天津大学

TJU_TCP 的设计与实现

学院名称 智能与计算学部

专　　业 计算机科学与技术

姓　　名 迪丽菲娅

学　　号 3019244005

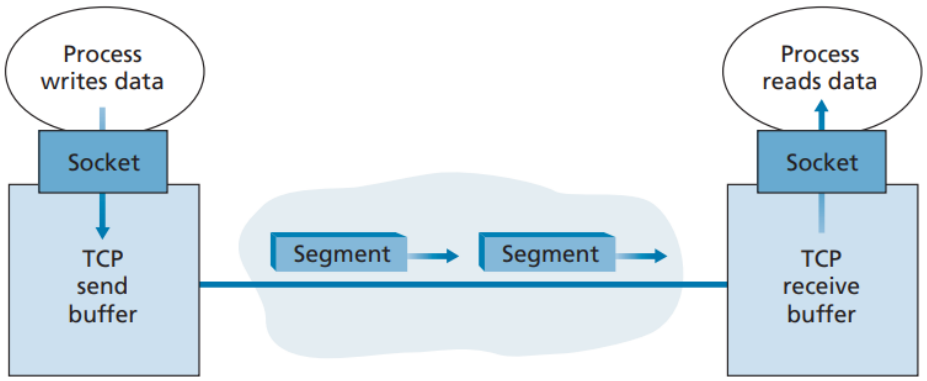
年　　级 2019 级

任课教师 石高涛

目录

报告摘要

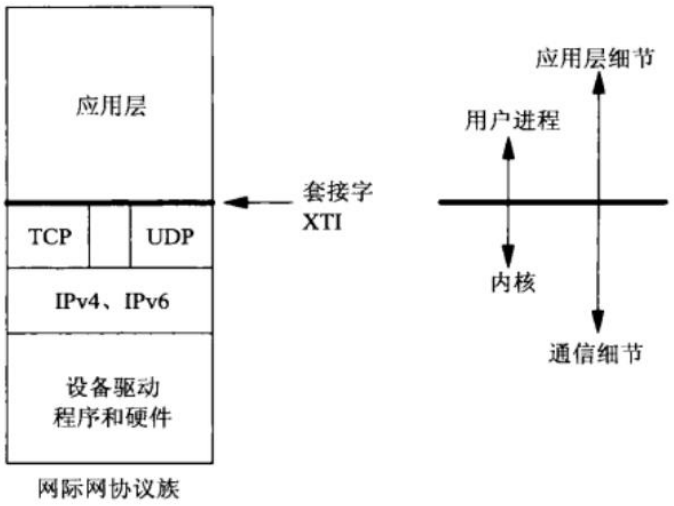
本实验中需要完成的是简化的传输层协议 TJU_TCP 的设计和实现。TCP 提供的是面向连接的、可靠的字节传输服务。面向连接意味着，客户端和服务端进程在交换数据之前需要通过客户端与服务端向彼此发送特殊的 TCP 报文段来先建立一个 TCP 连接，这个过程就被成为 “三次握手”。



一旦建立了 TCP 连接以后两个应用进程就可以互相发送数据了。客户进程通过套接字传递数据流，数据一旦经过该门，就会由客户端中运行的 TCP 控制。TCP 会将这些数据引导到该连接的发送缓存里，接下来 TCP 就会时不时的从缓存里取出数据并发往网络层。TCP 从网络层接收的数据则是会先存到接收缓冲区中，然后再通过 socket 发往应用层，如上图所示。

因此，可以看出 TCP 连接的组成包括：一台主机上的缓存、变量和与进程连接的 socket，以及另一台主机上的另一组缓存、变量和与进程连接的 socket。

在开始实验之前说明 socket 和 TCP 的关系显得比较重要。Socket 是同一台



主机内应用层与传输层之间的接口（在网络中的位置如上图所示）。也可以理解为，Socket 将复杂的 TCP/IP 协议族隐藏在接口函数（如 socket（）、bind（）、listen（）等等）后面。因此在此实验中 TJU_TCP 也会为应用层提供可调用的接口,通过完善提供的7个接口函数和其它函数来实现TJU_TCP提供的传输层服务。

具体实现包括按照四个模块（连接管理，可靠数据传输，流量控制，拥塞控制）对框架代码中的函数的完善和自己添加的一些函数。函数的补充和添加是按照具体的协议设计来实现的。最终的代码实现了 TCP 的连接管理，保证了可靠的数据传输，添加了拥塞控制，流量控制机制等。

任务分析

首先，TCP 提供客户与服务器之间的连接。TCP 客户先与某个给定服务器建立一个连接，然后通过该连接与服务器交换数据，最后终止该连接。该部分包括连接建立和断开：

1）三次握手 —— 建立连接

2）四次挥手 —— 断开连接

其次，TCP 提供了可靠性。当 TCP 向另一端发送数据时，要求对端返回一个确认 ACK。 如果没有收到确认， TCP 就自动重传数据并等待更长时间。可靠数据传输的实现部分选择了 GBN 协议。

1）实现发送、累计确认、超时重传功能

2）实时计算 RTT

TCP 提供流量控制和拥塞控制。流量控制：TCP 总是告知对端在任何时刻它一次能从对端接收多少字节的数据，即通知窗口，确保不会发生缓冲区溢出。拥塞控制：是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。需要实现的部分如下：

1）流量控制，基于滑动窗口协议

2）慢启动

3）拥塞避免

4）快速重传

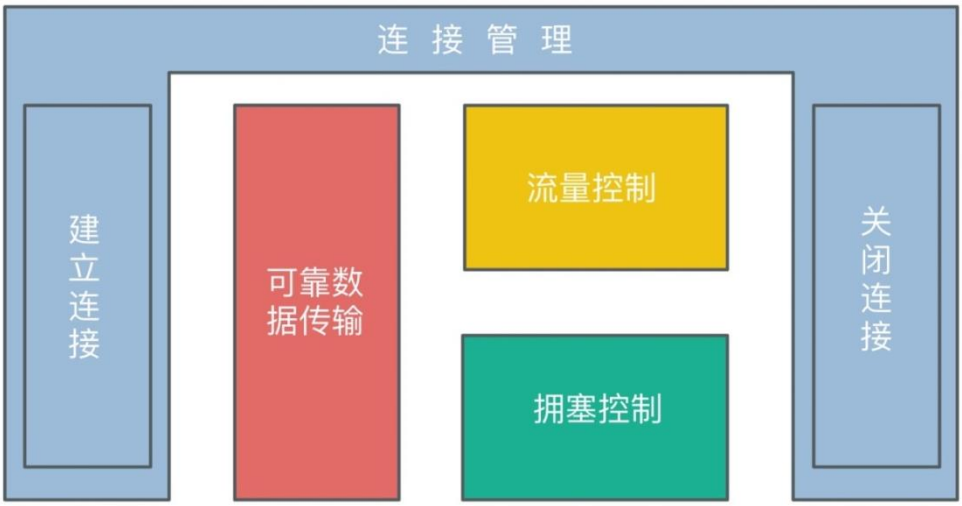
实验的目的是实现一个自己的 TCP，TCP 是一个面向连接的，能够可靠数据传输的,并且支持拥塞管理和流量控制机制的传输层协议。所以按照上述的分析，在协议设计部分将问题分为了四个模块。即连接管理模块，可靠数据传输模块，拥塞管理模块和流量控制模块。每个模块的具体设计细节见协议设计部分。

协议设计

1. 总体设计

1.1 模块关系图

根据 TCP 实现的主要功能，将 TJU_TCP 分为了如下四个功能模块：



1.2 模块功能介绍

1）连接管理模块

连接管理模块从中分为**建立连接模块**和**关闭连接模块**，这两个模块主要是来实现 TCP 连接的建立和拆除。**建立连接模块**对应的是“三次握手”的过程，**关闭连接模块**对应是“四次挥手”的过程。

2）可靠数据传输模块

TCP 的任务是在 IP 层的不可靠、尽力而为服务的基础上建立一种可靠数据传输服务。TCP 提供的可靠数据传输服务就是要保证接收方进程从缓冲区读出的字节流与发送方发出的字节流是完全一样的。TCP 使用了校验（本实验基于 UDP

来实现 TJU_TCP，并且 TJU_TCP 报头结构没有提供检验和部分，所以依靠 UDP 的差错检测机制来实现）、序号、确认和重传机制来达到这个目的。配合着流量控制和拥塞控制，使得整个传输过程得到保证。

3）流量控制模块

如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。利用滑动窗口机制可以很方便地在 TCP 连接上实现对发送方的流量控制。

4）拥塞控制模块

该模块处理网络出现拥塞的情况。该模块中的拥塞控制方法主要包括三种：慢启动、拥塞避免算法、快速恢复。通过这三个方法对应的三个函数来实现拥塞控制。

2. 数据结构设计

2.1 连接管理

在实现连接管理模块时，除了在框架代码中已经提供的数据结构外还用到了顺序队列，相关的数据设计如下：

```
// 结构和数据结构体声明
struct sock_node;
struct sock_queue;
typedef struct sock_node sock_node;
typedef struct sock_queue sock_queue;
```

```
// 队列结点的结构体定义
typedef struct sock_node{
    tju_tcp_t* sock;         //数据域 存放的是 socket
    struct sock_node* next; //指向队列的下一个节点
}sock_node;

// 队列的结构体定义(未完成队列和已完成队列)
typedef struct sock_queue{
    struct sock_node *front, *rear;
    int queue_size;
}sock_queue;
```

```
/* 以下是对 socket 队列的操作 */
sock_node* newNode(tju_tcp_t* sock);     // 创建一个队列结点
sock_queue* createQueue();               // 创建一个空队列
void enQueue(sock_queue* q, tju_tcp_t* sock);  // 入队操作
tju_tcp_t* deQueue(sock_queue* q);       // 出队操作 返回出队 socket
```

2.2 可靠数据传输

可靠数据传输实现离不开接收/发送方缓冲区和窗口。因此，接下来就给出它们的数据结构以及说明。

2.2.1 发送/接收缓冲区

注：由于篇幅原因这里仅给出了 socket 的数据结构中与缓冲区有关的部分

```
typedef struct {

    pthread_mutex_t send_lock;   // 发送缓冲区锁
    char* sending_buf;           // 发送缓冲区起始地址
    int sending_buf_send_len;    // 发送缓冲区中已发送数据的长度
    int sending_len;             // 发送缓冲区中所有数据的大小

    pthread_mutex_t recv_lock;   // 接收数据锁
    char* received_buf;          // 接收缓冲区起始地址
    int received_len;            // 接收数据缓存长度

} tju_tcp_t;
```

2.2.2 发送/接收窗口

```
typedef struct {

    uint16_t window_size;        // 发送窗口大小
    uint32_t base;               // 指向发送窗口第一个序号
    uint32_t nextseq;            // 指向发送窗口下一个可用序号
    struct itimerval timeout;    // 记录超时时间 RTO

} sender_window_t;
```

```
typedef struct {
   uint32_t expect_seq;          // 保存期待收到的报文序列号
} receiver_window_t;
```

2.3 拥塞控制和流量控制

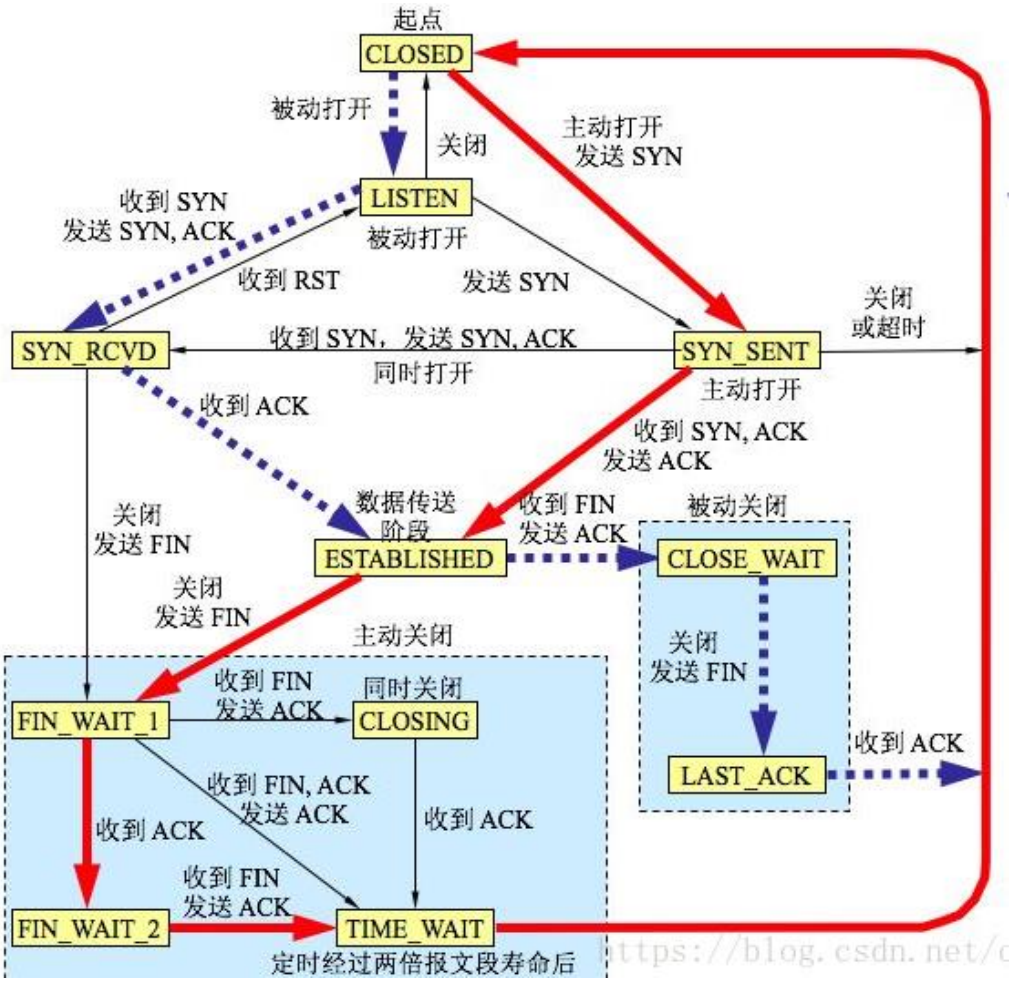
```
typedef struct {
    uint16_t window_size;
    uint32_t base;
    uint32_t nextseq;
    uint64_t estmated_rtt;          // 用于计算 timeout
    uint64_t dev_rtt;               // 用于计算 timeout
    int ack_cnt;                    // 重复 ACK 计数
```

```
    pthread_mutex_t ack_cnt_lock;   // 重复 ACK 计数的锁
    struct itimerval timeout;       // 重传时间
    uint16_t rwnd;                  // 流量控制
    int congestion_status;          // 拥塞状态
    uint16_t cwnd;                  // 拥塞窗口大小
    uint16_t ssthresh;              // 拥塞阈值
    bool is_estimating_rtt;         // 来表明是否在测量 SampleRTT
    struct timeval send_time;       // 记录发送时间
    uint32_t rtt_expect_ack;        // 用来测量 RTT 的报文期待的 ACK 号
} sender_window_t;
```

3. 协议设计规则

3.1 连接管理

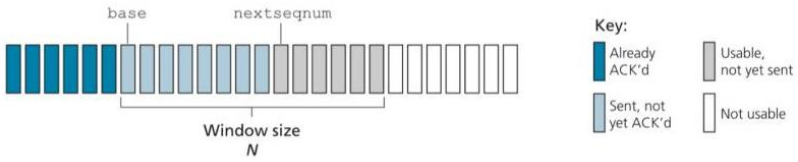
连接管理部分涉及到连接的建立和关闭。分别对应“三次握手”和“四次挥手”，该模块，该两个模块的设计涉及到 TCP 的状态的转变，如下图：



3.2 可靠数据传输

本次实验中用于实现可靠数据传输的协议是 GBN 协议。GBN 协议是停等协议的一种改进，它采用了流水线技术，允许发送方发送多个分组而无需等待确认，

同时发送方有一定的缓存能力来支持这种方式的实现。那些已经被发送但还未被确认的分组的许可序号范围可以被看成是一个在序号范围内长度为 N 的窗口。随着协议的运行，也就是收到正确的 ACKs，该窗口在序号空间向前滑动。因此 N 被称为窗口长度，GBN 协议也常被称为滑动窗口协议。下图所示为 GBN 协议中发送方看到的序号范围。



base：最早未确认分组的序号
nextseqnum：最小的未使用序号（即下一个待发分组的序号）

状态转换图

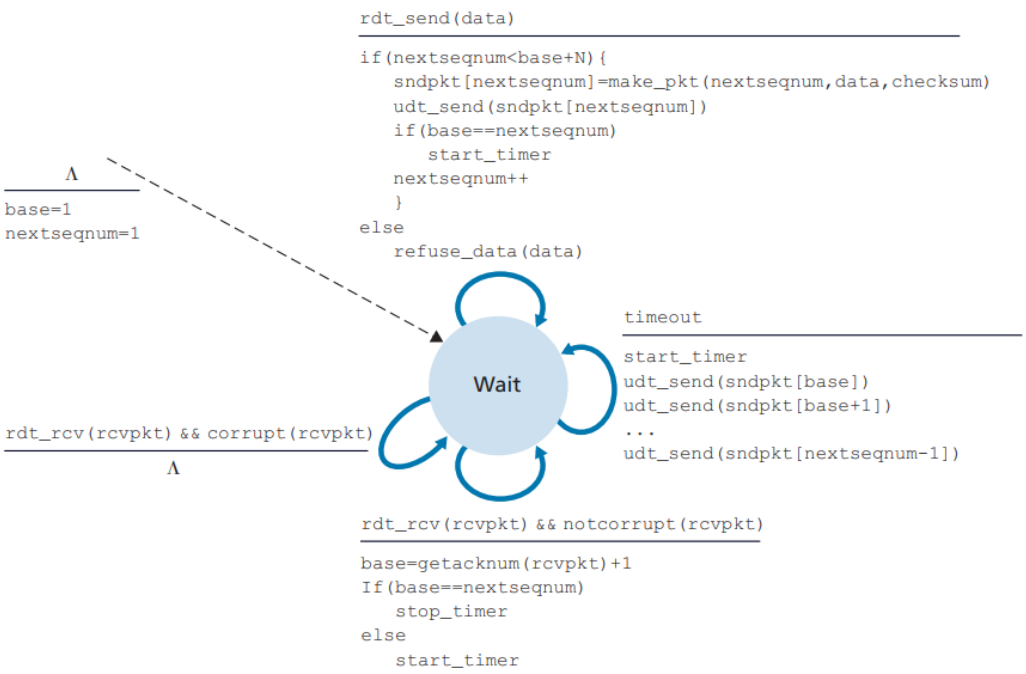


Figure 3.20 ♦ Extended FSM description of the GBN sender

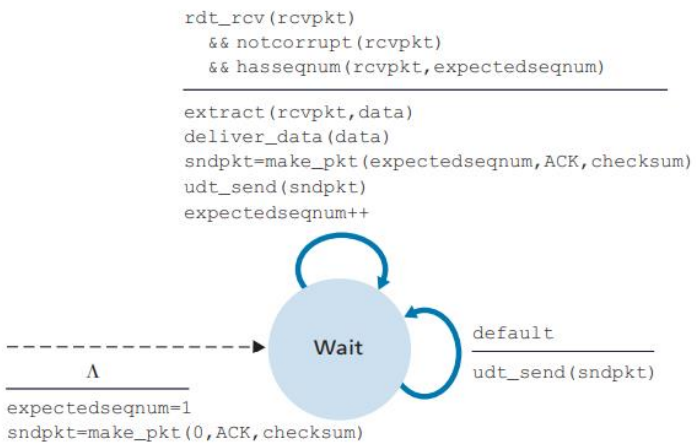
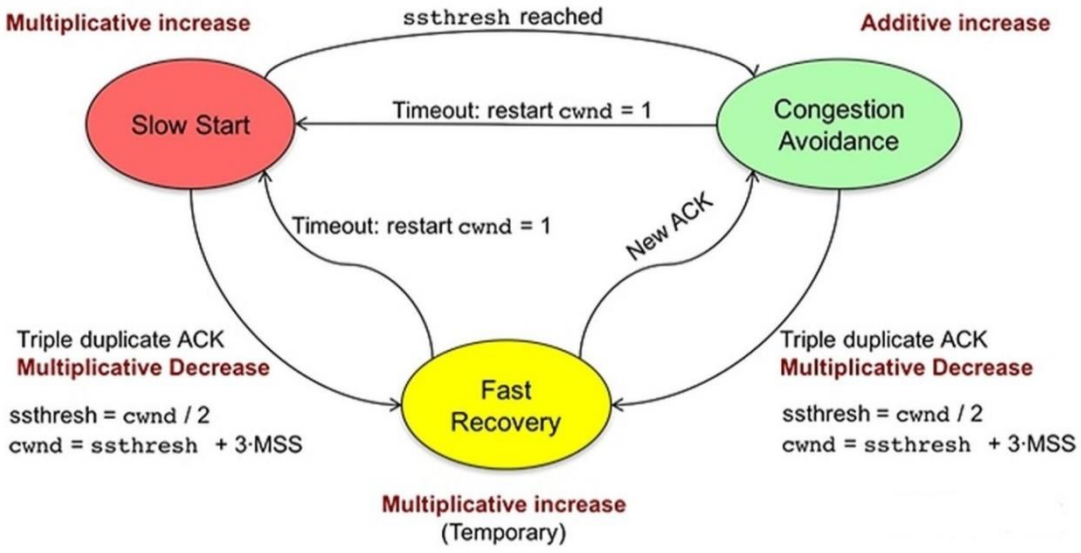


Figure 3.21 ♦ Extended FSM description of the GBN receiver

3.3 流量控制和拥塞控制

流量控制：如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。TCP 的流量控制是利用滑动窗口机制实现的，接收方在返回的数据中会包含自己的接收窗口的大小，以控制发送方的数据发送。

拥塞控制：拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。



协议实现

第一阶段：可靠数据传输

1. 实现思路

1.1 实现原理

服务器必须准备好接受外来的连接。这通过调用 tju_socket、tju_bind、tju_listen 这三个函数来实现，成为被动打开。在服务端 socket 进入监听状态后，客户端就可以发起三次握手;

（1)客户端通过调用 tju_connect 函数发起连接,向服务端发送 SYN 报文(seq 为客户端的初始序列号、SYN=1）

（2）服务端收到客户端发来的 SYN 报文后，向客户端发送 SYN_ACK 来（seq 为客户端的初始序列号 K，ack 值为 J+1、SYN=1）

（3）客户端收到 SYN_ACK 后向服务端发送 ACK 报文（ack 值为 K+1、SYN=1）

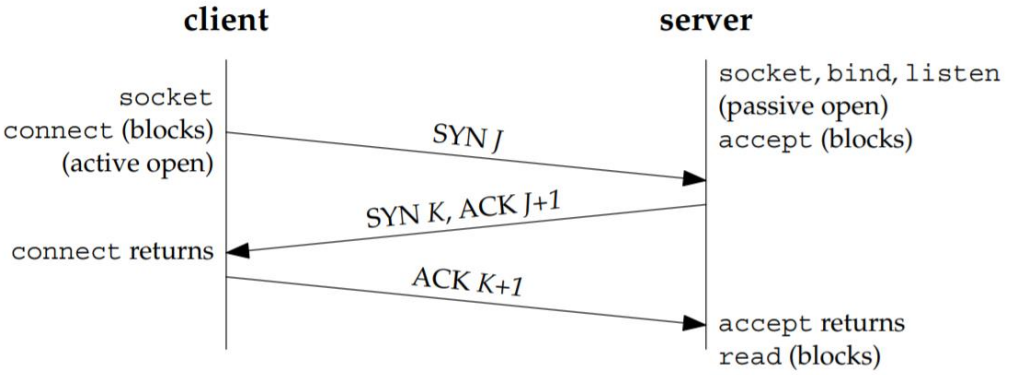


Figure 2.2 TCP three-way handshake.

1.2 各个函数的实现

在回顾一遍三次握手的过程以后就可以开始分析实现该过程涉及到的函数以及每个函数具体执行的操作。接下来，就以代码编写的顺序依次介绍每个函数实现的功能，具体实现以及各个函数间的关系。

1.2.1 tju_connect()函数

tju_connet()函数是三次握手的开始。在回顾完“三次握手”过程后，想到的该函数需要完成的操作如下：

① 将该 socket 绑定到本地地址（IP 地址 + 端口号）

② 向客户端发送 SYN 报文，并将状态改为 SYN_SENT

③ 阻塞等待，直至收到正确的 SYN_ACK

④ 收到 SYN_ACK 后向服务端发送 ACK 报文

⑤ 将 socket 的状态改为 ESTABLISH，并放入 Ehash 中

⑥ 完成三次握手，成功返回

在具体实现的时候发现，并不是以上的所有操作都能在 tju_connet()函数里完成，客户端方的三次握手需要 tju_connet()函数和 tju_handle_packet()函数（将在 1.2.2 中给出）来共同实现。接下来就单独分析每个函数的具体实现思路。

tju_connect()该函数可以实现上面提到的步骤 ①、②、③ 和 ⑥。剩下的步骤则需要在 tju_handle_packet()函数中实现。因为 tju_handle_packet()函数是对收到的报文进行处理的函数。因此，收到 SYN_ACK 报文之后的操作都可以交给该函数来完成。这样一来 tju_connet()函数的实现就变得相对简单了。

(1) 将该 socket 绑定到本地地址

这一步只需要将为该 sock 的 established_local_addr 的值改为本地地址即可（端口号可以自由分配）。

(2) 向客户端发送 SYN 报文，并将状态改为 SYN_SENT

需要先构造一个报文，这可以通过调用函数 create_packet（）来实现。需要先确定下来报文的序号，因此我在 global.h 文件中定义了两个序列号（客户端和服务端），只用在三次握手环节中：

```
#define SERVER_CONN_SEQ 0     //服务端的三次握手报文初始序号
#define CLIENT_CONN_SEQ 10    //客户端的三次握手报文初始序号
```

报文构造好以后就可以调用 sendToLayer3()函数将其发往 IP 层，然后将该 sock 的状态修改为 SYN_SENT。

(3) 将该 sock 存入 Ehash 中

按直觉来说，这一步应该放在“三次握手”完成后再实现（一开始我也是这么实现的，但是在运行的时候发现了错误，然后进行了修改），但是如果这样做客户端就没办法正确处理收到的 SYN_ACK 报文了。因为，无论是哪一方，在收到报文的时候都会先调用 onTCPPocket()函数，该函数会根据收到的报文的四元组算出一个哈希值后会从分别从 Ehash 和 Lhash 中寻找是否有对应的 socket。如果不在这一步把 sock 存入 Ehash 中，那么 onTCPPocket()函数就没有办法找到该报文对应的 socket，也更不能调用 tju_handle_packet()函数来进行下一步的处理。

因此，在发送完报文后就需要将该 sock 存入 Ehash 中，但这并不意味着已经成功建立连接了，因为虽然该 sock 在 Ehash 中能够找到，但是仍出于 SYN_SENT 状态，只有状态改为 ESTABLISH 时才意味着已经成功建立连接。

(4) 阻塞等待

阻塞的过程用 while 循环来实现了，该循环的判断条件为 sock->state != ESTABLISHED，也就是只有当完成“三次握手”后 tju_connect（）函数才会退出。而对 SYN_ACK 报文的处理以及状态的转换则在 tju_handle_packet()函数中实现。

1.2.2 tju_handle_packet() 函数

onTCPPocket()函数如果从 Ehash 或 Lhash 中找到了对应的 socket，就会调用 tju_handle_packet()函数是对收到的报文进行处理。该函数的参数是报文和收到该报文的 socket，因此就可以根据 socket 所处的状态以及收到的报文的类型来确定需要进行哪些操作。该函数实现的步骤如下：

(1) 判断收到报文的 socket 的状态是否为 SYN_SENT

如果收到报文的 socket 处于 SYN_SENT 状态,这说明目前还处于“三次握手”环节，且收到该报文的是客户端（因为根据状态转换图可知，只有客户端的 socket 才能进入 SYN_SENT 状态)。因此,可以在该函数中添加不同的 if 语句(或者 switch 语句）来实现不同状态的 socket 的不同功能。

(2) 判断收到的报文的类型

如果收到的是 SYN_ACK 报文，且 ACK 序号为客户端所期待的序号（即 SERVER_CONN_SEQ + 1)，则就开始执行 1.1.2 中提到的步骤 ④ 和 ⑤ 了。

(3) 向服务端发送 ACK 报文

根据收到报文以及收到报文的 socket 的信息来创建一个 ACK 报文。需要注意的是，该 ACK 报文的 seq 号是 SYN_ACK 报文的 ack 号，ack 号是 SYN_ACK 报文的 seq + 1。创建好报文后调用 sendToLayer3（）发送报文。

(4) 将该 socket 的状态该为 ESTABLISH

将该 socket 的状态该为 ESTABLISH 以后，一直处于阻塞状态的 tju_connect()函数就可以成功返回了。

1.2.3 tju_listen（）函数

tju_listen（）函数除了将 socket 的状态该为 LISTEN 并存入 Lhash 之外还需要完成另一件事情，为该 socket 创建两个队列，分别为**未完成连接队列**和**已完成连接队列。**在协议设计模块中已给出数据结构。

除此之外，还实现了队列上的一些基本操作的函数。队列的结构体定义好后，就在 tju_tcp_t 的结构体中加了下面两行代码：

```
sock_queue* incomplete_conn_queue;
sock_queue* complete_conn_queue;
```

在调用 tju_socket()函数创建 socket 时将这两个队列的值初始化为 NULL。在调用 tju_listen()函数的时候就要调用 createQueue()函数创建两个空队列，以便在后面的函数中使用。

1.2.4 tju_accept（）函数

tju_accept()函数是服务端完成三次握手的主要函数，在写该函数的实现之前可以先罗列出服务端在“三次握手”过程中需要完成的任务：

① 等待客户端的 SYN 报文

② 收到 SYN 报文后，向客户端返回 SYN_ACK 报文

③ 将 socket 的状态改为 SYN_SENT 并放入半连接队列中

④ 等待客户端的 ACK 报文

⑤ 收到 SYN 报文后，取出半连接队列中的 socket，将其状态改为 ESTABLISH 后，放入全连接队列中去

⑥ 若全连接队列中有 socket 取出该 socket，记录到 Ehash 中并返回

观察以上的步骤可以发现 ① 到 ⑤ 的操作都以收到报文为条件。因此，这些操作需要放到 tju_handle_packet()函数中去。如此一来，tju_accept（）函数所需要实现的操作就变得简单了。tju_accept()函数只需实现以下两个操作：

(1) 判断全连接队列中是否有 socket

该函数在全连接队列中有 socket 时才取出返回,因此在这里写了一个 while 循环，只有当队列不为空时退出循环，执行接下来的操作。

```
while(!listen_sock->complete_conn_queue->queue_size);
```

循环退出的条件转变在 tju_handle_packet（）函数中实现。

(2) 创建新的 socket，存入 Ehash 并返回

全连接队列中有 socket，意味着该 socket 已经完成了三次握手。因此，就可以创建一个新的套接字 new_conn,取出全连接队列中的 socket 赋给 new_conn，并将记录到 Ehash 中，最后返回 new_conn 即可。

1.2.5 tju_handle_packet() 函数

该函数在何种情况下会被调用在 1.2.2 中已经讨论过了，因此在这一部分就只关注该函数实现的操作。

“三次握手”过程中 tju_handle_packet() 函数在服务端被调用无非就两种情况，收到了 SYN 报文或者 ACK 报文。因此只要将这两种情况区分开来，并且把收到报文后对应的操作弄清楚，该函数的实现就会变得相对清楚。

1. 收到 SYN 报文

收到 SYN 报文后需要做 1.2.4 中提到的步骤①、②、③。在执行这些操作前需要先判断收到的是否为 SYN 报文。

(1) 判断收到报文的 socket 的状态是否为 LISTEN

如果收到报文的 socket 处于 LISTENT 状态，这说明目前还处于“三次握手”环节，且收到该报文的是服务端（因为根据状态转换图可知，只有服务端的

socket 才能进入 LISTEN 状态）。

(2) 判断收到的报文的类型

如果收到的是 SYN 报文，这说明服务端还处在“三次握手”的第一阶段，就可以开始执行步骤 ② 和 ③ 了。

(3) 将 socket 存入半连接队列中

首先，需要创建一个新的套接字 new_conn，并将监听套接字的所有信息拷贝到 new_conn 套接字，然后将该套接字的状态修改为 SYN_SENT，接着调用 enQueue（）函数将 new_conn 存入监听套接字的半连接队列中去。

(4) 向客户端发送 SYN_ACK 报文

根据收到报文以及收到报文的 socket 的信息来创建一个 SYN_ACK 报文。需要注意的是，该 SYN_ACK 报文的 seq 号是 SERVER_CONN_SEQ，ack 号是 SYN 报文的 seq + 1。创建好报文后调用 sendToLayer3（）发送报文。

2.收到 ACK 报文

收到 ACK 报文后需要做 1.2.4 中提到的步骤④ 和 ⑤。在执行这些操作前需要先判断收到的是否为 ACK 报文。

(1) 判断收到报文的 socket 的状态是否为 SYN_SENT

如果收到报文的 socket 处于 SYN_SENT 状态，这说明目前处于“三次握手”环节，且收到该报文的是服务端（因为根据状态转换图可知，只有服务端的 socket 才能进入 SYN_SENT 状态）。

(2) 判断收到的报文的类型

如果收到的是 ACK 报文，并且 ack 号是 SERVER_CONN_SEQ + 1，这说明服务端还处在“三次握手”的最后阶段，就可以开始执行步骤 ⑤ 了。

(3) 将 socket 存入全连接队列中

首先，需要将半连接队列中的 socket 取出，为其 established_local_addr 和 established_remote_addr 赋值，并将状态改为 ESTABLISH 后调用 enQueue 函数把该 socket 存入监听套接字的全连接队列中。一直处于阻塞状态的 tju_accept()就可以执行其函数体内的操作并成功返回了。

第二阶段：可靠数据传输

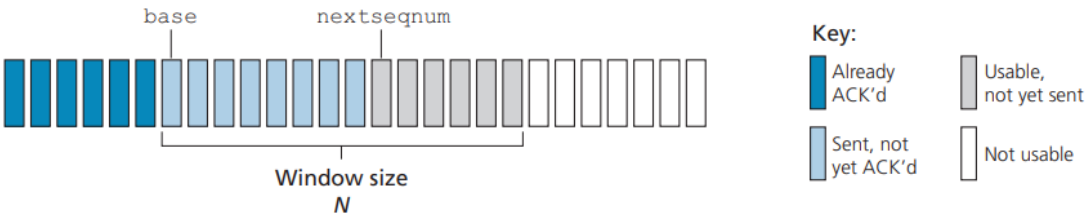
1. 实现思路

1.1 实现原理

在第二阶段要实现的是 TCP 提供的可靠数据传输机制，TCP 利用的是窗口滑动协议，SR 和 GBN 的选择权利 TCP 留给了实现者。在本实验中选择 GBN 来实现可靠数据传输。以下是 GBN 协议中的一些机制：

1）窗口

将基序号(base)定义为最早的未确认分组序号，将下一个序号(nextseqnum)定义为最小的未使用的序号（即下一个待发分组的序号），那些已经发送但未被确认的分组的许可或范围可以被看成是一个在序号范围内长度为 N 的窗口（N 被称为窗口长度），随着协议的运行，该窗口在序号空间向前滑动。



2）buffer

Buffer 用来存储从上层接收到的数据，即当触发重传机制时，发送方需要从该 buffer 中取出分组并重新发送。

1.1.1 发送方

发送方只有一个状态。下面来讨论一下发送方必须响应的三种类型的事件：

1）上层的调用：当上层调用 send()时，发送方检查窗口是否已满。如果未满，则产生分组将其发送，并相应更新变量。

2）收到一个 ACK n ：（n 表示 ACK 序号）在 GBN 协议中，采用的是累积确认，

表示接收方已经正确的收到字节 n 的以前的所有字节。将 base 设为 n，如果 base 与 nextseqnum 相同则停止计时器，否则重启计时器。

3）超时事件。如果出现超时，发送方重传所有已发送但未被确认过的分组。

1.1.2 接收方

接收方也只有一个状态。下面来具体讨论一下接收方的动作方。当接收方接收到序号为 seq 的分组，且 seq 等于接收方期待的序列号，则发送一个 ACK seq + data_len（表示 ack 号），来表明在此之前的所有字节都以按序收到。其它情况下，接收方丢弃该分组，并且发送 ACK expected。

1.2 各个函数的实现

在回顾一遍 GBN 中接收方和发送方中可能出现的事件以及相应的操作以后，就可以开始分析实现该过程涉及到的函数以及每个函数具体执行的操作。接下来，就以写代码的顺序依次介绍每个函数实现的功能，具体实现以及各个函数间的关系。最后，会给出各个函数间的关系图。

1.2.1 tju_buffered()函数

GBN 的实现主要关系到连个 API，分别为 tju_sned（）和 tju_recv（）函数。在查阅书籍和相关的资料后发现，这两个函数直接关联到的只是发送缓冲区和接收缓冲区，并不会参与到可靠数据传输实现的部分。因此，当发送方调用 tju_send（）函数时，该函数只是将应用层传过来的数据存到发送缓冲区后返回即可。数据缓存的实现正是通过 tju_buffered（）函数来实现的。函数原型：

```
int  tju_buffered(tju_tcp_t *sock, char *data, int len);
```

该函数实现的步骤如下：

① 先判断发送缓冲区是否已满，如果已满阻塞等待

② 如果发送缓冲区为空，用 malloc 函数开辟新的空间

③ 如果发送缓冲区不为空，用 realloc 函数重新分配空间

④ 改变发送缓冲区的大小 sending_len

1.2.2 sending_thread() 函数

tju_buffered（）函数返回以后 tju_send()也就会随之返回。应用层交付给传输层的数据就会被保存在发送缓冲区中。这时，就需要另外一个函数来实现，从发送缓冲区取出数据并根据发送窗口的状况为其分配序列号，再将该报文发送带下一层的工作。这些功能都将在 sending_thread()函数中实现。

当三次握手成功返回后，就要保证数据可以可靠传输。因此，tju_connect（）和 tju_accept（）函数在返回前就会创建一个线程来运行 sending_thread()函数。这样一来，该函数的运行就不会干扰到其它函数的继续执行。函数原型：

在确定好在哪里，以什么样的形式调用该函数后，就可以考虑该函数的参数以及内部实现的功能了。

```
void *sending_thread(void *arg);
```

1.2.2.1 参数

由于该函数在线程中执行，因此参数的形式和个数就比较受限。实现从缓冲区发送报文这个功能，只需要知道关于该 socket 的信息即可。因此，可以传一个指向该 socket 的哈希值的指针，在 sending_thread()中解引用，并用该 hashval 在 establish 哈希表中寻找，就能得到指向该 socket 的指针。

1.2.2.2 可靠数据传输

首先，在发送数据之前需要先判断能不能发送数据。判断条件为两个：① 发送缓存中有还没有被发送过的数据 ② 发送窗口未满。只有当着两个条件同时满

足就能可以开始发送数据。发送缓冲区中待发送的数据和发送窗口中可用的序列号数目都会影响到数据发送的多少，因此，分了两种情况来解决。

1. 需发送的数据 小于或等于 发送窗口剩余的大小

在这种情况下，就可以放心的将发送缓冲区中所有的未发送过数据都发送出去，不用担心窗口越界的情况。发送数据的多少只取决于发送缓冲区中待发送的数据的多少。实现步骤如下：

(1) 如果数据可以装满一个报文

待发送的数据足够大，就可以先将它们打包成满报文发出去，最后剩余的部分再单独发送。可以用一个 while 循环来实现该过程 while (buf_len - buf_send_len > MAX_DLEN)，如果能进入该循环，意味着发送的报文的 data_len 部分为 MAX_DLEN。进入循环后实现如下操作：

① 报文的序列号设为发送窗口的 nextseq，

② 从发送缓冲区中取出未被发送过的数据，（可以通过 sock->sending_buf + buf_send_len 来得到），用序列号和其它信息（标志位为 NO_FLAG），创建一个报文并发送

③ 如果发送窗口的 base 和 nextseq 相同就起动计时器

GBN 协议中只用到了一个计时器，只为发送窗口中的第一个序号计时（当 base == nextseq 就意味着这是发送窗口的第一个字节）。计时器相关的函数会在后面说明。

④ 更新发送窗口的 nextseq 和发送缓冲区的 sending_buf_send_len

(2) 剩余的待发送数据小于 MAX_DLEN

退出循环或者没有进入循环都意味着发送窗口中未发送过的数据小于 MAX_DLEN，发送的过程与(1)中提到的相同，唯一不同的是 data_len 的大小是发送缓冲区中剩余数据的大小。

2. 需发送的数据 大于 发送窗口剩余的大小

在这种情况下，发送的过程与第一种情况中的一样，只是发送数据的多少取决于发送窗口中剩余的可用序列号的多少。

1.2.3 startTimer（）/stopTimer（）函数

在 sending_thread() 函数中用到了计时器，在接收到 ACK 报文时也会有关于计时器的操作。计时器的启动和关闭由 startTimer（）和 stopTimer（）两个函数来实现。

1.2.3.1 startTimer（）

通过查询资料知道，Linux C 编程中提供的库函数 setitimer（）可以实现这个功能，该函数中最重要的参数是 itimerval 类型的，类型的结构体如下：

```
struct itimerval{
        struct  timerval it_interval;  // 计时器触发周期
        struct  timerval it_value;     // 计时器触发时间
};

struct timerval{
        long tv_sec;            // 秒
        long tv_usec;           // 微秒
};
```

知道了各个部分的含义后就可以为其赋值，设置计时器了。计时器在经过 timeout 时间后仍为收到报文，则说明计时器超时，而 timeout 这个值的估算在 TimeoutInterval（）函数中实现（1.3.6 中给出）。计时器函数 setitimer（）超时后，会调用一个回调函数来处理超时后对应的事件。但是，该函数的格式受限，传入的参数只能是 setitimer（）返回的信号量。因此，定义了一个 int 型的全局变量 RETRANS，在定义的回调函数 timeout_handler（）中只需要将这个变量的值改为 1，这个变化会使 **retrans_thread（）**函数开始重传报文（该函数的说明会在 1.3.4 中给出）。函数原型如下：

```
                void startTimer(tju_tcp_t *sock);
```

1.2.3.2 stopTimer（）

该函数实现还是基于 setitimer（）函数来实现。setitimer（）的特点是，一个程序中只会有一个计时器，当该函数被第二次调用时，一开始的计时器会相当与重启。就利用了这个特点来实现了 stopTimer（）函数。stopTimer（）只需要再次调用 setitimer（）函数，并将其传入的参数中计时器触发时间值该为零即可。函数原型如下：

```
void stopTimer(void)
```

1.2.4 retrans_thread（）函数

在 GBN 中，当计时器超时时就要重传发送窗口中从 base 到 nextseq 的所有字节，这个功能就在 retrans_thread（）函数中实现。该函数会在单独的线程中被调用，该线程与发送线程一样，在 tju_connect（）和 tju_accept（）函数中创建，但是只有当 RETRANS 变量的值变为 1 的时候，才会进行重传。函数原型如下:

```
void *retrans_thread(void *arg)
```

该函数中重传是将窗口中从 base 到 nextseq 的所有字节封装成报文发送出去，因此该部分的代码实现与 sending_thread（）中的类似。重传报文以后就需要重新启动计时器，还是为第一个还没收到 ACK 的序号计时。在最后只需将 RETRANS 的值改为 0。

1.2.5 tju_handle_packet（）函数

前面已经实现了发送以及重传等功能，接下来就要考虑接收方收到数据报文时会有什么样的动作，发送收到 ACK 报文时又会如何处理。这些都与收到报文这个动作有关，因此它们实现就会在 tju_handle_packet（）函数中。

1.2.5.1 接收方

当收到报文的socket处于ESTABLISH状态且收到的报文的标志位为NO_FLAG时，就表明该报文是用来数据传输的报文，且是接收方。在GBN协议中，接收方的接收窗口结构相对简单，只有一个expect_seq来记录期待收到的报文的序列号。主要实现以下操作：

（1）收到的报文的序列号等于所期待的序列号

① 如果接受缓存已满（或者报文的data_len大于缓冲区剩余的大小）丢弃报文并返回

② 如果缓冲区未满，将报文的数据部分存到接收缓冲区中

③ 更新接收缓存的长度和接收窗口的expect_seq。expect_seq等于收到的报文的序列号加上报文的data_len

④ 构造ACK报文并发送。报文中ack号等于expect_seq，表明在此之前的字节都以按序收到。

（2）收到的报文的序列号不是所期待的序列号

① 直接丢弃报文，因为不是按序到达

② 构造ACK报文并发送。报文中ack号仍等于expect_seq

1.2.5.1 发送方

当收到报文的socket处于ESTABLISH状态且收到的报文的标志位为ACK_FLAG时，就表明该报文是用来数据报文的ACK报文，且是发送方收到。发送在收到ACK报文时主要实现以下操作：

① 如果收到的报文的ack号小于或等于base则直接丢弃，因为base之前的序列号表示的都是已发送且已收到ACK的字节。如果不是继续进行如下操作。

② 用收到的报文的ack减去base来得到发送缓冲区中需要释放的空间的大小free_len。

③ 更新接发送窗口的base，令base等于ack。

④ 如果更新后的base等于nextseq，表明发送窗口中没有使用中的序列号，

因此只需调用 stopTimer（）函数来关闭原先的计时器。

④ 如果更新后的 base 不等于 nextseq，说明 base 指向一个已发送但未收到 ACK 的序列号，因此需要为其启动计时器。需要先调用 startTimer（）再调用 stopTimer（）。

⑤ 利用 ② 中算出来的 free_len 来释放发送缓冲区中已被确认过的数据。

1.2.6 TimeoutInterval（）函数

在 1.3.3 中介绍了有关计时器实现的函数，在计时器的实现过程中有一个重要的值是重传超时间隔，该值的计算公式如下：

$$EstimatedRTT = (1 - \alpha) \cdot EstimatedRTT + \alpha \cdot SampleRTT$$

$$EstimatedRTT = 0.875 \cdot EstimatedRTT + 0.125 \cdot SampleRTT$$

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot | SampleRTT - EstimatedRTT |$$

$$TimeoutInterval = EstimatedRTT + 4 \cdot DevRTT$$

可以看出来，为了计算重传超时间隔，就要先测出 SampleRTT。我尝试测量的 RTT 是从发出一个报文到收到该报文相应的 ACK 报文所花费的时间。实现的过程入下：

1）sending_thread（）函数

测量 RTT 的思路是，在发送一个报文的时候记录下该报文的发送时间，在收到该报文的 ACK 报文时再记录一次时间，两个时间计算差值就能得到 SampleRTT。那第一步要实现的就是，记录发送报文的时间，这就需要在 **sending_thread()函数**中实现。该过程涉及到发送窗口中的三个变量：

```
bool is_estimating_rtt;     // 来表明是否在测量 SampleRTT
struct timeval send_time;   // 记录发送时间
uint32_t rtt_expect_ack;    // 用来测量 RTT 的报文期待的 ACK 号
```

在 sending_thread()函数中只有在 is_estimating_rtt == false 时(目前没有在测量 RTT 时)才会为发送的报文记录时间，调用库函数 gettimeofday()把发送时间记录在 send_time 变量中，来开始测量 RTT。因为，如果不设值这样一个变量，每一次发送报文的时候 send_time 值就会被更新一次，导致不能准确测量 RTT。这样一来，sending_thread()函数中实现功能如下：

① 判断 is_estimating_rtt == false 语句是否为真，如果为真，执行以下操作

② is_estimating_rtt 的值设为 true

③ 调用 gettimeofday(&sock->window.wnd_send->send_time, NULL)来记录发送时间

④ 令 rtt_expect_ack 等于 seq + len（其中 len 为报文的长度)。来记录该报文所期待的 ACK 的值

2）tju_handle_packet（）函数

在收到 ACK 报文，并且该 ACK 报文时可用于更新的 ACK 报文时，就要判断该 ACK 报文是否为测量 RTT 所需的 ACK 报文。实现过程如下：

① 判断 is_estimating_rtt 是否为真，如果为真说明目前已经为一个已发送的报文记录了发送时间，执行如下步骤。

② 判断报文的 ack 序号是否等于 rtt_expect_ack，如果等于说明收到了可以用于测量 RTT 的 ACK 报文。调用 TimeoutInterval()函数来更新重传时间间隔。

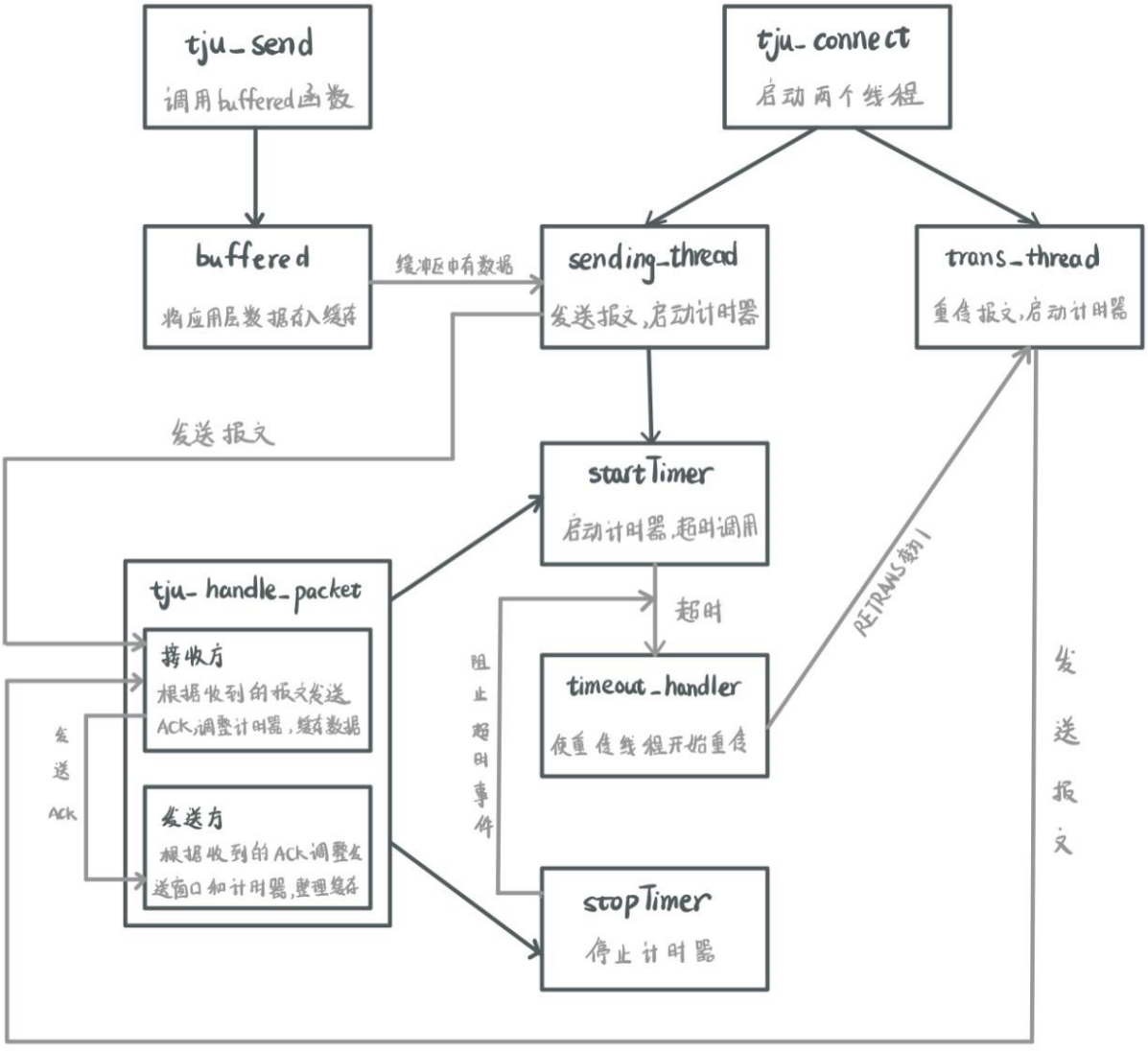
③ 无论受到的 ACK 报文的序列号是否为 rtt_expect_ack，都要将 is_estimating_rtt 改为 false。因为，如果收到的是期待的 ACK 报文，调用 TimeoutInterval()函数后就已经完成了一次对 timeout 的更新，可以把状态置为 is_estimating_rtt = false；如果收到的不是期待的 ACK 报文，说明用于测量 RTT 的报文丢失了,因此没有必要再继续等待,可以直接将 is_estimating_rtt 设为 false。

2）TimeoutInterval（）函数

在收到期待的 ACK 报文后，该函数就会被调用，用来更新重传时间间隔。该函数实现的功能相对简单。只需要将发送时间与现在的时间做差，求出 SampleRTT 后，根据前面给到的公式计算重传时间间隔即可。

1.3 函数关系图

在下图中，**深灰色**线表示函数调用，**浅灰色**线表示函数间的相互影响。每个框内简要的说明了函数功能。



第三阶段：连接管理之“四次挥手”

1. 实现思路

1.1 实现原理

1.1.1 双方先后关闭

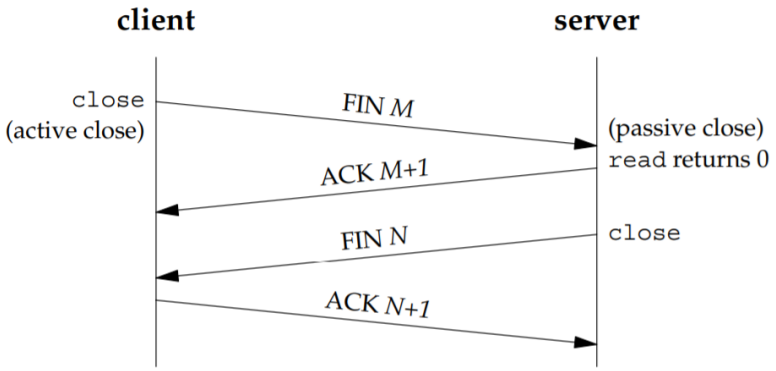
TCP 的连接过程需要经过“三次握手”，相应的关闭连接的过程就需要“四次挥手”。在分析该过程中，将调用 tju_close 执行主动关闭的一方称为客户端，被动关闭的一方称为服务端。双方先后关闭时的过程如下：

1）客户端调用 tju_close（），执行主动关闭，该端发送 FIN 报文

2）服务端接收到 FIN 报文，发送一个 ACK 报文作为回复

3）一段时间后，接收到 FIN 的一端调用 tju_close(),并发送 FIN 报文

4）客户端收到 FIN，返回 ACK 报文作为回复，并等待一段时间后释放资源



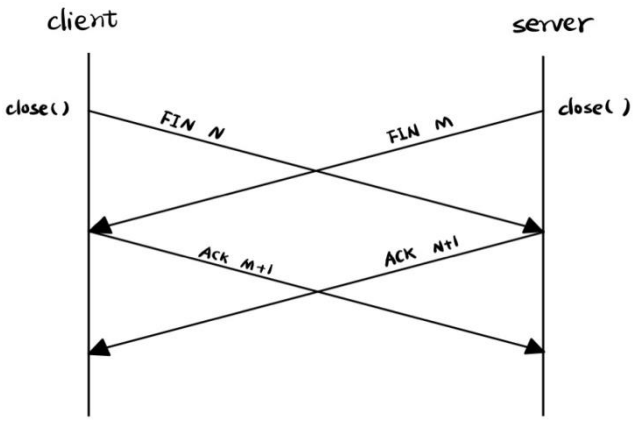
1.1.2 双方同时关闭

TCP 的连接过程需要经过“三次握手”，相应的关闭连接的过程就需要“四次挥手”。在分析该过程中，将调用 tju_close 执行主动关闭的一方称为客户端，被动关闭的一方称为服务端。双方先后关闭时的过程如下：

1）两端都调用 tju_close（），执行主动关闭，发送 FIN 报文

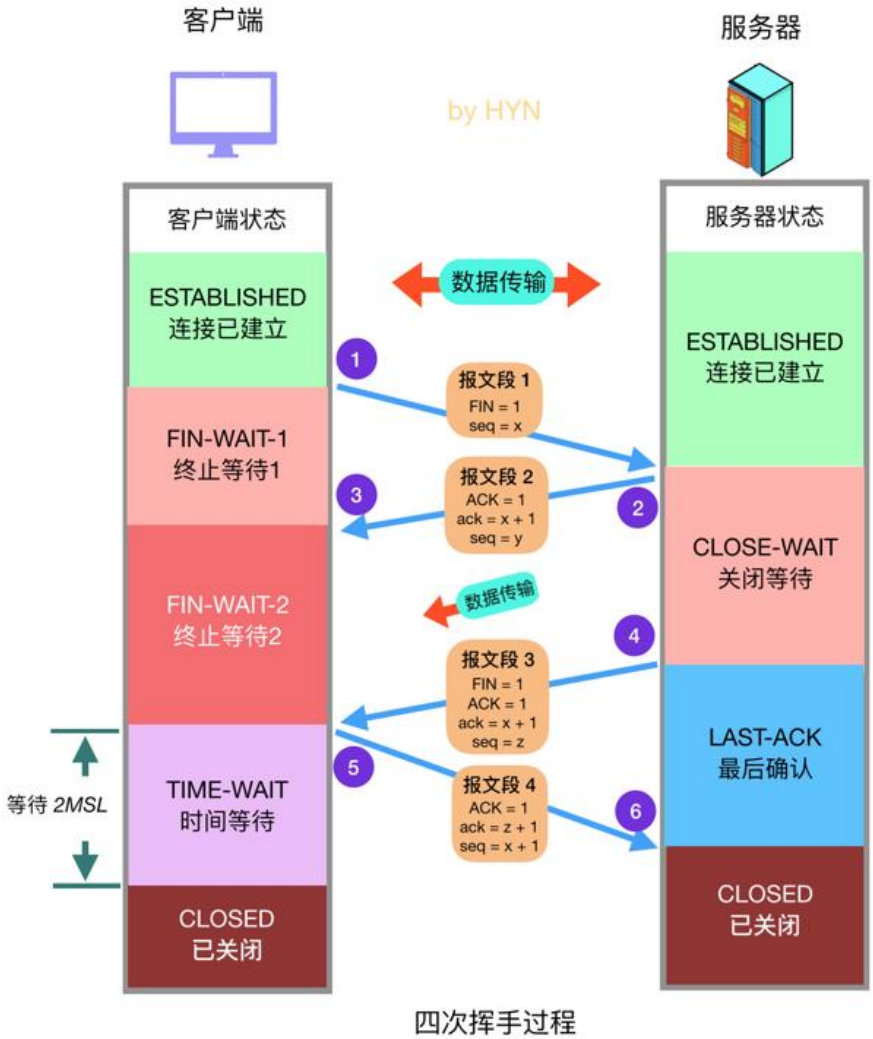
2）两端同时接收到 FIN 报文，发送 ACK 报文作为回复

3）两端都接收到 ACK 报文，并等待一段时间后释放资源



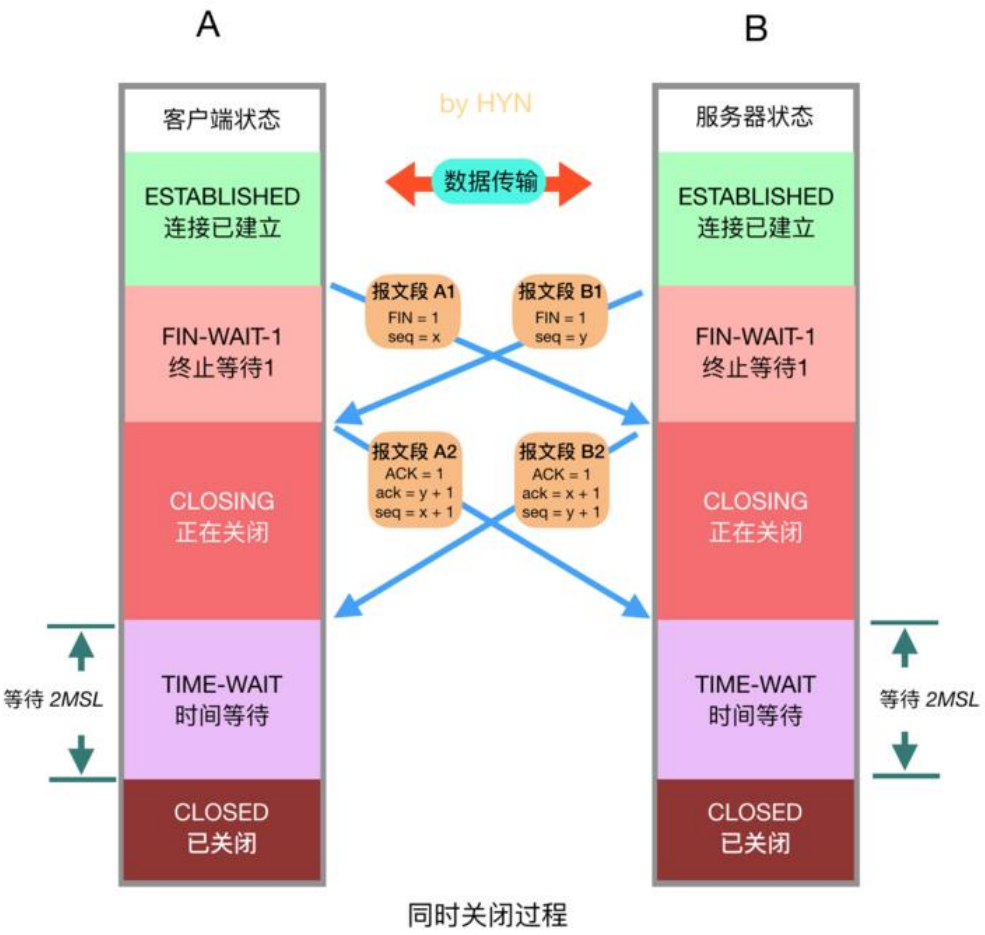
1.1.3 状态改变

1）双方先后关闭



四次挥手过程

2）双方同时关闭



同时关闭过程

1.2 各个函数的实现

在实现 TJU_TCP 的关闭连接功能时需要注意的一点是，两方先后关闭和同时关闭的时候，socket 的状态转变是不一样的。因此，在实现该部分的代码时，先考虑了双方先后关闭的情况，然后再此基础上补充了同时关闭时的处理。

1.2.1 tju_close()函数

tju_close()函数是四次挥手的开始。该函数只有在“四次挥手”完成的时候才会返回。tju_close()函数会在两个情况下被调用：

① 客户端主动调用（对应 1.1.3 图中的 1 过程）

② 服务端发送完响应客户端 FIN 的 ACK 报文后被调用（对应 1.1.3 图中的 4 过程）

可以根据图片发现，在这两种情况下，tju_close（）函数都会发送 FIN 报文，但是发送完 FIN 报文后的状态转变有所不同。在 ① 这种情况下，socket 的状态在发送完 FIN 报文后改变为 FIN_WAIT_1；在 ② 情况下，socket 的状态在变为 LAST_ACK。因此，tju_socket（）所要执行的操作如下：

① 发送 FIN 报文

② 如果当前状态为 ESTABLISH,则把状态改为 FIN_WAIT_1（表明主动调用）

③ 如果当前状态为 CLOSE_WAIT，则为 TIME_WAIT（表明被动调用）

④ 在 socket 的状态改变为 CLOSED 之前都阻塞等待（该状态的改变在 tju_handle_packet 函数中实现）

⑤ 释放资源后返回

1.2.2 tju_handle_packet() 函数

无论是哪一方，在收到有关“四次挥手”的报文时所做出的操作都在该函数中实现。接下来，就按照“四次挥手”的过程来梳理 tju_handle_packet() 函数所要进行的操作。先完成先后关闭的实现，后补充同时关闭的情况。

(1) 收到 FIN 报文（服务端）

如果收到报文的 socket 处于 ESTABLISH 状态,这说客户端收到了 FIN 报文，且收到该报文的是客户端。在收到报文后需要做如下操作：

① 发送 ACK 报文

② 将 socket 的状态改为 CLOSE_WAIT

③ 等待一秒后，调用 tju_close 函数（这时 tju_close 函数被调用后会发送 FIN 报文，由于目前处在 CLOSE_WAIT 状态，tju_close 函数会将 socket 的状态转变为 LAST_ACK）

(2) 收到 ACK 报文（客户端）

如果收到的是 ACK 报文，收到报文的 socket 处于 FIN_WAIT_1 状态，说明收到报文的是客户端，且处于“四次挥手”的第二阶段。这时候只需要将 socket 的状态改为 FIN_WAIT_2 即可。

(3) 收到 FIN 报文（客户端）

在（1）中提到了，当服务端收到 FIN 并返回 ACK 报文后会调用 tju_close 函数，该函数会发送 FIN 报文。当收到报文的 socket 处于 FIN_WAIT_2 状态并且该报文为 FIN 报文，则说明目前处于“四次挥手”的第三阶段。这时需要发送一个 ACK 报文，socket 状态改为 TIME_WAIT,并等待 2MSL 长的时间后将状态改为 CLOSED。这时，客户端阻塞着的 tju_close（）函数会继续往下执行，完成资源释放。

(4) 收到 ACK 报文（服务端）

如果收到的是 ACK 报文，收到报文的 socket 处于 LAST_ACK 状态，说明收到报文的是客户端，且处于“四次挥手”的最后阶段。只需要将 socket 的状态改为 CLOSED 即可。服务端的 tju_close（）成功返回后就说明 “四次挥手”过程的成功。

(5) 收到 FIN 报文（双端）

前面（1）到（4）对应着双方先后关闭的状态的情况。接下来就要分析同时关闭时的状况。所谓的同时关闭，意味着双方都主动调用了 tju_close（）函数，这时双方都会发送 FIN 报文并进入 FIN_WAIT_1 状态。因此，在 socket 的状态为 FIN_WAIT_1 时收到 FIN 报文意味着这时双方同时关闭的情况。这时就需要发送 ACK 报文并将 socket 的状态改为 CLOSING。

(6) 收到 ACK 报文（双端）

如果收到报文的 socket 处于 CLOSING 并且是 ACK 报文，说明这是双方同时关闭的情况，因为只有在这种情况下才会进入 CLOSING 状态。这时只需要将 socket 状态改为 TIME_WAIT,并等待 2MSL 长的时间后将状态改为 CLOSED 即可。

第四阶段：拥塞管理

1.实现思路

1.1 实现原理

为实现拥塞控制的机制，发送方维持一个拥塞窗口 cwnd （ congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些,以便把更多的分组发送出去。但只要网络出现拥塞,拥塞窗口就减小一些，以减少注入到网络中的分组数。窗口的调整细节通过慢启动，拥塞避免，快速恢复三个算法实现。

1.1.1 慢启动算法

当主机开始发送数据时，如果立即所大量数据字节注入到网络，那么就有可能引起网络拥塞，因为现在并不清楚网络的负荷情况。因此，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是说当一条 TCP 连接开始时处于慢启动状态，cwnd 的值以 1 个 MSS 开始并且每当传输的报文段首次被确认就增加一个 MSS，并发送出两个最大长度的报文段，这两个报文段被确认，使得拥塞窗口变为 4 个 MSS，以此类推。除了指数增长的情况，慢启动状态的改变还有以下三种情况：

① 当发生超时，TCP 将 cwnd 设置为 1 并重新开始慢启动过程，同时将 ssthresh（慢启动阀值）设置为 cwnd/2。

② 当 cwnd 达到 ssthresh 值时，进入拥塞避免状态

③ 当检测到 3 个冗余的 ACK，TCP 执行快速重传并进入快速恢复状态。

1.1.2 拥塞避免算法

当进入拥塞避免状态时，cwnd 的值是上次拥塞时的值的一半并且 TCP 更加

谨慎地增加 cwnd。也就是说，当 TCP 发生方收到一个新确认（这里并不像慢启动算法一样对每一个收到的 ACK 增加一个 MSS），就将 cwnd 增加一个 MSS（MSS/cwnd）字节。该状态下 cwnd 的改变与慢启动算法第① ③情况相同。需要注意的一点是，当发生超时事件时 TCP 将回到慢启动状态。

1.1.3 快速恢复算法

快速恢复是对丢失恢复机制的改进。当进入快速恢复状态时，置 sshtesrh=cwnd/2、cwnd=ssthresh+3MSS，对每一个冗余的 ACK，cwnd 的值增加一个 MSS。状态的转变：

① 当丢失报文段的一个 ACK 到达，TCP 降低 cwnd 后进入拥塞避免状态

② 出现超时事件时，执行与慢启动和拥塞避免中相同的动作后进入慢启动状态。

TCP 拥塞控制机制的状态转换图：

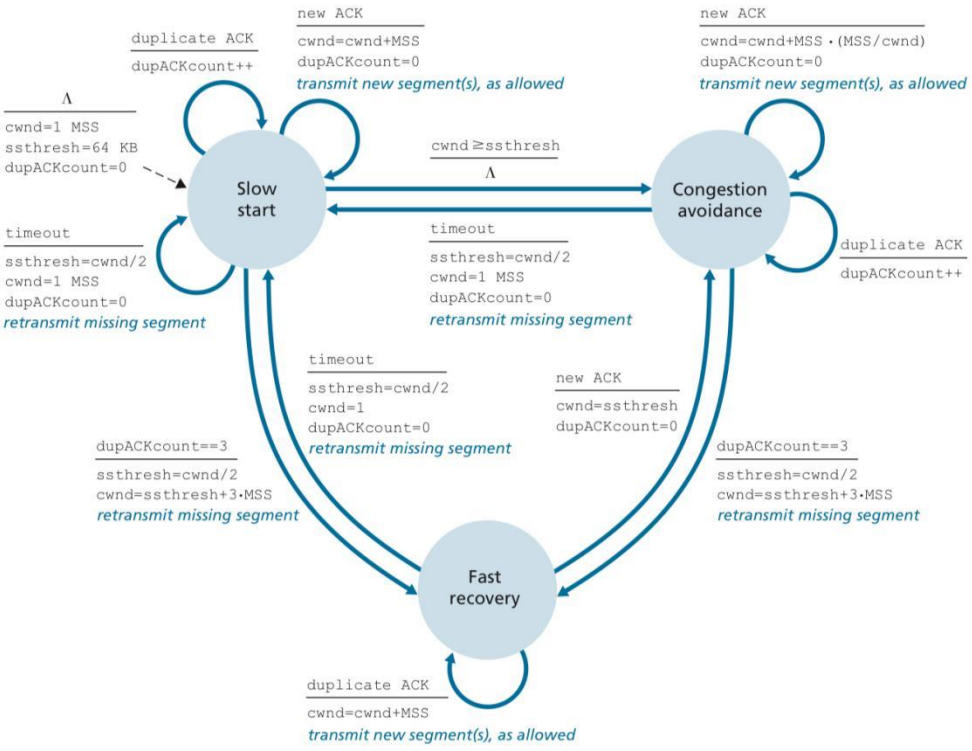


Figure 3.51 ♦ FSM description of TCP congestion control

1.2 各个函数的实现

1.2.1 tju_socket()函数

初始化套接字时将 TCP 状态初始化为慢启动状态，所以刚开始接收窗口按照慢启动算法去调整。cwnd 初始值为 1 个 MSS，ssthresh 初始值为 10 个 MSS。

```
sock->window.wnd_send->cwnd = MAX_DLEN;
sock->window.wnd_send->congestion_status = SLOW_START;
sock->window.wnd_send->ssthresh = 10 * MAX_DLEN;
```

1.2.2 sending_thread()函数

为达到拥塞控制和流量控制的目的需要保证已发送但未被确认的的数据量不会超过 cwnd 与 rwnd 中的最小值。因此在发送函数中加入了以下条件。

```
while (buf_len - buf_send_len > MAX_DLEN && wnd_next + MAX_DLEN - wnd_base <= min(cwnd, rwnd))
{
```

1.2.3 retrans_thread()函数

状态转换图中的 timeout 事件对应于 retrans_thread 函数中的如下所示部分（不管在何种状态都做相同的操作）：

```
if (TIMEOUT_FLAG)
{
    sock->window.wnd_send->ssthresh = sock->window.wnd_send->cwnd / 2;
    sock->window.wnd_send->cwnd = MAX_DLEN;
    while (pthread_mutex_lock(&(sock->window.wnd_send->ack_cnt_lock)) != 0);
    sock->window.wnd_send->ack_cnt = 0;
    pthread_mutex_unlock(&(sock->window.wnd_send->ack_cnt_lock));
    sock->window.wnd_send->congestion_status = SLOW_START;
}
```

1.2.4 tju_handle_packet()函数

该函数中更新的内容主要包括了对冗余的 ACK 和新 ACK 的处理方式及拥塞控制状态的改变。如算法原理中提到的一样，慢启动和拥塞避免状态下对冗余的 ACK 计数，当达到 3 时 TCP 执行快速重传，并置 sshtesrh=cwnd/2、cwnd=ssthresh+3MSS 并进入快速恢复状态；而在快速恢复状态下只是增加 cwnd，代码实现如下：

```
if (sock->window.wnd_send->congestion_status == SLOW_START || sock->window.wnd_send->congestion_status == CONGESTION_AVOIDANCE)
{
    while (pthread_mutex_lock(&(sock->window.wnd_send->ack_cnt_lock)) != 0)
        ;
    sock->window.wnd_send->ack_cnt += 1;
    pthread_mutex_unlock(&(sock->window.wnd_send->ack_cnt_lock));
}

if (sock->window.wnd_send->congestion_status == FAST_RECOVERY)
{
    sock->window.wnd_send->cwnd += MAX_DLEN;
}

if (sock->window.wnd_send->ack_cnt == 3 && sock->window.wnd_send->congestion_status != FAST_RECOVERY)
{
    sock->is_retransing = true;
    sock->window.wnd_send->ssthresh = sock->window.wnd_send->rwnd / 2;
    sock->window.wnd_send->cwnd = sock->window.wnd_send->ssthresh + 3 * MAX_DLEN;
    sock->window.wnd_send->congestion_status = FAST_RECOVERY;
    printf("收到3个重复ACK 开始快速重传\n");
    RETRANS = 1;
}
```

当收到可用于更新的 ACK 时，各状态的处理方式不相同，已在算法原理部分说明。代码实现如下：

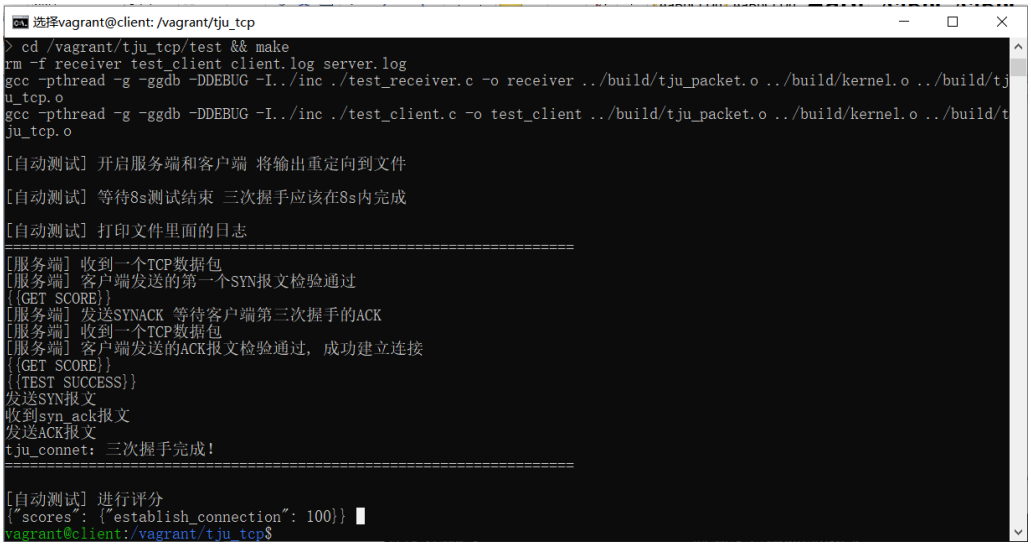
```
if (sock->window.wnd_send->congestion_status == SLOW_START)
{
    sock->window.wnd_send->cwnd += MAX_DLEN;
    if (sock->window.wnd_send->cwnd >= sock->window.wnd_send->ssthresh)
    {
        sock->window.wnd_send->congestion_status = CONGESTION_AVOIDANCE;
    }
}
else if (sock->window.wnd_send->congestion_status == CONGESTION_AVOIDANCE)
{
    sock->window.wnd_send->cwnd = sock->window.wnd_send->cwnd + MAX_DLEN * (MAX_DLEN / sock->window.wnd_send->cwnd);
}
else if (sock->window.wnd_send->congestion_status == FAST_RECOVERY)
{
    sock->window.wnd_send->cwnd = sock->window.wnd_send->ssthresh;
    sock->window.wnd_send->congestion_status = CONGESTION_AVOIDANCE;
}
```

实验结果及分析

1. 实验结果

1.1　连接管理之“三次握手”

1）使用 test 文件测试

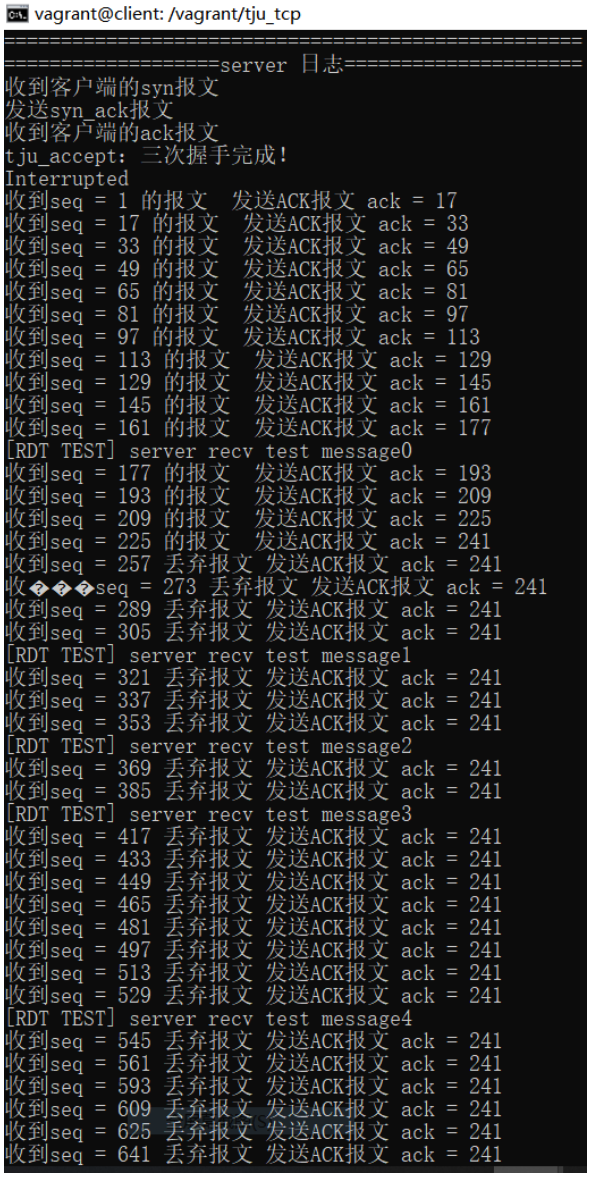


2）单独运行

由该截图可以看出“三次握手”过程成功实现了，并且建立连接后可以调用 tju_send（）和 tju_recv（）函数收发信息。

1.2 可靠数据传输

实现可靠数据传输的时候，发送方调用 tju_send 函数时应用层数据会先被存在发送缓冲区中。sending_thread 函数会一直循环，只要发送缓冲区中有未被发送过的数据且发送窗口有可用序列号时就发送。但因为没法预算计算及内部程序运行的速度，有些时候会出现 50 条信息全被装进一个报文里发送的情况。这种情况下，比较难体现本次实现的可靠数据传输。因此，我进行多次测试，截取了消息分多个报文发送的情况。




vagrant@client: /vagrant/tju_tcp
[RDT TEST] server recv test message19
[RDT TEST] server recv test message20
[RDT TEST] server recv test message21
[RDT TEST] server recv test message22
[RDT TEST] server recv test message23
[RDT TEST] server recv test message24
[RDT TEST] server recv test message25
[RDT TEST] server recv test message26
[RDT TEST] server recv test message27
[RDT TEST] server recv test message28
[RDT TEST] server recv test message29
[RDT TEST] server recv test message30
[RDT TEST] server recv test message31
[RDT TEST] server recv test message32
[RDT TEST] server recv test message33
[RDT TEST] server recv test message34
[RDT TEST] server recv test message35
[RDT TEST] server recv test message36
[RDT TEST] server recv test message37
[RDT TEST] server recv test message38
[RDT TEST] server recv test message39
[RDT TEST] server recv test message40
[RDT TEST] server recv test message41
[RDT TEST] server recv test message42
[RDT TEST] server recv test message43
[RDT TEST] server recv test message44
[RDT TEST] server recv test message45
[RDT TEST] server recv test message46
[RDT TEST] server recv test message47
[RDT TEST] server recv test message48
[RDT TEST] server recv test message49
==========client 日志==========
==========
[数据传输测试] 进行评分 共发送50条数据 每成功接收一条得2分
[数据传输测试] 可靠数据传输得分为 100 分
========== 所有测试项目得分汇总 ==========
{"scores": {"establish_connection": 100, "reliable_data_transfer":100}}
vagrant@client:/vagrant/tju_tcp$

1.3 连接管理之“四次挥手”

1） 双方先后关闭

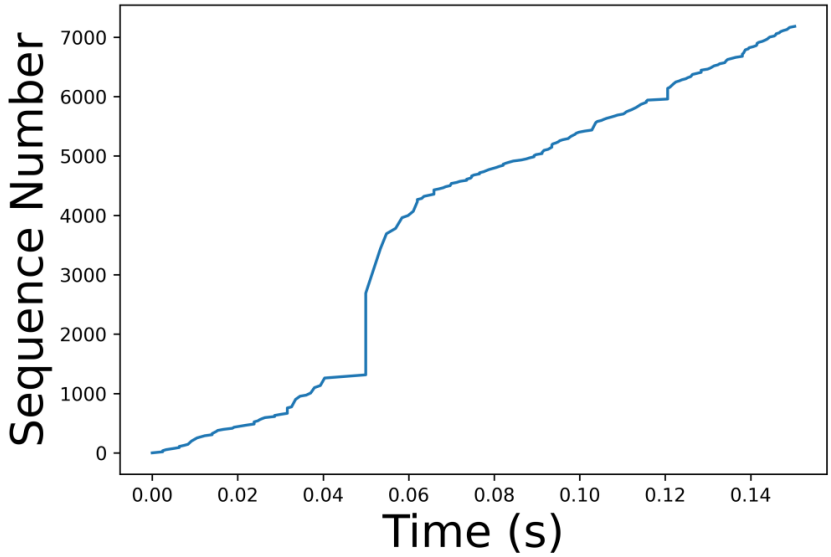
```
========================= 断开连接的测试 ============================
======= 测试双方先后关闭连接的情况 =======
[双方先后关闭测试] 开启服务端和客户端 将输出重定向到文件

[双方先后关闭测试] 等待12s测试结束 三次握手以及四次挥手放在一起应该在12s内完成

[双方先后关闭测试] 打印文件里面的日志
==================================================
==================server 日志====================
[服务端] 测试双方先后断开连接的情况
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=0 ack=0 flags=8
[服务端] 此时服务器状态为 LISTEN, 检查SYN数据包
[服务端] 客户端发送的第一个SYN报文检验通过
[服务端] 发送SYNACK 进入SYN_RECV状态 等待客户端第三次握手的ACK
[服务端] 发送SYNACK src=1324 dst=5678 seq=646 ack=1 flags=12
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=1 ack=647 flags=4
[服务端] 此时服务器状态为 SYN_RECV, 检查ACK数据包
[服务端] 客户端发送的ACK报文检验通过, 成功建立连接, 服务端状态转为ESTABLISHED
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=1 ack=0 flags=6
[服务端] 此时服务器状态为 ESTABLISHED/FIN_WAIT_1, 检查FIN数据包
[服务端] 客户端发送的FIN报文检验通过
{{FIRST FIN PASSED TEST}}
[服务端] 模拟正常双方先后断开连接情况 服务端先响应客户端的FIN 发送ACK 然后等待1s 发FIN ACK
[服务端] 发送ACK src=1324 dst=5678 seq=647 ack=2 flags=4
[服务端] 发送FINACK src=1324 dst=5678 seq=647 ack=2 flags=6
[服务端] 状态转为 LAST_ACK
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=2 ack=648 flags=4
[服务端] 此时服务器状态为 LAST_ACK/CLOSING, 检查ACK数据包
[服务端] 客户端发送的ACK报文检验通过
{{FINAL ACK PASSED TEST}}
==================client 日志====================
发送SYN报文
收到syn_ack报文
发送ACK报文
tju_connet: 三次握手完成!
进入发送线程
[断开连接测试-客户端] 调用 tju_close
发送FIN报文
收到 FIN_ACK 报文
收到 FIN 报文
发送 ACK 报文
[断开连接测试-客户端] 等待10s确保连接完全断开
==================================================

[双方先后关闭测试] 进行评分

[双方先后关闭测试] 双方先后关闭测试部分的得分为 60 分 (满分60分)
```

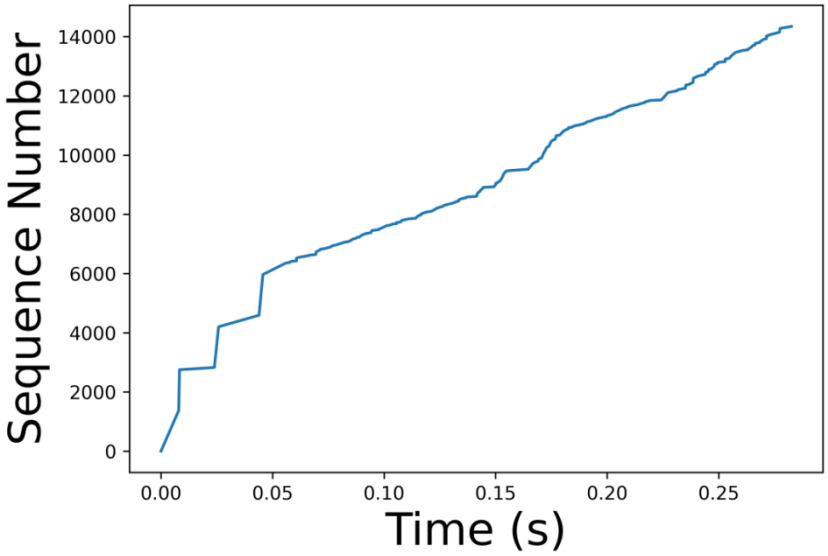
2） 双方同时关闭

```
======= 测试双方同时关闭连接的情况 =======
[双方同时关闭测试] 开启服务端和客户端 将输出重定向到文件

[双方同时关闭测试] 等待12s测试结束 三次握手以及四次挥手放在一起应该在12s内完成

[双方同时关闭测试] 打印文件里面的日志
==================================================
==================server 日志====================
[服务端] 测试双方同时断开连接的情况
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=0 ack=0 flags=8
[服务端] 此时服务器状态为 LISTEN, 检查SYN数据包
[服务端] 客户端发送的第一个SYN报文检验通过
[服务端] 发送SYNACK 进入SYN_RECV状态 等待客户端第三次握手的ACK
[服务端] 发送SYNACK src=1324 dst=5678 seq=646 ack=1 flags=12
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=1 ack=647 flags=4
[服务端] 此时服务器状态为 SYN_RECV, 检查ACK数据包
[服务端] 客户端发送的ACK报文检验通过, 成功建立连接, 服务端状态转为ESTABLISHED
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=1 ack=0 flags=6
[服务端] 此时服务器状态为 ESTABLISHED/FIN_WAIT_1, 检查FIN数据包
[服务端] 客户端发送的FIN报文检验通过
{{FIRST FIN PASSED TEST}}
[服务端] 模拟正常双方同时断开连接情况 服务端假装没有收到FIN包 先按照没有收到FIN的���况发FIN
[服务端] 然后再收到客户端的FIN, 发送ACK响应
[服务端] 发送FINACK src=1324 dst=5678 seq=647 ack=1 flags=6
[服务端] 发送ACK src=1324 dst=5678 seq=648 ack=2 flags=4
[服务端] 状态转为 CLOSING
[服务端] 收到一个TCP数据包 src=5678 dst=1234 seq=2 ack=648 flags=4
[服务端] 此时服务器状态为 LAST_ACK/CLOSING, 检查ACK数据包
[服务端] 客户端发送的ACK报文检验通过
{{FINAL ACK PASSED TEST}}
==================client 日志====================
发送SYN报文
收到syn_ack报文
发送ACK报文
进入发送线程
tju_connet: 三次握手完成!
[断开连接测试-客户端] 调用 tju_close
发送FIN报文
收到 FIN 报文
发送 ACK 报文
收到 ACK 报文
[断开连接测试-客户端] 等待10s确保连接完全断开
==================================================

[双方同时关闭测试] 进行评分

[双方同时关闭测试] 双方同时关闭测试部分的得分为 40 分 (满分40分)
[断开连接的测试] 两种情况的总得分为 100 分 (满分100分)
```

1.4 连接管理之“四次挥手”

多次测试后得到如下结果：





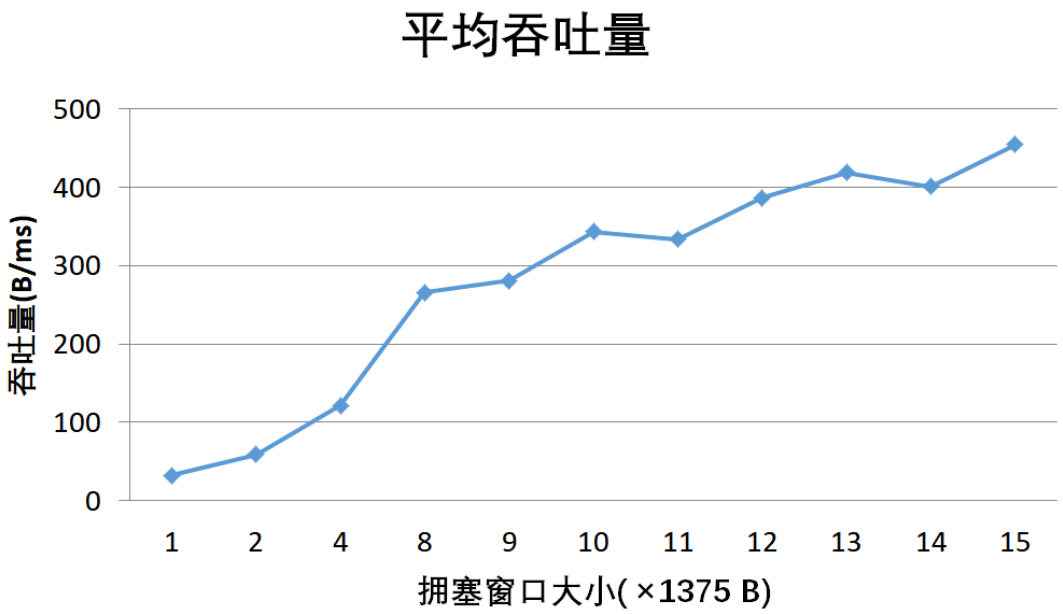
从图中测试结果中可以清楚的看到慢启动过程（也就是指数增长的过程）和改变为拥塞避免（线性增长）的过程。

2. 性能测试

在一个特定的往返间隔内，TCP 发送数据的速率是拥塞窗口与当前 RTT 的函数。当窗口长度是 w 字节，且当 前往返时间是 RTT 毫秒时，则 TCP 的发送速率大约是 w/RTT 。一条连接的平均吞吐量为：

$$\text{一条连接的平均吞吐量} = \frac{0.75 \times w}{RTT}$$

在网络传输速度为 100Mbps，网络延迟为 15ms 的环境下进行了数据的测量，得出了多组 RTT 和拥塞窗口的大小，并绘制下图：



该图中横坐标表示拥塞窗口的大小（实际大小为:横坐标的值×1375B），纵坐标为吞吐量（B/ms）。可以根据折线图看出来，随着拥塞窗口变大，吞吐量也随之变大。

个人总结

本次实验是一次理论和实践的充分结合。在本次实验中实现了一个自己的TCP 协议，在这过程中我充分了解和掌握了计算机网络的运输层，明白了 TCP 的最主要的几个部分（连接管理，可靠数据传输，流量控制，拥塞控制）是怎么工作的，它采用了哪些机制来保证了可靠的，不拥塞的网络环境等。在实际实践过程中我遇到了很多在理论学习过程中没有遇到的问题和困难，也因此对 TCP 的实现过程理解的更加透彻了，查阅资料的能力，写代码的能力，网络协议设计与实现的能力都在一定程度上提高了。最终、实现了所有实践内容，掌握了 TCP 协议设计与实现的方法。