 SMSS 字节。当 cwnd 超过 ssthresh（或者，可选地，当它达到它时，如上所述）或当观察到拥塞时，慢启动结束。根据：cwnd +=min(N,SMSS) (1)增加 cwnd，其中 N 是在传入 ACK 中确认的先前未确认字节数。

在拥塞避免期间，每个往返时间 (RTT) 将 cwnd 增加大约 1 个完整大小的段。拥塞避免一直持续到检测到拥塞为止。在拥塞避免期间增加 cwnd 的基本准则是：

* 可以通过 SMSS 字节增加 cwnd

* 每个 RTT 应该为每个等式 (1) 增加一次 cwnd

* 不得将 cwnd 增加超过 SMSS 字节

在拥塞避免阶段 TCP 以以下公式增加 cwnd：cwnd+=SMSS*SMSS/cwnd。在实现要注意：由于在 TCP 实现中通常使用整数算法，当拥塞窗口大于 SMSS*SMSS 时，公式（3）中给出的公式可能无法增加 cwnd。如果上述公式产生 0，则结果应向上舍入为 1 个字节。

当 TCP 发送方使用重传定时器检测到段丢失并且给定的段还没有通过重传定时器重发时，ssthresh 的值必须设置为不超过下面等式中给出的值：ssthresh = max（FlightSize/2，2*SMSS） (2)，FlightSize 是网络中未完成的数据量

2.快速恢复

1）在发送方收到的第一个和第二个重复 ACK 上，TCP 应该按照 [RFC3042] 发送一段先前未发送的数据，前提是接收方的广告窗口允许，总 FlightSize 将保持小于或等于 cwnd 加 2* SMSS，并且新数据可用于传输。此外，TCP 发送方不得更改 cwnd 以反映这两个段。除非传入的重复确认包含新的 SACK 信息，否则使用 SACK [RFC2018] 的发送方不得发送新数据。

2）当接收到第三个重复的 ACK 时，TCP 必须将 ssthresh 设置为不超过等式（2）中给出的值。当使用 [RFC3042] 时，在有限传输中发送的附加数据不得包含在此计算中。

3）必须重传从 SND.UNA 开始的丢失段，并将 cwnd 设置为 ssthresh 加 3*SMSS。这人为地将拥塞窗口“膨胀”了离开网络且接收器已缓冲的段数（三个）。

4）对于接收到的每个额外的重复 ACK（在第三个之后），cwnd 必须由 SMSS 递增。这人为地扩大了拥塞窗口，以反映已经离开网络的附加

段。

5）当先前未发送的数据可用且 cwnd 的新值和接收方的广告窗口允许时，TCP 应该发送 1*SMSS 字节的先前未发送数据。

6）当下一个确认先前未确认数据的 ACK 到达时，TCP 必须将 cwnd 设置为 ssthresh（在步骤 2 中设置的值）。这被称为“放气”窗口。

四、 协议实现

4.1 连接管理

客户端首先创建 socket，之后调用 tju_connect 函数与服务端进行三次握手，同时 server 创建好了 server_socket,调用 tju_bind 绑定端口，调用 tju_listen 函数将创建的 socket 放入 listen 哈希表。两者三次握手之后，就可以进行数据传输。在关闭过程中，首先客户端调用 tju——close 函数主动发起关闭请求，与服务端进行四次挥手，最终关闭了连接。在这个过程中 tju_handle_packet 函数负责处理收到的包，根据当前 socket 的状态以及收到的包的属性，进行状态的转变以及发送响应的包等操作。

4.1.1 建立连接——三次握手

客户端：

（1）主动建立连接，调用 tju_connect(),发送 SYN 包给服务端，状态由 CLOSED 变为 SYN_SEN，

（2）收到 SYNACK 包，发送一个 ACK 包并状态变为 ESTABLISHED，此时 tju_connect()处的阻塞停止，将 sock 放入 established 哈希表。

服务端：

（1）首先半连接队列和全连接队列全为空，调用 tju_bind()，tju_listen(),对 socket 与本地 ip 和 port 绑定，并加入到 listen 哈希表，状态由 CLOSED 变为 LISTEN。

（2）当收到 SYN 包，复制当前 sock 为 newconn，将 newconn 放入半连接队列中，向客户端发送一个 SYNACK 包，状态变为 SYN_RECV。

（3）若收到 ACK 包，将 newconn 状态变为 ESTABLISHED，将其从半连接队列放到全连接队列中，同时放入 established 哈希表中，此时 tju_accpt 处的阻塞停止，tju_accpt 返回全队列中的 sock，即为 newconn。

4.1.2 关闭连接——四次挥手

客户端：

（1）主动关闭连接，调用 tju_closet(),发送 FIN 包给服务端，状态由 CLOSED 变为 FIN_WAIT_1。

（2）收到 ACK 包，状态变为 FIN_WAIT_2。

（3)收到 FINACK 包,向服务端发送一个 ACK 包,状态变为 TIME_WAIT，等待 2MSL，状态再变为 CLOSED。此时 tju_closett()处的阻塞停止，连接关闭成功。

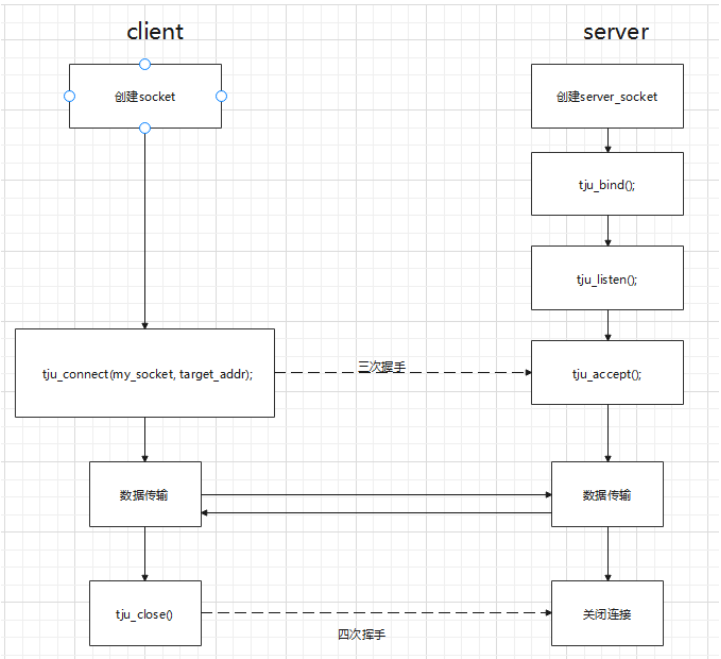
服务端：

（1）被动关闭连接，在 ESTABLISHED 状态下，若收到 FIN 包，回发一个 ACK，状态变为 CLOSED_WAIT。

（2）等待一段时间，再向客户端发送一个 FINACK 包，状态转为 LAST_ACK。

（3）收到 ACK 后，状态变为 CLOSED，连接关闭。

连接管理的流程图如下：



连接管理的伪代码如下：

```
tju_socket(){
    sock_state = CLOSED;
    Accept_num =0;
}
```

```
tju_bind(){
    sock->bind_addr = bind_addr;
}
tju_listen(){
    sock_state = listen;
    listen_sock[hashval] = sock;
}
tju_accept{
    while(AcceptQueue_n) ;//若全连接列表为空，就阻塞
    AcceptQueue_n  = 0;
    return AcceptQueue[Accept_num];//返回建立了连接的 sock
}
tju_connect{
    sock->state = SYN_SENT;
    sendtolayer3 (SYN);
    while(sock->state != ESTABLISHED);//若没建立连接则阻塞
    established_socks[hashval] = sock;
}
tju_close(){
    sendToLayer3(FIN);
    sock->state = FIN_WAIT_1;
    while (my_socke_state != CLOSED);//若没关闭就一直阻塞
}
tju_handle_packet(){
    if (sock->state == LISTEN){
        //服务端若收到 SYN
        new_conn = sock;//复制 sock 为 new_conn
        new_conn_state = SYN_RECV;
        sock->state = SYN_RECV;
        SynQueue[SynQueue_n] = new_conn;//放入半连接队列
        SynQueue_n = 1;
        sendtolayer3 (SYNACK);
    }
    else if(sock->state == SYN_SENT){
        //客户端若收到对应 SYNACK
```

```
        sock->state = ESTABLISHED;
        sendToLayer3(ACK);
    }
    else if(sock->state == SYN_RECV){
        //服务端若收到对应 ACK
        //将其从半连接队列删除,放到全连接队列中
        AcceptQueue[AcceptQueue_n] = SynQueue[SynQueue_n];
        established_socks[hashval] = new_conn;
        AcceptQueue[AcceptQueue_n]_state = ESTABLISHED;
        AcceptQueue_n = 1;
        SynQueue_n = 0;
    }
    else if(sock->state==ESTABLISHED && flags==FIN_FLAG_MASK){
        //服务端若收到 FIN
        sendToLayer3(ACK);
        sock->state = CLOSE_WAIT;
        //等一段时间
        sendToLayer3(FINACK);
        sock->state = LAST_ACK;
    else if(sock->state == FIN_WAIT_1){
        //客户端若收到对应 ACK
        sock->state = FIN_WAIT_2;
    }
    else if(sock->state == FIN_WAIT_2){
        //客户端若收到对应 FINACK
        sendToLayer3(ACK);
        sock->state = TIME_WAIT;
        //等待 2MSL
        sock->state = CLOSED;
    }
    else if(sock->state == LAST_ACK){
        //服务端收到对应 ACK
        sock->state = CLOSED;
    }
}
```

4.2 流量控制

在发送数据包的过程中，发送方调用 tju_send()函数，将数据打包并放入发送缓冲区。在创建 socket 时会新建一个发送线程，此发送线程会循环调用发送函数，当缓冲区不为空，且满足发送窗口的约束时，就发送数据包。

在接收数据包的过程中，tju_handle_packet()函数里处理接收的数据包，放入接收缓冲区。接收方调用 tju_recv()函数，从缓冲区中拿出相应长度的数据。

对于发送方：

当发送方发现接收方接收窗口大小不够时，就持续发送探测报文，直到 rwnd>=len，其次当发送缓存无法存放要发送的数据时，要阻塞等待之前的数据收到对应 ACK，之后将数据放入缓存区中，并将数据打包发送给接收方，并开启计时器，记录这次发送的数据的长度，更新 nextseq。

当发送方收到 ACK，要更新 rwnd 为 ack 中的 advertised_window，base 移动，如果当前还有未被确认的报文段，若 computing_RTT 为 1 则更新计时器，启动计时器;若没有未被确认的报文段，若 computing_RTT 为 1 则更新计时器，关闭计时器。

当收到报文的 flag 为 RETURN_NEW__RWND,说明接收方发来的更新 rwnd 的报文，直接将 rwnd 更新为包中 advertised_window。

对于接收方：

在 tju_rece()函数处窗口中有足够长度的数据到来，否则阻塞，当接收到指定长度的数据就将其放入缓冲区，并更新 LasByteRead，使其加上收到数据长度。

当收到报文的 flag 为 GET_NEW_RWND,说明是发送端发来的探测报文，首先记下窗口中已经使用了的空间，计算出最新的接收窗口大小，封装发送 RETURN_NEW__RWND 包给发送端。

当收到数据，当当前的接收窗口放不下,，则重新不断监听接收窗口的大小，直到放得下才结束监听，在按序接收到分组的情况下，若另有一个按序的报文到达，则发送累积的 ACK，否则发送改分组的 ACK，当 LastByteRcvd 等于 expect_seq，就更新 LastByteRcvd 为 pkt_seq+data_len，这里 expect_seq 是可靠数据功能的参数，在这里表示传输正常，同时计算 rwnd 时也需要用到。将数据放入缓冲区，更新 data_mark 数组，查询之后的数据是否已经被缓存，再决定发送 ACK 的

序号和确认号，最后计算当前的接收窗口，发送给发送方让其更新 rwnd 窗口。

流量控制的伪代码如下：

```
tju_socket(){
    //初始化
}
tju_send(){
    while(rwnd < len){
        //当发送方认为接收方没有可用空间时，持续发送探测报文
        sendToLayer3(get_new_rwnd)
    }
    window_size = MIN(rwnd, cwnd);
    while(nextseq - base + len > window_size){
        //当窗口无法存放当前所有数据时，所有数据等待前面的数据收
        //到 ACK 后才可以退出循环，继续发送数据
        window_size=MIN(rwnd,cwnd);
    }
    //将要发送的数据放入发送缓冲区中
    //启动定时器
    sendToLayer3(sending_pkt);
    //记下数据长度
    nextseq += len;
}
tju_recv(){
    while(expect_seq - LastByteRead < len);
    //等待指定长度的数据到来
    //将数据放入缓冲区
    LastByteRead += len;
}
tju_handle_packet(){
    if(sock->state == ESTABLISHED && && flags != FIN_FLAG_MASK){
        switch (flags){
            case ACK_FLAG_MASK:{
                //收到 ACK
```

```
    if(get_ack(pkt) > base){
        //更新 rwnd，base 前移
        if(base !=nextseq){
            //当前还有未被确认的报文段
            if(computing_RTT == 1)//更新计时器
            //启动计时器
        }
        else{
            //没有未被确认的报文段了
            if(computing_RTT == 1)//更新计时器
            //关闭计时器
        }
    }
    break;
}
case GET_NEW_RWND:{
    //此时发送端认为 rwnd 已经不够接收了，所以持续发来
//探测报文
    used_sum = 0;
    for( i from expect_seq to LastByteRcvd){
        //如果缓冲区对应位置有数据就记录
    }
    // 向对方发送 ACK，并通知对方更新 rwnd
    sendToLayer3(update_rwnd_pkt);
    break;
}
case RETURN_NEW_RWND:{
    //收到数据，更新相关参数
}
default:{
    //此时接收到对方发来的数据，需要更新对方的 rwnd 值
    for( i from expect_seq to LastByteRcvd){
        //如果缓冲区对应位置有数据就记录
    }
    //计算剩余的 rwnd
```

```
                while(temp_rwnd - data_len < 0){
                    //接收不下数据，重新计算
                    for( i from expect_seq to LastByteRcvd){
                        //如果缓冲区对应位置有数据就记录
                    }
                    //计算剩余的 rwnd
                }
                if(get_seq(pkt) == expect_seq){
                    //按序收到分组
                    if(LastByteRcvd == expect_seq){
                        LastByteRcvd = get_seq(pkt) + data_len;
                    }
                    //收到数据，把数据存入接收缓冲区
                    //更新 data_mark
                    //查询之后的数据是否已经被缓存
                    //更新 expect_seq
                    //再决定发送 ACK 的序号和确认号
                    for( i from expect_seq+data_len to TCP_RECVWN_SIZE ){
                        if(data_mark[i] == 0) break;
                    }
                    expect_seq = i;
                    //计算当前可用缓存
                    //通知对方更新 rwnd，大小为 temp_rwnd
                }
            }
        }
    }
}
```

4.3 可靠数据传输

在可靠数据传输部分中，我们需要将 ack 与 seq 分开看待，即发送方的 seq 始终与接收方的 ack 关联，而发送方的 ack 始终与接收方的 seq 关联，而同一个包的 ack 与 seq 实际上是没有关联的，此外，发送回的 ack 包实际上是一个普通的、没有数据的包。

4.3.1 收到预期的 packet（正常情况）

正常情况下，接收方将数据包中的数据放入缓存区，并回送一个 ack 包，由于 ack 不携带数据，故不必考虑滑动窗口，可以直接发送。

对于接收到数据的一方而言，返回的数据包中

```
ack = get_seq(pkt) + get(plen) - DEFAULT_HEADER_LEN;
seq = get_ack(pkt);
```

4.3.2 检验和

根据 TCP 校验和的计算方法，把伪首部、TCP 报头、TCP 数据分为 16 位的字(若为奇数字节补 0)；用反码相加法累加所有的 16 位字，使用循环进位的方法，最后结果为校验和。

4.3.3 丢包、超时

每次发送数据包的时候，都需要使用定时器对其进行响应的计时，如果在超时间隔内收到了响应的 ack，就停止计时；若超时则重新发送，并更新超时间隔。

4.3.4 乱序

接收方在接收到不是预期的包时，就返回 duplicated_ack，并发送回去，返回的 ack 等于上一个发送回的 ack。

发送方若接收到连续的三个重复冗余 ACK（其实是收到 4 个同样的 ACK，第一个是正常的，后三个才是冗余的），发送端便知晓哪个报文段在传输过程中丢失了，于是重发该报文段。

对于发送方：

调用 tju_send(),发送数据并开启计时器

当收正常到 ACK,更新 base，rwnd。若还有未确认的报文段，重启计时器，若 computing_RTT 为 1，则更新计时器，否则不更新；若没有未确认的报文段，关闭计时器，若 computing_RTT 为 1，则更新计时器，

否则不更新。

若收到冗余 ack，记录其数量，当冗余 ack 数量为 3 时，进行快速重传，取得 paket_len[base]的长度，即第一个未被确认的报文的数据长度，重新发送给接收方。

收到最后一个发出去的 pkt 的 ack，更新 rwnd，base 前移，关闭计时器

当计时器超时时，若 nextseq==base 则不重传。其他情况下，取得 paket_len[base]的长度，即第一个未被确认的报文的数据长度，打包重新发送给接收方。

对于接收方：

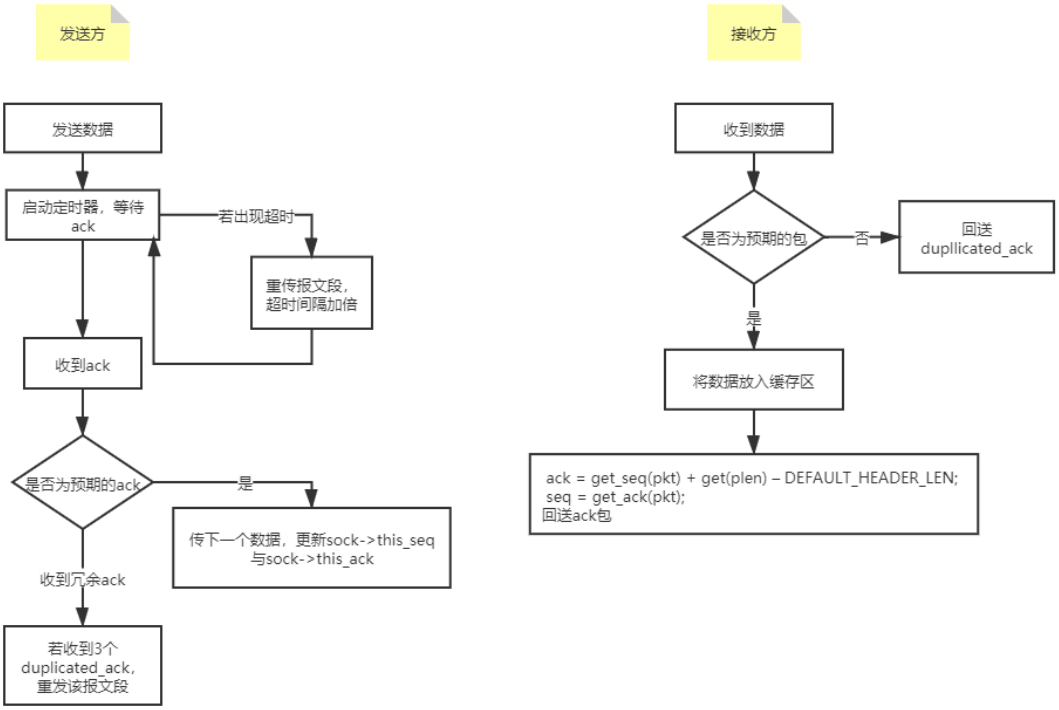
调用 tju_recv(),等待指定长度的数据到来,收到数据就放入缓冲区首先若收不下传来的数据，就要不断监听 rwnd 的大小，若有足够空间则进行接下来的操作

当收到按序分组，则按流量控制中的操作进行更新参数，移动窗口。

当收到失序分组，将数据放入缓冲区，更新 data_mark[],并发送冗余的 ACK。

当收到已经接收过且发送过 ACK 的数据。因为若发送在还没有接收到快速重传发来的 ACK 之前，又触发了超时重传。而之后收到快速重传的 ACK 后，发送窗口的 base 后移，之前超时重传数据到达接收方。此时若不返回 ACK，发送方以为快速重传数据又丢失了，于是继续重传数据，此时接收方必须给予回应此时收到的是已被接收过的数据，但也要发送一个 ACK。

可靠数据传输的流程图为：



可靠数据传输的伪代码如下：

```
tju_socket(){
    //初始化
}
tju_send(){
    sendToLayer3(sending_pkt);
    //开启计时器
    computing_RTT = 1;
}
tju_recv(){
    while(expect_seq - LastByteRead < len);
    //等待指定长度的数据
    //收到数据就放入缓冲区
}
tju_handle_packet(){
    if(sock->state == ESTABLISHED && pkt_flags != FIN_FLAG_MASK){
        switch (flags){
            case ACK_FLAG_MASK:{
```

```
    //收到正常 ACK
    if(get_ack(pkt) > base){
        //更新 rwnd，base 前移
        if(base !=nextseq){
            //当前还有未被确认的报文段
            if(computing_RTT == 1) //更新计时器
            //启动计时器
        }
        else{
            //没有未被确认的报文段了
            if(computing_RTT == 1) //更新计时器
            //关闭计时器
        }
    }
    else if(get_ack(pkt) == base &&base < nextseq){
        //冗余 ack
        ack_cnt++;
        if(ack_cnt == 3){
            //触发超时,快速重传
            ack_cnt = 0;
            computing_RTT = 0;
            if(窗口状态不是拥塞避免或慢启动){
                //快速重传
            }
        }
    }
    else if(get_ack(pkt) == base && base == nextseq){
        //收到最后一个发出去的 pkt 的 ack，更新 rwnd
    }
}
default:{
    //此时接收到对方发来的数据，需要更新对方的 rwnd 值
    for(i from expect_seq to LastByteRcvd){
        //如果缓冲区对应位置有数据就记录
    }
```

```
                while(temp_rwnd - data_len < 0){
                    //接收不下数据，重新计算
                    for(i from expect_seq to LastByteRcvd){
                        //如果缓冲区对应位置有数据就记录
                    }
                    if(expect_seq  <  get_seq(pkt)  &&  get_seq(pkt)  <
LastByteRead + TCP_RECVWN_SIZE){
                        //收到失序分组，将数据标记，
                        //发送冗余 ACK，指示下一期待字节的序号
                        //数据放入缓冲区
                        //计算当前 rwnd
                        //发送冗余 ack
                    }
                    else if(get_seq(pkt) < expect_seq){
                        //接收到已经接收过且发送过 ACK 的数据
                        sendToLayer3(sent_ack);
                    }
                }
            }
        }
    }
}
timeout_thread(){
    //用于监控定时器的线程
    while(1){
        if(计时器启用了){
            if(超时){
                //超时间隔加倍
                //如果 nextseq==base，就不用重传了
                ack_cnt = 0;
                //重传
            }
        }
    }
}
```

4.4 拥塞控制

在拥塞控制的实现中，发送方分为三种状态：慢启动 SLOW_START、拥塞避免 CONGESTION_AVOIDANCE 和快速恢复 FAST_RECOVERY。下面将分为不同情况进行状态转化的分析：

4.4.1 若发送方接收到正常的 ACK

当发送方处于慢启动状态下，cwnd 扩大，发送效率呈指数上升，当 cwnd > ssthresh 时，慢启动状态转变为拥塞避免状态；

当发送方处于拥塞避免状态下，cwnd 线性增大；

当发送方处于快速恢复状态下，cwnd 缩小为 ssthresh，并转变为拥塞避免状态。

4.4.2 若发送方接收到 3 个冗余的 ACK

发送方均需要将 ack_cnt 回调为 0，ssthresh 设置为 cwnd/2，cwnd 设置为 ssthresh + 3*MAX_DLEN，并转变为快速恢复状态。

4.4.3 若发送的数据已超时

发送方需要将 ssthresh 设置为 cwnd/2，cwnd 设置为 MAX_DLEN，ack_cnt 回调为 0，并转变为慢启动状态。

拥塞控制实现部分的伪代码如下：

```
tju_socket(){
    //初始化
    cwnd = MAX_DLEN;
    ssthresh = TCP_SENDWN_SIZE;
    congestion_status = SLOW_START;
}
tju_send{
    //确定窗口大小，发送方的窗口大小取 rwnd 和 cwnd 最小值
    window_size = MIN(rwnd,cwnd);
}
tju_handle_packet{
    //处理收到的数据包
    if (sock_state == ESTABLISHED && not_FIN){
        //当已经建立连接且收到的不是 FIN 包
```

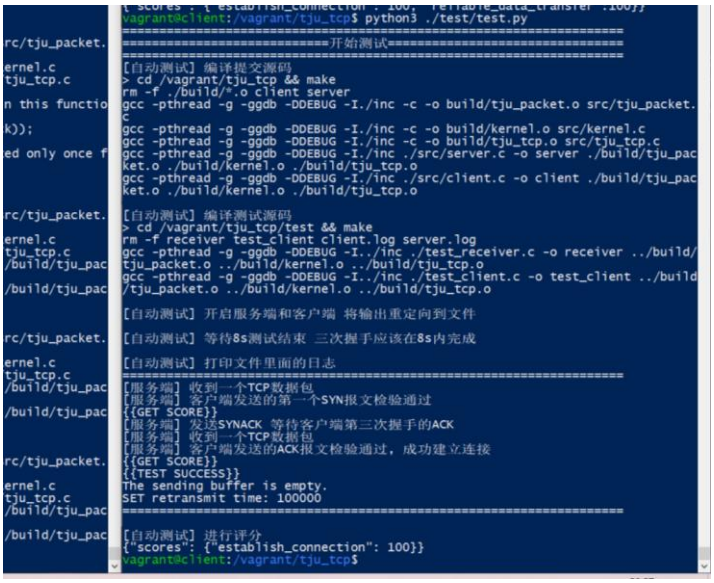
```
        if ( pkt_ack > base){
            //不是重复 ack
            //拥塞窗口大小更新为接收方发来的窗口大小
            //重置冗余 ack 数量为 0
            rwnd =pkt_dvertised_window();
            ack_cnt = 0;
            if(congestion_status == SLOW_START){
                //慢启动
                cwnd +=min(next_seq  -  base ,MAX_DLEN);
                //MSS 这里用 MAX_DLEN 代替
                if(cwnd >= ssthresh)
                    congestion_status=CONGESTION_AVOIDANCE;
            }
            else if(congestion_status == CONGESTION_AVOIDANCE){
                //拥塞避免
                cwnd += MAX_DLEN * MAX_DLEN / cwnd;
            }
            else{
                //快速恢复
                cwnd = ssthresh;
                congestion_status = CONGESTION_AVOIDANCE;
            }
        }
        else if(pkt_ack == base && base < nextseq){
            //收到冗余 ACK
            ack_cnt ++;
            rwnd = pkt_dvertised_window();
            if(ack_cnt == 3){
                //三个冗余 ack
                if(congestion_status == SLOW_START || congestion_ status
= CONGESTION_AVOIDANCE){
                    //转为快速恢复
                    ssthresh =cwnd / 2;
                    congestion_status = FAST_RECOVERY;
                    cwnd=ssthresh + 3 * MAX_DLEN;
```

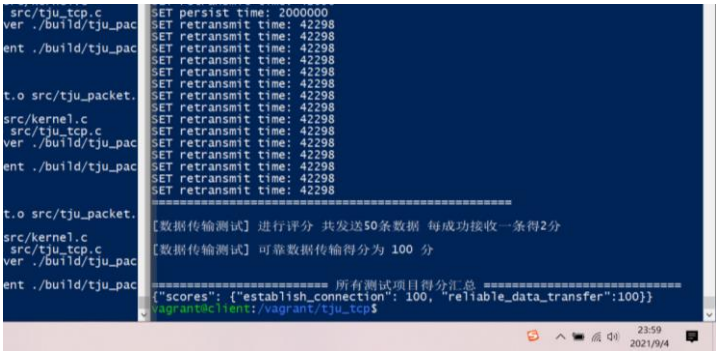
```
                }
            }
            if(congestion_status ==FAST_RECOVERY){
                //快速恢复
                cwnd += MAX_DLEN;
            }
        }
    }
}
overtime_thread(){
    //处理拥塞控制中的超时事件
    if(time_out){
        ssthresh = cwnd/2;
        cwnd = MAX_DLEN;
        ack_cnt = 0;
    }
}
```

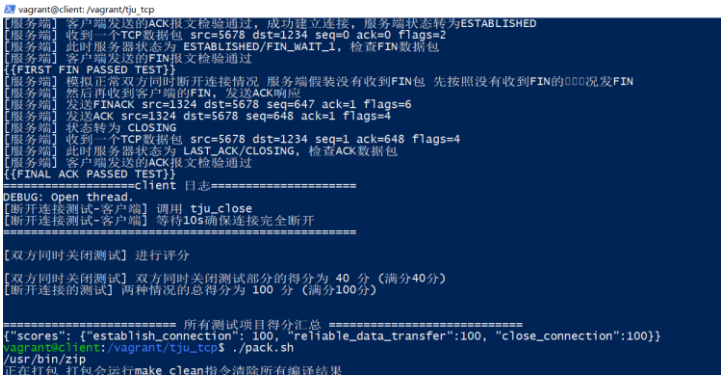
五、 实验结果及分析

5.1 本地测试分析

建立连接、可靠数据传输以及关闭连接已在本地测试通过，本地测试结果如下图所示：

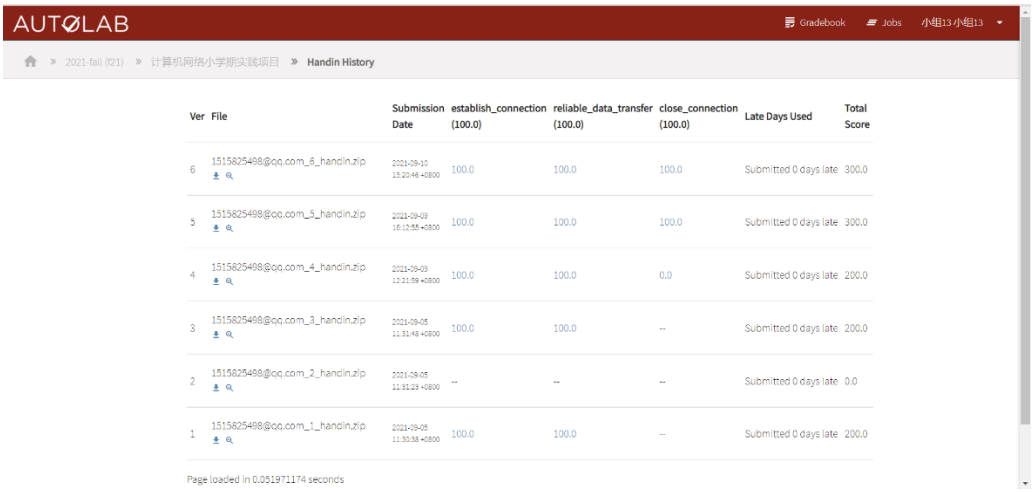






5.2 自动测试网站测试分析

自动测试网站测试结果如下图所示：



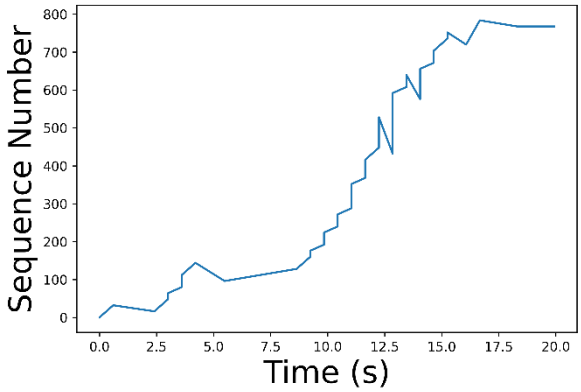
5.3 拥塞控制性能分析

在拥塞控制中，我们需要使用命令 python3 ./test_congestion.py 进行本地验证并绘图分析。

我们将 MSS 设置为 20，带宽 rate 设置为 100Mbps，延迟设置为 300ms，丢包率设置为 10%，进行 60 秒内的测试。

5.3.1 客户端发送 50 条数据

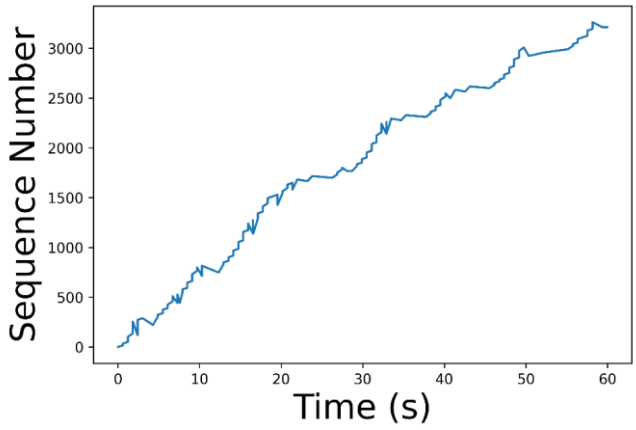
发送的序列号与时间的图像如下图所示：



开始的 5s 内处于慢启动阶段，但在刚开始时出现过一次冗余 ACK 的出现，故在慢启动阶段出现了快速恢复，5s 后又出现一次冗余 ACK 的出现，恢复后进入慢启动与拥塞避免阶段，直到 12.5s 左右先后出现了三次恢复，最终 50 条数据已全部发送完成。

5.3.2 客户端发送 200 条数据

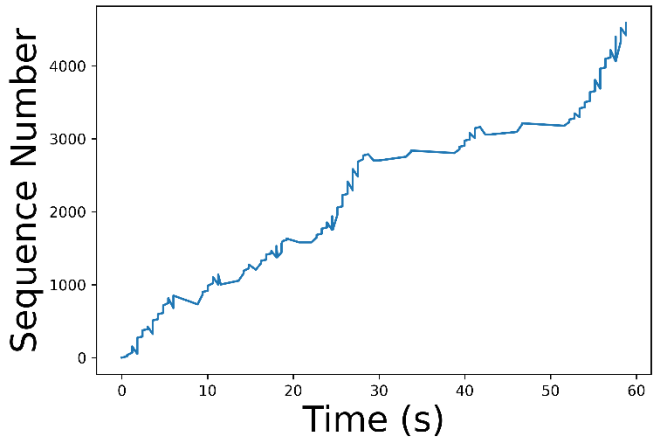
发送的序列号与时间的图像如下图所示：



由图可得，在前 3s 内处于慢启动状态，此后出现了多次重传，可知在相对平稳上升的部分为拥塞避免阶段，例如 5-10s，而后面也出现了呈指数上升的曲线，为慢启动阶段，例如 30-32s 与 46-50s，中间夹杂着 seq 下降的部分，为快速恢复阶段。最终 60s 内 200 条数据基本全部传输完毕。

5.3.3 客户端发送 1000 条数据

发送的序列号与时间的图像如下图所示：



曲线所反映的拥塞状态同 5.3.1 与 5.3.2。

虽然客户端要求发送 1000 条数据，但 seq 只发到了 4500 左右，因为 MSS 设置的过小，为 20，故在 60s 内并没有全部发送完毕。

六、 个人总结

在本次实验的过程中，小组成员收获颇丰，在上学期学习了计算机网络的理论知识后，这学期初的实践课通过进行 TCP 协议实现让我们对可靠的数据传输协议有了更加详尽的认识。

在进行实践的过程中，也或多或少的遇到了一些难题，例如流量控制中窗口的移动以及可靠数据传输定时器的使用是我在实现过程中遇到的问题，甚至在实现过程中，小组不得不重新编写数据结构，耽误了过多的时间。但通过对 RFC 文档的阅读以及资料的查询，使我们得以按时实现相应的任务，这将对我们在计算机方面的学习会有巨大的帮助，并提升自己代码编写的能力。