WORK SUMMARY REPORT

工作总结汇报

汇报人 / 迪丽菲娅

2021.10.15

目 录
CONTENTS

01

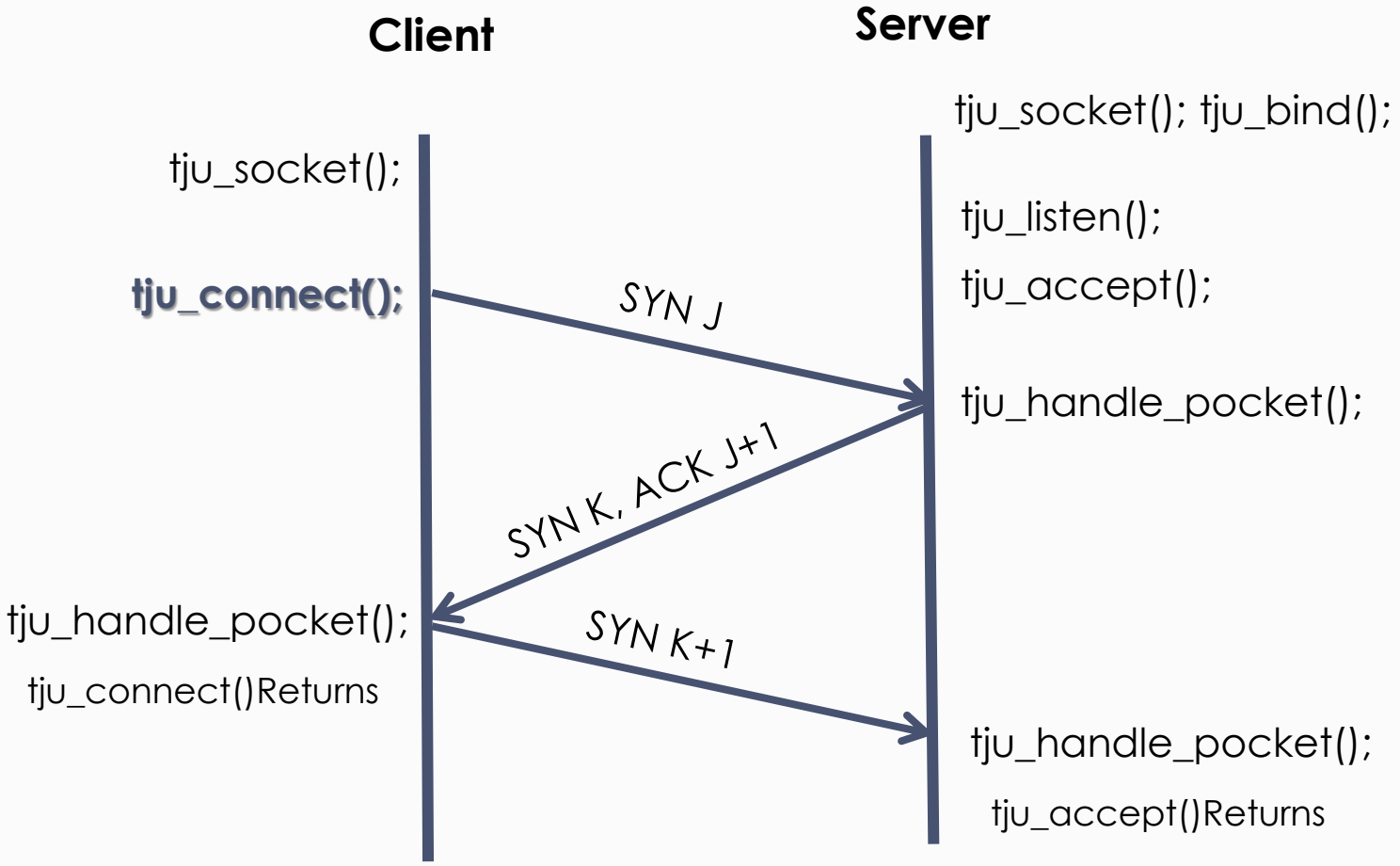
连接管理

三次握手 & 四次挥手

01 连接管理：三次握手 Client

tju_connect();

- 将该socket绑定到本地地址
- 向客户端发送SYN报文，并将状态改为SYN_SENT
- 将该sock存入Ehash中
- 阻塞等待
- 创建发送线程和重传线程

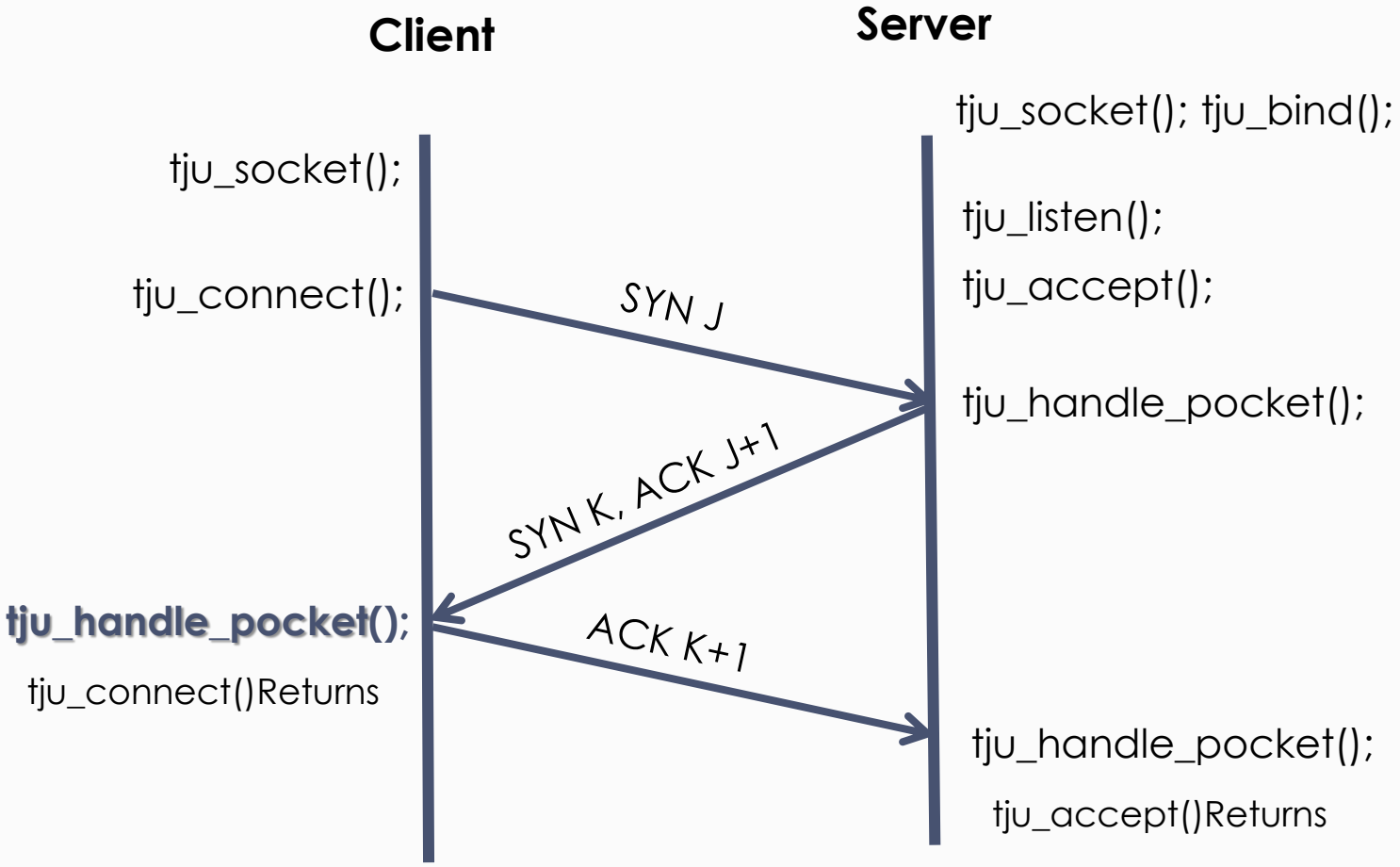


01 连接管理：三次握手 Client

tju_handle_pocket();

- 判断socket的状态
- 判断收到的报文的类型
- 向服务端发送ACK报文
- 将该socket的状态该为ESTABLISH

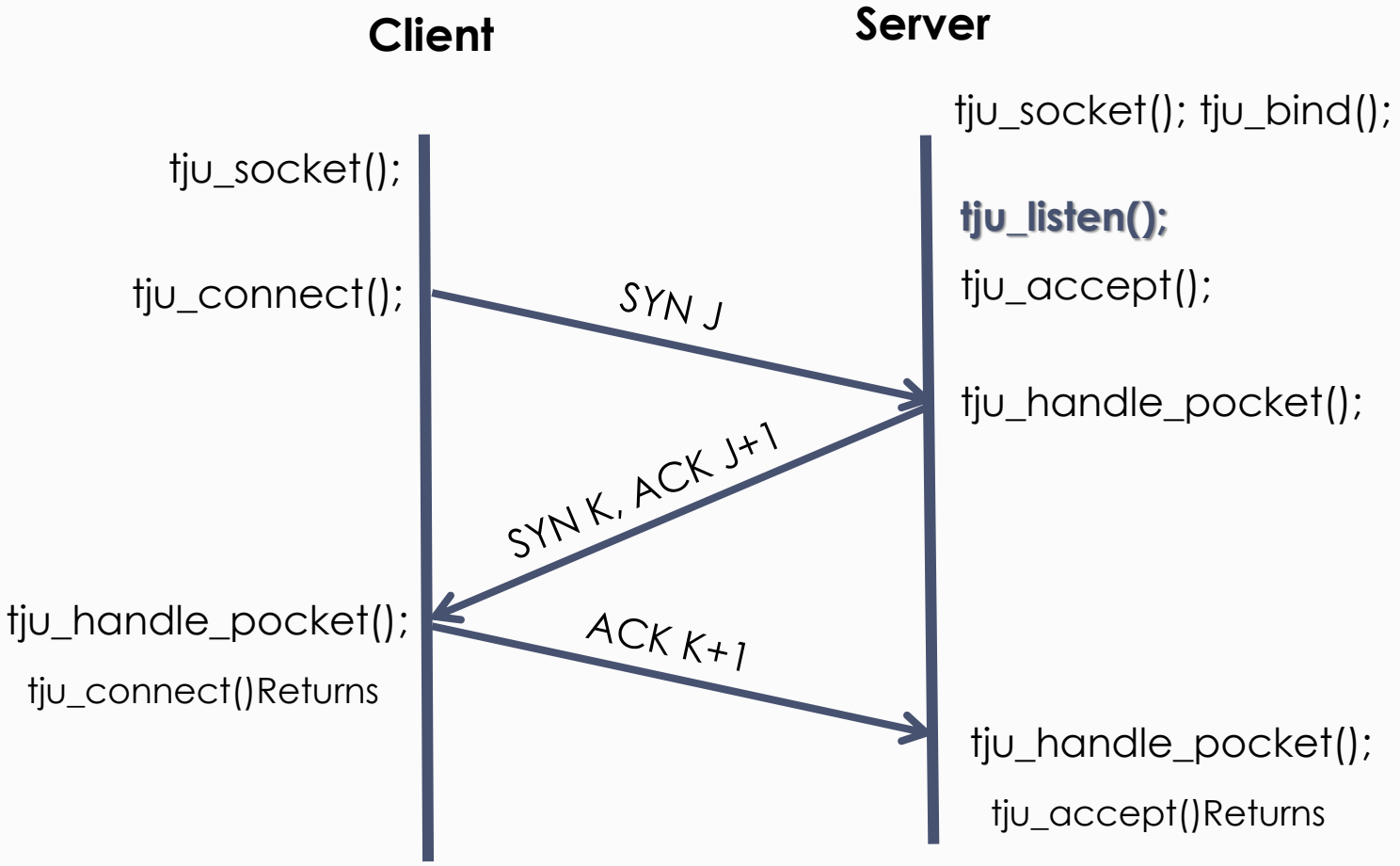
  阻塞状态的tju_connect函数返回



01 连接管理：三次握手 Server

tju_ilsten();

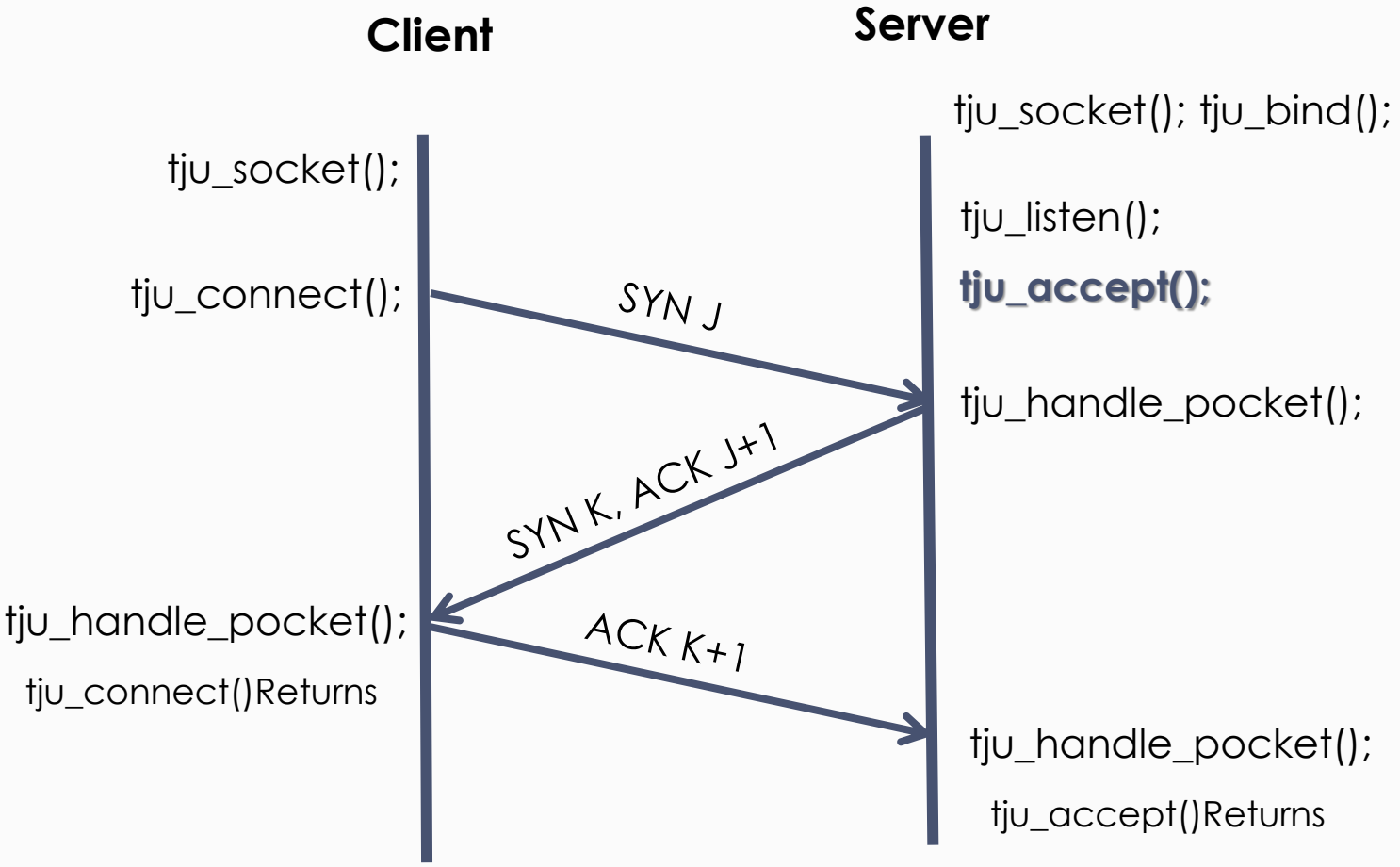
- 将socket的状态该为LISTEN并存入Lhash
- 为socket创建两个队列：未完成连接队列 和 已完成连接队列



01 连接管理：三次握手 Server

tju_accept ();

- 判断全连接队列中是否有socket，若有取出socket，赋值给连接套接字new_conn，并返回；否则阻塞等待
- 创建发送线程和重传线程



01 连接管理：三次握手 Server

tju_handle_pocket();

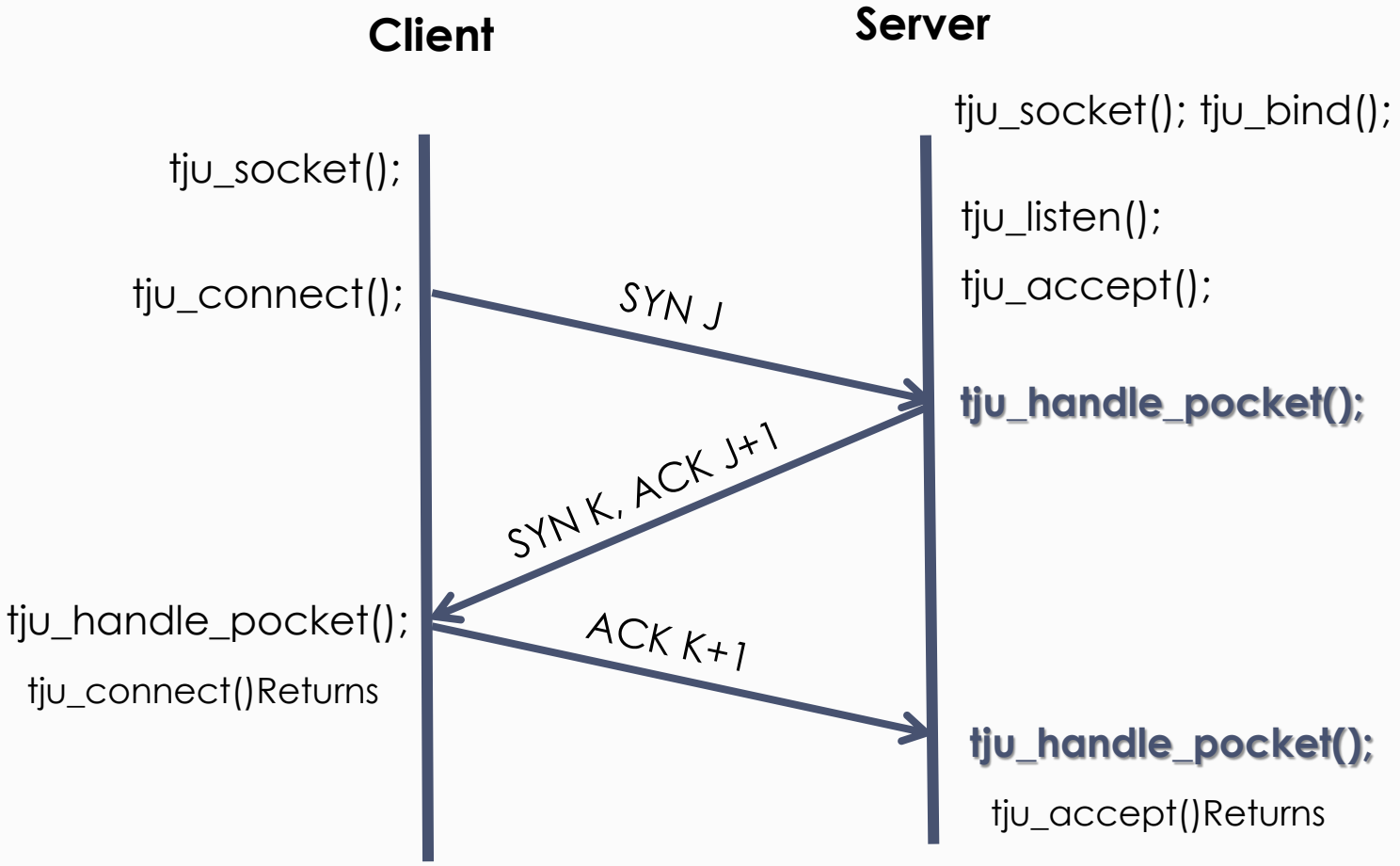
- 收到SYN报文

  将socket存入半连接队列中，套接字的状态修改为SYN_SENT，发送SYN_ACK报文

- 收到SYN报文

  将socket存入全连接队列中，套接字的状态修改为ESTABLISH

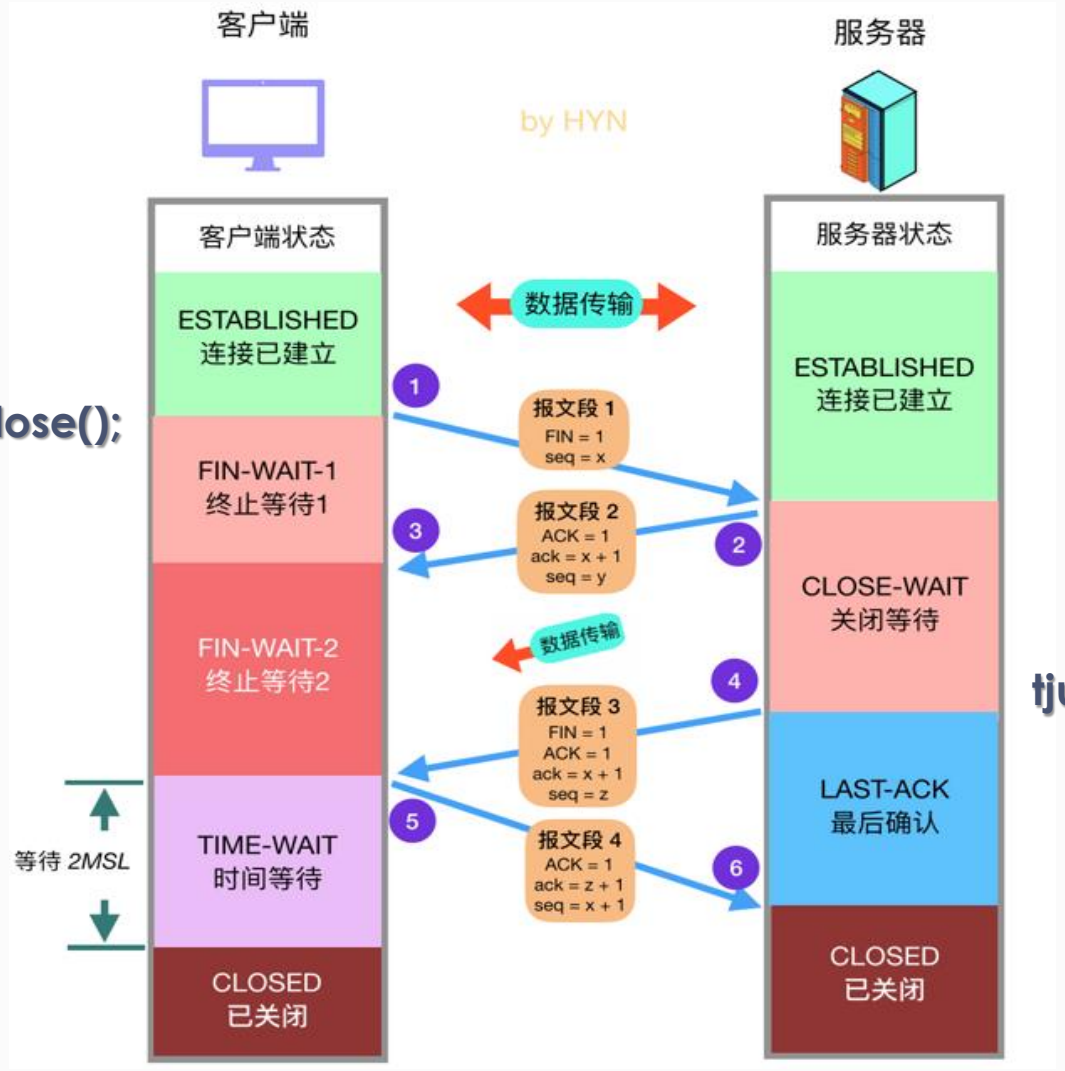
  阻塞状态的tju_accept函数执行



01 连接管理：四次挥手

tju_close();

- 发送FIN报文
- 如果socket状态为ESTABLISH，将状态为FIN_WAIT
- 如果socket状态为CLOSE_WAIT，将状态为LAST-ACK
- 如果状态为CLOSED，释放资源；否则阻塞等待



01

连接管理：四次挥手

tju_handle_pocket();

- ESTABLISHED & FIN

  发送FIN_ACK报文，状态改为CLOSE-WAIT,等待一段时间后，调用tju_close

- FIN_WAIT_1 & FIN_ACK

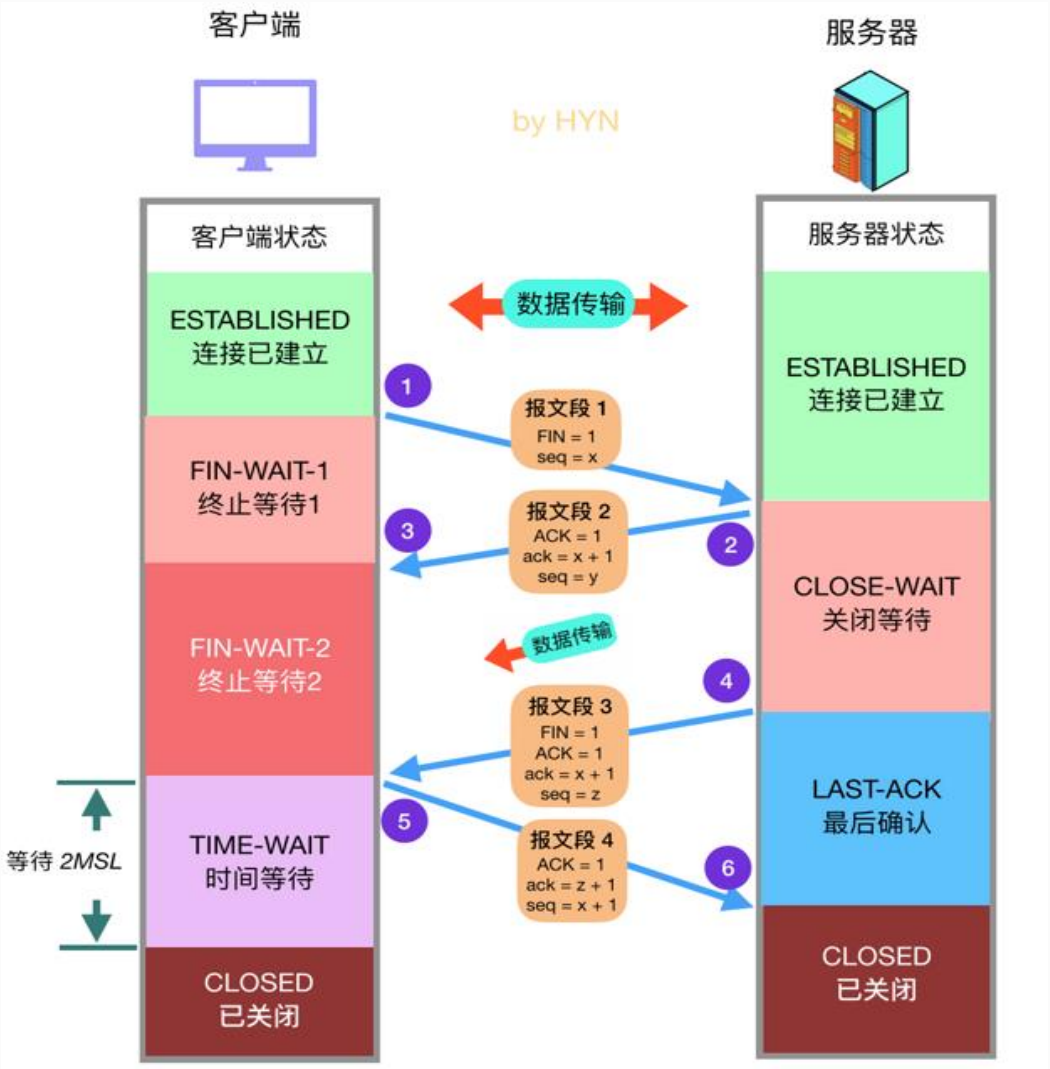
  状态改为FIN_WAIT_2

- FIN_WAIT_2 & FIN

  发送FIN_ACK报文，状态改为TIME-WAIT,等待一段时间后，改为CLOSED

- LAST_ACK & FIN_ACK

  状态改为CLOSED



01 连接管理：四次挥手

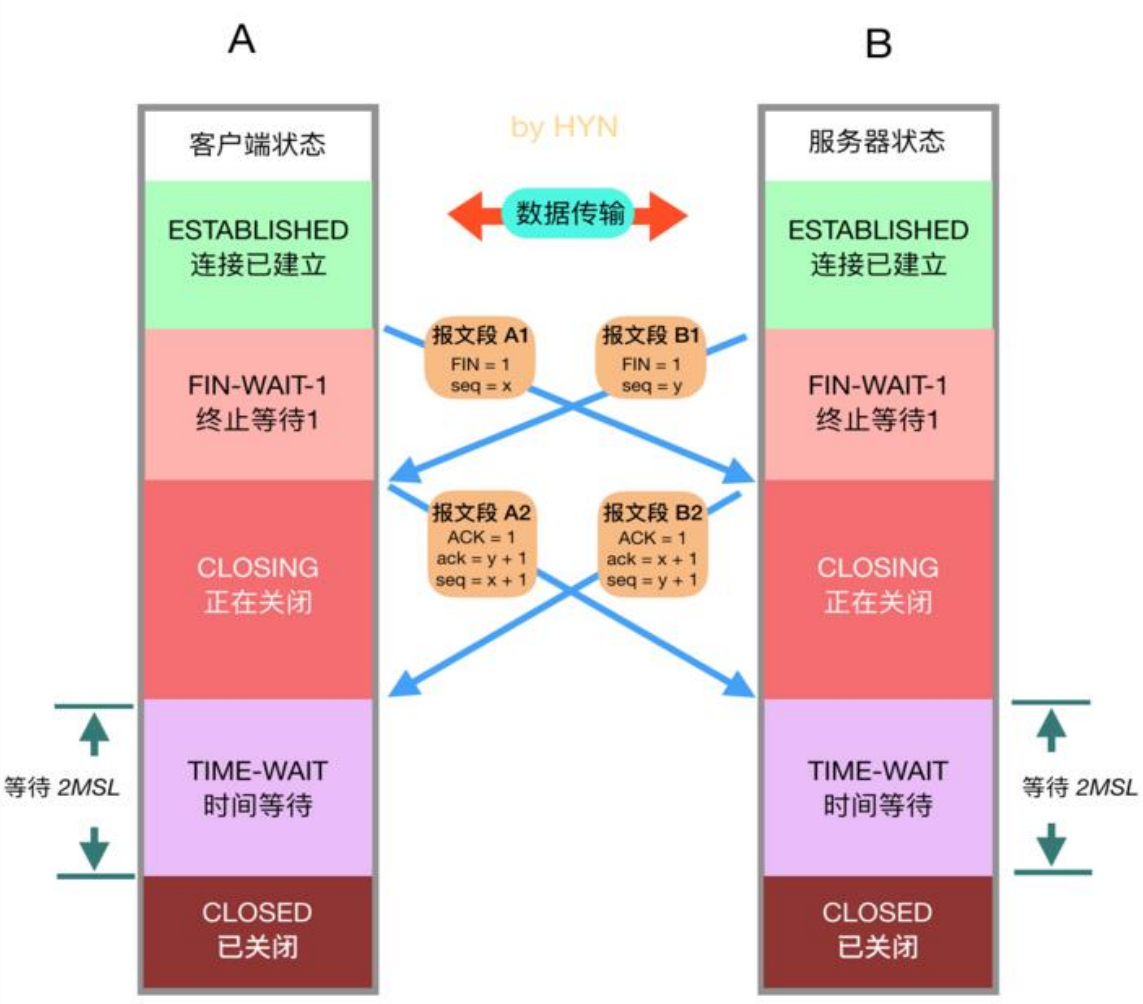
tju_handle_pocket();

- FIN_WAIT_1 & FIN

  发送FIN_ACK报文，状态改为CLOSING

- CLOSING & FIN_ACK

  发送FIN_ACK报文，状态改为TIME-WAIT,等待一段时间后，改为CLOSED



02

可靠数据传输

GBN协议的实现 & RTT的测量

02

可靠数据传输：GBN协议的实现

tju_send();

- 调用tju_buffered()函数

tju_buffered();

- 如果缓冲区已满，阻塞等待；否则，将数据存到发送缓冲区

tju_recv ();

- 如果接收缓冲区有数据取出数据

sending_thread();

- 满足：发送缓冲区中还有未发送过的数据 & 没有在重传 & 发送窗口有剩余序列号，时开始发送

  sock->send_len：发送缓冲区内已发送的数据大小

  sock->is_retansing：值为false表明没有在重传

- 发送情况：如果需发送的数据可以装满一个报文，先按整报文发；剩余的数据再装进一个报文发送

- 如果发送窗口的base==nextseq，调用startTimer()启动计时器

02

可靠数据传输：GBN协议的实现

startTimer();

- 调用time.h头文件中setitimer()函数来实现计时器，参数为sock->window.wnd_send->timeout。当settimer()函数超时时会调用回调函数timeout_handler()

stopTimer();

- 重新调用settimer()来停止计时器

Timeout_handler ();

- 将RETRANS和TIMEOUT_FLAG的值置为1
- RETRANS：值为1，进入重传
  TIMEOUT_FLAG：表明发生超时事件(拥塞管理)

02

可靠数据传输： GBN协议的实现

retrans_thread();

- 在RETRANS值不为1时阻塞等待
- 将sock->is_retansing置为true
- 重传从base到nextseq的所有报文
- 如果发送窗口的base==nextseq，调用start_timer()启动计时器

tju_handel_pocket();

- 收到的报文的序号等于expect_seq，并且接收缓冲区有空间，数据段存入缓冲区，更新相关变量，发送ACK
- 收到的报文的序号不等于expect_seq，发送ACK
- 如果收到的ACK序号小于base直接丢弃，否则更新base。
  ① 更新后的base==nextseq调用stopTimer()
  ② 更新后的base!=nextseq调用stopTimer()再调用startTimer()
- 释放发送缓冲区中

02

可靠数据传输： RTT的测量

sending_thread();

- 判断is_estimating_rtt 是否为false，如果是，将其的值设为true
- 调用gettimeofday()库函数，将发送时间记录在sock->window.wnd_send->send_time
- 用rtt_expect_ack来保存期待的报文的ack序号

tju_handel_pocket();

- 判断is_estimating_rtt是否为真
- 判断报文的ack序号是否等于rtt_expect_ack，如果相等调用TimeoutInterval()来计算出RTT的值
- 将is_estimating_rtt改为false

可靠数据传输： RTT的测量

TimeoutInterval();

- 用gettimeofday()获取收到报文的时间，与send_time做差，得到RTT的值。再根据下面的四个公式计算出超时时间，存在sock->window.wnd_send->timeout中。

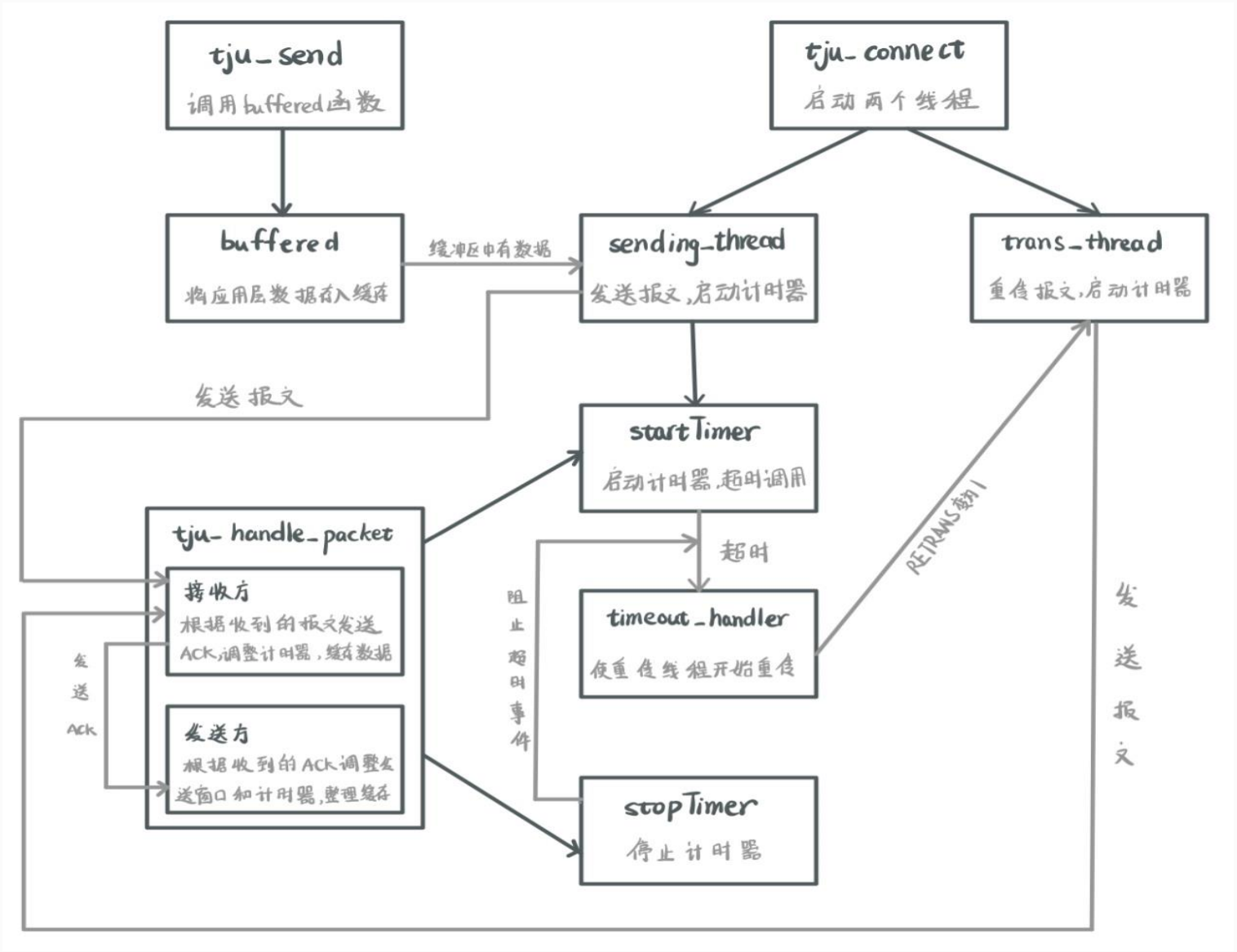
$$EstimatedRTT = (1-\alpha)\cdot EstimatedRTT + \alpha \cdot SampleRTT$$

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

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

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

可靠数据传输


tju_send
调用buffered函数
tju_connect
启动两个线程
buffered
将应用层数据存入缓存
缓冲区中有数据
sending_thread
发送报文,启动计时器
trans_thread
重传报文,启动计时器
发送报文
startTimer
启动计时器,超时调用
tju_handle_packet
接收方
根据收到的报文发送ACK,调整计时器,缓存数据
发送方
根据收到的ACK调整发送窗口和计时器,整理缓存
发送ACK
超时
timeout_handler
使重传线程开始重传
阻止超时事件
RETRANS数1
stopTimer
停止计时器
发送报文

拥塞控制

拥塞控制 & 流量控制

03 拥塞控制

在拥塞控制中需要满足发送方中未被确认的数量不会超过cwnd和rwnd中的最小值，即：

$$LastAcked - LastByteAcked \leqslant \min\{cwnd, rwnd\}$$

rwnd

- 在发送ACK报文时，计算出发送缓冲区内剩余的空间，将其赋值给报文头部的rwnd字段
- 在收到ACK报文时把头部字段中rwnd的值赋值给sock->window.wnd_send->rwnd

sending_thread();

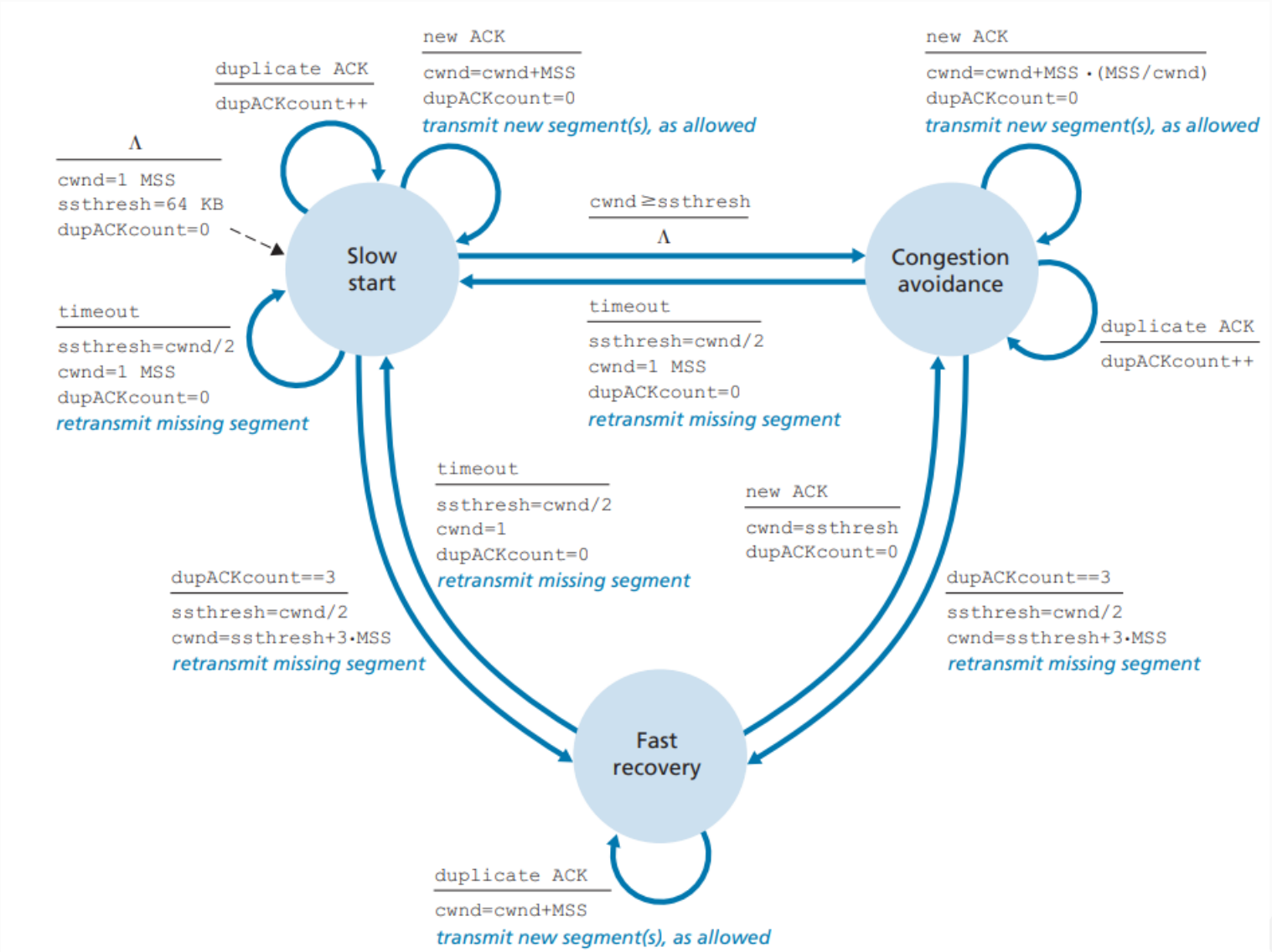
- 在调用sending_thread()发送报文时，需要满足上述式子，因此在发送前需要先判断：

$$wnd\_next + dlen - wnd\_base \leqslant \min(cwnd, rwnd)$$

只有上述式子时，才发送报文

03 拥塞控制

1. 收到新的ACK
2. cwnd ⩾ ssthresh
3. 收到重复ACK
4. 收到的重复ACK的数量为3
5. 发生超时

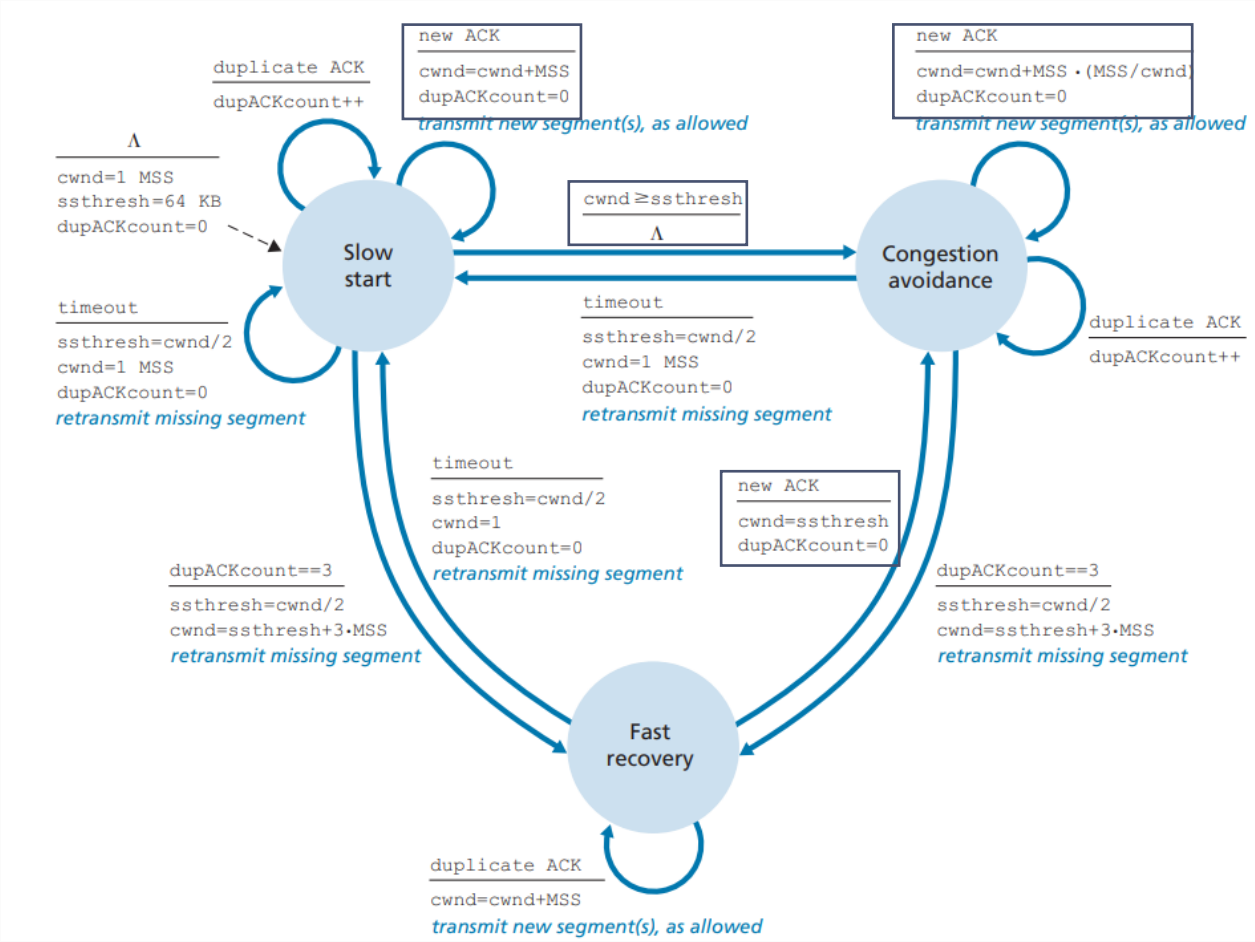


tju_handle_pocket();

```
IF(status == SLOW_START){
    cwnd = cwnd + MSS;
    IF(cwnd ⩾ ssthresh)
        status = CONGESTION_AVOIDENCE;
}ELSE IF(status == CONGESTION_AVOIDENCE){
    cwnd = cwnd + MSS * (MSS / cwnd);
} ELSE IF(status == FAST_RECOVERY){
    cwnd = ssthresh;
    status = CONGESTION_AVOIDENCE;
}
dupACKcount = 0;
```

1 收到新的ACK

2 cwnd ⩾ ssthresh

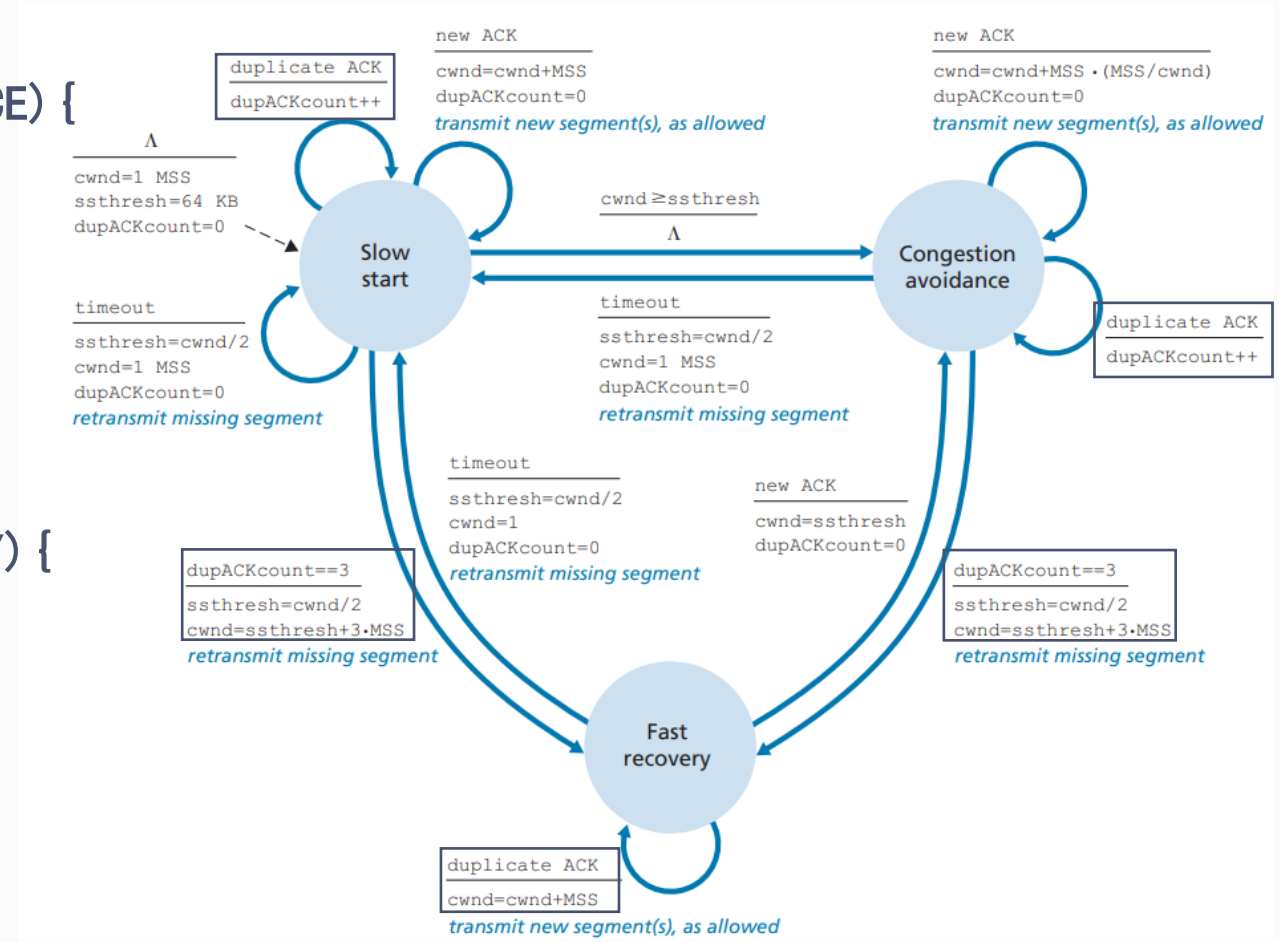


tju_handle_pocket();

```
IF(status == SLOW_START or CONGESTION_AVOIDENCE){
      dupACKcount ++;
}ELSE IF(status == FAST_RECOVERY){
     cwnd = cwnd + MSS;
     status = CONGESTION_AVOIDENCE;
}
IF(dupACKcount == 3 && status != FAST_RECOVERY){
     ssthresh = cwnd / 2;
     cwnd = ssthresh + 3 * MSS;
     status = FAST_RECOVERY;
     RETRANS = 1;
}
```

3 收到重复ACK

4 收到的重复ACK的数量为3

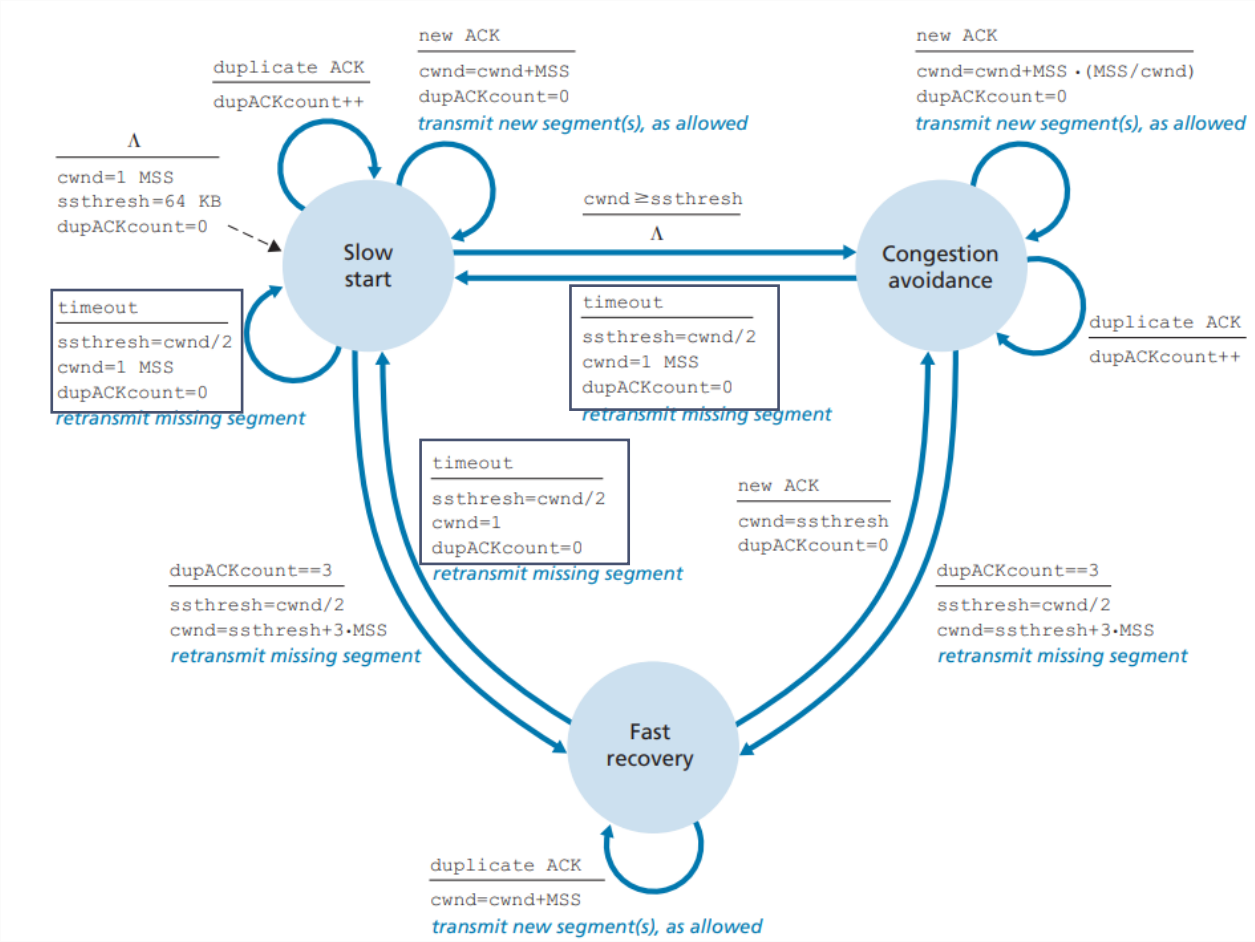


retrans_thread();

```
IF(TIMEOUT_FLAG){
    ssthresh = cwnd / 2;
    dupACKcount = 0;
    status = SLOW_START;
}
```

进入retrans_thread()重传报文有两种可能性：发生超时 或者 收到三个重复ACK。可以用TIMEOUT_FLAG将两种情况区分开来。

5 发生超时



04

实验结果

04 实验结果：连接管理

选择vagrant@client: /vagrant/tju_tcp

```
> cd /vagrant/tju_tcp/test && make
rm -f receiver test_client client.log server.log
gcc -pthread -g -ggdb -DDEBUG -I../inc ./test_receiver.c -o receiver ../build/tju_packet.o ../build/kernel.o ../build/tju_tcp.o
gcc -pthread -g -ggdb -DDEBUG -I../inc ./test_client.c -o test_client ../build/tju_packet.o ../build/kernel.o ../build/tju_tcp.o

[自动测试] 开启服务端和客户端 将输出重定向到文件

[自动测试] 等待8s测试结束 三次握手应该在8s内完成

[自动测试] 打印文件里面的日志
======================================================================
[服务端] 收到一个TCP数据包
[服务端] 客户端发送的第一个SYN报文检验通过
{{GET SCORE}}
[服务端] 发送SYNACK 等待客户端第三次握手的ACK
[服务端] 收到一个TCP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
{{GET SCORE}}
{{TEST SUCCESS}}
发送SYN报文
收到syn_ack报文
发送ACK报文
tju_connet: 三次握手完成!
======================================================================

[自动测试] 进行评分
{"scores": {"establish_connection": 100}}
vagrant@client:/vagrant/tju_tcp$
```

04 实验结果：连接管理

```
========================= 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)
```

```
======= 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)
```

04 实验结果：可靠数据传输

vagrant@client: /vagrant/tju_tcp

```
==================================================
==================server 日志=====================
收到客户端的syn报文
发送syn_ack报文
收到客户端的ack报文
tju_accept: 三次握手完成!
Interrupted
收到seq = 1 的报文  发送ACK报文 ack = 17
收到seq = 17 的报文  发送ACK报文 ack = 33
收到seq = 33 的报文  发送ACK报文 ack = 49
收到seq = 49 的报文  发送ACK报文 ack = 65
收到seq = 65 的报文  发送ACK报文 ack = 81
收到seq = 81 的报文  发送ACK报文 ack = 97
收到seq = 97 的报文  发送ACK报文 ack = 113
收到seq = 113 的报文  发送ACK报文 ack = 129
收到seq = 129 的报文  发送ACK报文 ack = 145
收到seq = 145 的报文  发送ACK报文 ack = 161
收到seq = 161 的报文  发送ACK报文 ack = 177
[RDT TEST] server recv test message0
收到seq = 177 的报文  发送ACK报文 ack = 193
收到seq = 193 的报文  发送ACK报文 ack = 209
收到seq = 209 的报文  发送ACK报文 ack = 225
收到seq = 225 的报文  发送ACK报文 ack = 241
收到seq = 257 丢弃报文 发送ACK报文 ack = 241
收���seq = 273 丢弃报文 发送ACK报文 ack = 241
收到seq = 289 丢弃报文 发送ACK报文 ack = 241
收到seq = 305 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message1
收到seq = 321 丢弃报文 发送ACK报文 ack = 241
收到seq = 337 丢弃报文 发送ACK报文 ack = 241
收到seq = 353 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message2
收到seq = 369 丢弃报文 发送ACK报文 ack = 241
收到seq = 385 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message3
收到seq = 417 丢弃报文 发送ACK报文 ack = 241
收到seq = 433 丢弃报文 发送ACK报文 ack = 241
收到seq = 449 丢弃报文 发送ACK报文 ack = 241
收到seq = 465 丢弃报文 发送ACK报文 ack = 241
收到seq = 481 丢弃报文 发送ACK报文 ack = 241
收到seq = 497 丢弃报文 发送ACK报文 ack = 241
收到seq = 513 丢弃报文 发送ACK报文 ack = 241
收到seq = 529 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message4
收到seq = 545 丢弃报文 发送ACK报文 ack = 241
收到seq = 561 丢弃报文 发送ACK报文 ack = 241
收到seq = 593 丢弃报文 发送ACK报文 ack = 241
收到seq = 609 丢弃报文 发送ACK报文 ack = 241
收到seq = 625 丢弃报文 发送ACK报文 ack = 241
收到seq = 641 丢弃报文 发送ACK报文 ack = 241
```

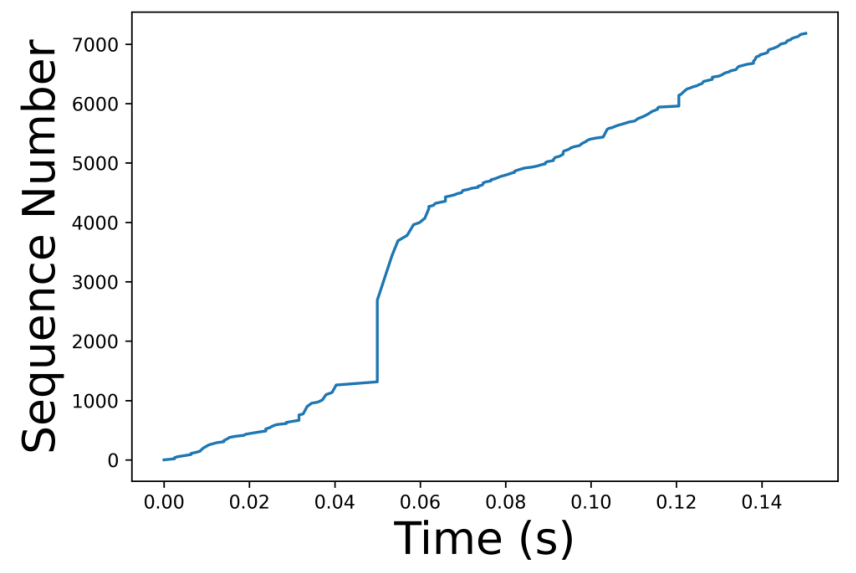
vagrant@client: /vagrant/tju_tcp

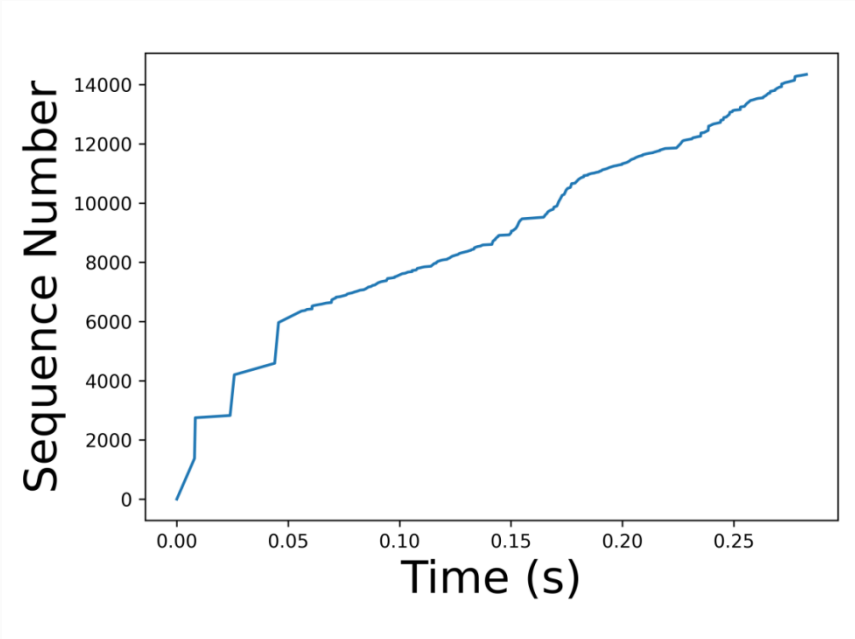
```
收到seq = 641 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message5
收到seq = 657 丢弃报文 发送ACK报文 ack = 241
收到seq = 673 丢弃报文 发送ACK报文 ack = 241
收到seq = 689 丢弃报文 发送ACK报文 ack = 241
收到seq = 705 丢弃报文 发送ACK报文 ack = 241
收到seq = 721 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message6
收到seq = 737 丢弃报文 发送ACK报文 ack = 241
收到seq = 753 丢弃报文 发送ACK报文 ack = 241
收到seq = 785 丢弃报文 发送ACK报文 ack = 241
[RDT TEST] server recv test message7
[RDT TEST] server recv test message8
[RDT TEST] server recv test message9
[RDT TEST] server recv test message10
[RDT TEST] server recv test message11
[RDT TEST] server recv test message12
[RDT TEST] server recv test message13
[RDT TEST] server recv test message14
收到seq = 241 的报文  发送ACK报文 ack = 801
[RDT TEST] server recv test message15
[RDT TEST] server recv test message16
[RDT TEST] server recv test message17
[RDT TEST] server recv test message18
[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
```

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$
```

03 实验结果：拥塞管理





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。

THANKS