**《计算机网络实践》课程报告**





**TCP 在应用层的设计与实现**

**学 号 3022244290**

**姓 名 陈秋澄**

**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 赵增华**

**2024年8月31日**

# 一、报告摘要

简要介绍需要解决的具体问题、协议设计和实现，以及主要实验结果。（可放到最后写）

# 二、任务要求

本次实践需要在应用层使用 UDP 作为基本协议实体并在其基础上实现 TCP 的基本功能。

UDP协议作为基础的网络层通信协议，其核心职责仅限于将报文传递给下层的协议实体，并不包含如TCP协议所提供的可靠性保障等特性。然而，正因如此，UDP协议能够实现更高的平均传输速率，使其特别适用于那些对可靠性要求较低，但对实时性要求较高的应用场景，如流媒体传输服务。

TCP的主要目标是在不同进程间提供稳定且安全的面向连接服务。为了在相对不可靠的通信环境中实现这一目标，系统需要具备以下功能：连接的建立与关闭、基础的数据传输机制、确保通信的可靠性以及实施流量控制。

## **2.1连接管理**

TCP客户端首先在三次握手后与指定的服务器建立连接，随后通过此连接与服务器进行数据交换，最终通过四次挥手关闭该连接。此外，需要在有丢包和延迟的情况下，正确处理在建立连接时每一种数据包丢失的情况以及关闭连接时服务器端和客户端先后或同时断开时的情况；并且保证数据传输的稳定性。

连接管理部分，主要针对：连接建立——三次握手以及连接关闭——四次挥手两方面。

## **2.2可靠数据传输**

TCP给发送的每一个数据包都分配一个唯一的序列号。接收端在接收到数据包后，会根据序列号对数据进行排序，并将有序的数据传送给应用层。序列号确保了数据的按序到达，并允许接收端丢弃重复的数据包。并且，接收端在成功接收到数据包后，会向发送端发送一个确认应答，告知其已接收到的最后一个数据包的序列号。如果发送端在预设的时间内未收到确认应答，则认为数据包可能已丢失，将重新发送该数据包。此外，TCP为每一个已发送的数据包设置一个超时计时器。如果在超时计时器到期之前未收到相应的确认应答，则发送端将重发该数据包。在本模块中，我们需要完成以下任务：

1. 开发并集成序列号分配、数据校验、累积确认机制以及超时重传功能，确保数据传输的完整性和可靠性。
2. 部署定时器机制，依据 RFC793 Sec 3.7的描述，以动态追踪并实时评估往返时延（RTT），优化数据传输效率。
3. 正确管理发送与接收缓冲区、实现窗口滑动机制，以控制数据流、避免拥塞：在TCP协议的数据传输优化策略中，滑动窗口技术通过自适应地调整窗口尺寸，控制两台通信主机间的数据流。具体到数据传输协议的选择上，采用GBN协议来实现以上关键功能。

## **2.3流量控制**

流量控制的核心目的是对系统中的数据流、请求或信息流量进行管理和调节，以确保系统的稳定运行和资源的合理利用。本模块需保证：

1. 接收端具备评估当前空闲缓冲区容量的能力，据此设定“Advertised Window”值。
2. 发送端依据从接收端获取的“Advertised Window”信息，灵活调整其自身的数据发送范围（发送窗口）。
3. 针对接收端“Advertised Window”显示为0的特殊情形，发送端采取恰当的应对措施，执行有效的零窗口探测机制，以确保数据传输的顺畅与高效。

## **2.4拥塞控制**

拥塞控制是计算机网络中一个重要的概念，旨在防止过多的数据注入网络，通过调节发送方的发送速率来避免网络中的路由器或链路过载，确保网络性能的稳定性和高效性。

依照TCP Reno要求，本模块需要实现以下三种机制：慢启动、拥塞避免、快速重传，即在发生超时、冗余重传、收到new ack等几种情况下的拥塞状态转化。

# 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”要求的所有内容。这是评分的依据。**

## **3.1 总体设计**

**根据任务分析划分功能模块、理清模块组织结构和关系等**。

### 3.1.1模块设计

根据实践指导书的说明，可将TJU\_TCP的模块架构划分成以下层次，如图3-1所示：

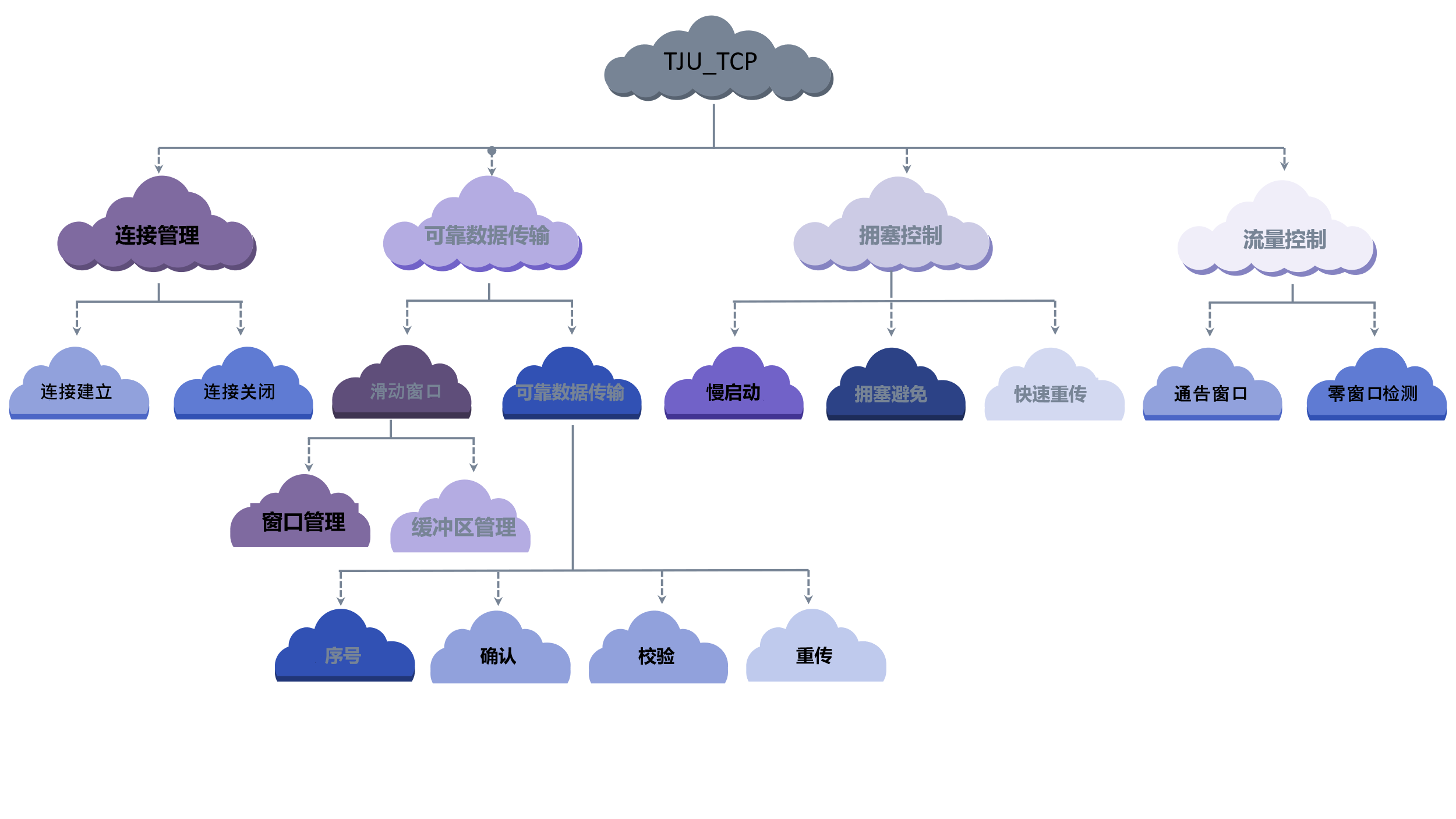


图3-1 功能模块关系

## **3.2 连接管理的设计**

说明三次握手建立连接、四次挥手关闭连接的原理、主要数据结构和协议规则。用FSM图表示主要工作流程。

### 3.2.1原理

**1. 连接建立相关原理**

TCP进行三次握手建立连接时，操作系统会为其维护两个队列，分别是半连接队列和全连接队列。三次握手过程原理如下：

首先client端发起第一次握手，发送请求连接的报文段报文完成发送后，进入SYN\_SENT状态。相关参数为SYN = 1，seq = n。

接着，server端收到报文，将该连接加入到SYN半连接队列中，并在对该报文段校验后向client端发送响应报文，进入SYN\_RECV状态。相关参数为SYN = 1，ACK = 1，seq = m，ack = n+1。此时，第二次握手完成。

最后，第三次握手。client端收到报文段，分别设置ACK = 1，seq = n+1，ack = m+1，并发送报文，此后进入ESTABLISHED状态；server端收到client端第三次握手响应后，将该连接从半连接队列中取出，建立1个新连接，加入到ACCEPT全连接队列中，同样将状态转为ESTABLISHED。

此后当进程调用accept()函数时，再将该连接取出来。

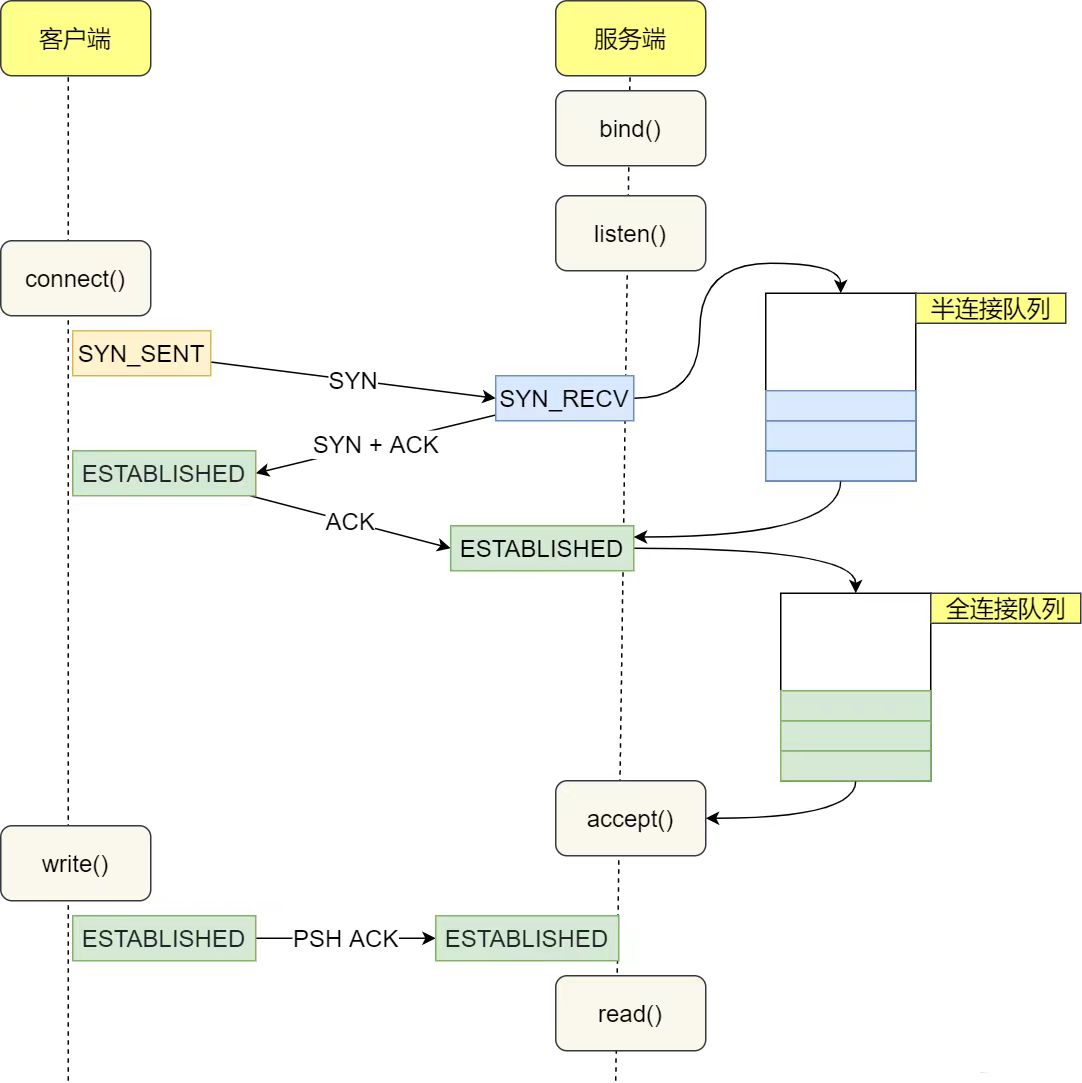


图3-2 三次握手流程

**2. 连接关闭相关原理**

(1)双方先后断开连接

首先，client端发起第一次挥手，状态由ESTABLISHED转为FIN\_WAIT\_1，向server端发送的报文相关参数为连接终止位FIN = 1，seq = u。当client端发送FIN报文时，表示其已没有数据要发送了。当然，client端此时还是可以接收server的数据的。

Server端收到client端的报文，发起第二次挥手并将状态从ESTABLISHED转为CLOSE\_WAIT，并等待server端自身的socket关闭等操作。报文相关参数为ACK =1, seq = v，ack = u+1。

Client端收到server端发起的第二次挥手，将状态由FIN\_WAIT\_1转为FIN\_Wait\_2,等待server端关闭。

Server端发起第三次挥手，状态由CLOSE\_WAIT转为LAST\_ACK。报文相关参数为FIN = 1，ACK = 1，seq = w，ack = u+1。

Client收到server端发起的第三次挥手，并发起第四次挥手，状态也随之从FIN\_WAIT\_2转为TIME\_WAIT。报文相关参数为ACK =1, seq = u+1，ack = w+1。

最后，server收到client端发起的第四次挥手，进入CLOSED状态。与此同时，client端等待2MSL后，也同样进入CLOSED状态。

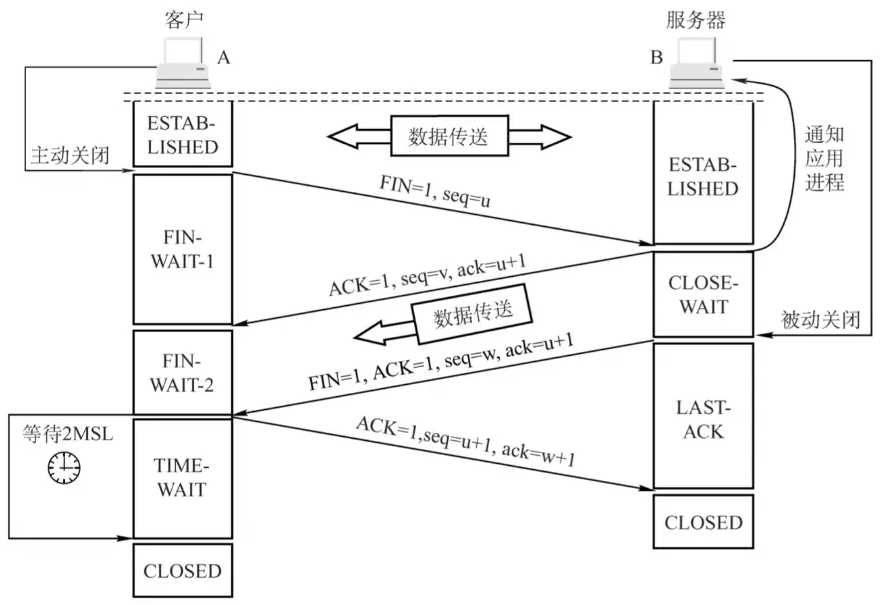


图3-3 四次挥手流程

（2）双方同时断开连接

客户端和服务端状态对称，因为同时断开连接即同时发出FIN包，所以二者的状态转换和上文（1）中的客户端的状态转换类似，只不过把FIN\_WAIT\_2叫做CLOSING。此外，双方要在没有收到自己发出的FIN包对应的ACK包的情况下，对对方的FIN包做出应答。

### 3.2.2主要数据结构

依据上文提到的原理，TCP报文首部结构如下表所示：

**表1 TCP报文首部结构**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| source\_port | 源端口 |
| destination\_port | 目的端口 |
| seq\_num | Sequence序号 |
| ack\_num | Acknowledgement序号 |
| hlen | 包头长度 |
| plen | 整个数据包长度 |
| flags | 标志位，如SYN、FIN、ACK |
| advertised\_window | 接收方发送给发送方用于流量控制的建议窗口大小 |
| ext | 额外的数据，无实际意义 |

此外，这里厘清一些概念：

1. 序号字段seq占4字节，TCP报文首部的序号字段值指本报文所发送的数据的第一个字节的序号。
2. 确认号字段ack同样占4字节，指期待收到对方下一个报文段的第一个数据字节的序号。
3. 确认ACK：当且仅当ACK = 1，确认号字段ack才有效，且建立连接后所有报文段的ACK均为1。
4. 同步SYN：用于在连接建立时同步序号，SYN = 1指该报文为连接请求或连接接受报文。
5. 终止FIN：当FIN = 1，该报文段的发送方的数据已经全部发送完成，并请求释放传输连接。

依据上文，TCP进行三次握手建立连接时，操作系统会为其维护两个队列，分别是半连接队列和全连接队列。因此，增加相关数据结构创建队列。

**表2 socket相关数据结构**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| tju\_tcp\_t\* syn\_queue[MAX\_SOCK] | 半连接队列 |
| uint16\_t syn\_num | 半连接队列元素个数 |
| pthread\_mutex\_t syn\_lock | 半连接队列的锁 |
| tju\_tcp\_t\* accept\_queue[MAX\_SOCK] | 全连接队列 |
| uint16\_t accept\_num | 全连接队列元素个数 |
| pthread\_mutex\_t accept\_lock | 全连接队列的锁 |
| tju\_tcp\_t\* get\_from\_syn() | 从半连接队列中取出socket |
| tju\_tcp\_t\* get\_from\_accept() | 从全连接队列中取出socket |

下面，对socket地址数据结构进行说明：

**表3 socket地址数据结构**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| typedef struct {  uint32\_t ip;  uint16\_t port;  }tju\_sock\_addr; | 结构体  IP地址  端口号 |

### 3.2.3协议规则

**1. 连接建立的协议规则**

（1）基本流程

处于CLOSED状态的Server端调用tju\_sock接口获取处于LISTEN状态的socket。

Client端从tju\_conn接口向Server端发送报文a，以请求连接（必要时进行超时重传，在3.3节可靠数据传输章节与RDT一并阐述），并将状态转为SYN\_SENT。

Server端收到报文a，发送报文b，将状态转为SYN\_RCVD。

Client端收到报文b，将状态转为ESTABLISHED，向上层接口返回一个建立连接的socket。同时向server端发送报文c。

Server端1收到报文c，建立新的socket并将其放到初始的socket全连接队列。此后当进程调用accept()函数时，再将该连接取出来。

FSM图如下图所示：

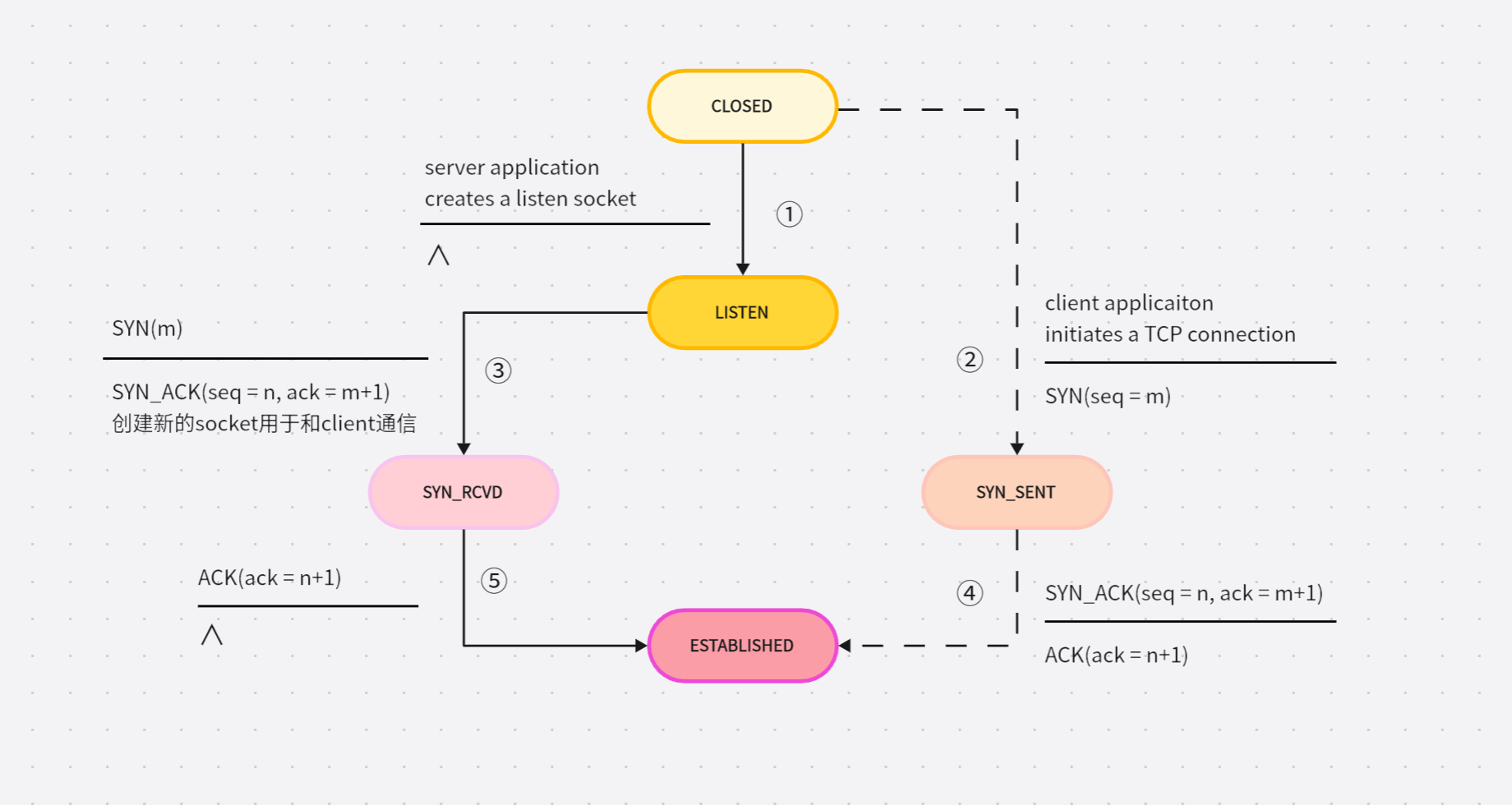


图3-4 连接建立的状态机

（2）异常情况处理

下面，介绍针对不同丢包情况设计的不同的应对思路。

1. SYN包丢失

当遇到SYN包丢失的问题时，客户端会启动一个专门的定时器线程来监控。若此过程中发生超时，客户端将周期性地重新发送SYN包，并且每次重传后，其超时重传时间（RTO）会加倍。这一过程将持续进行，直至客户端接收到来自服务器的第二个包的确认信号，随后关闭该定时器线程。

1. SYN | ACK包丢失

若SYN | ACK包在传输过程中丢失，客户端将无法顺利过渡到下一个状态，因此其定时器线程将保持开启状态。在此情境下，客户端的定时器线程将负责重新发送SYN包，以触发服务器再次发送SYN | ACK包。同时，服务器端也会启动其定时器线程，以应对可能的重传需求。

1. ACK包丢失

当ACK包未能成功送达服务器时，服务器将停留在当前状态，此时服务器的定时器线程将保持开启状态以监测超时。一旦超时发生，服务器的定时器线程将负责重新发送SYN | ACK包，以促使客户端再次发送ACK包。这一过程将重复进行，直到服务器成功接收到ACK包并更新其状态，随后关闭定时器线程。

**2. 连接关闭的协议规则**

（1）双方先后断开连接

该情况对应下图右侧流程。

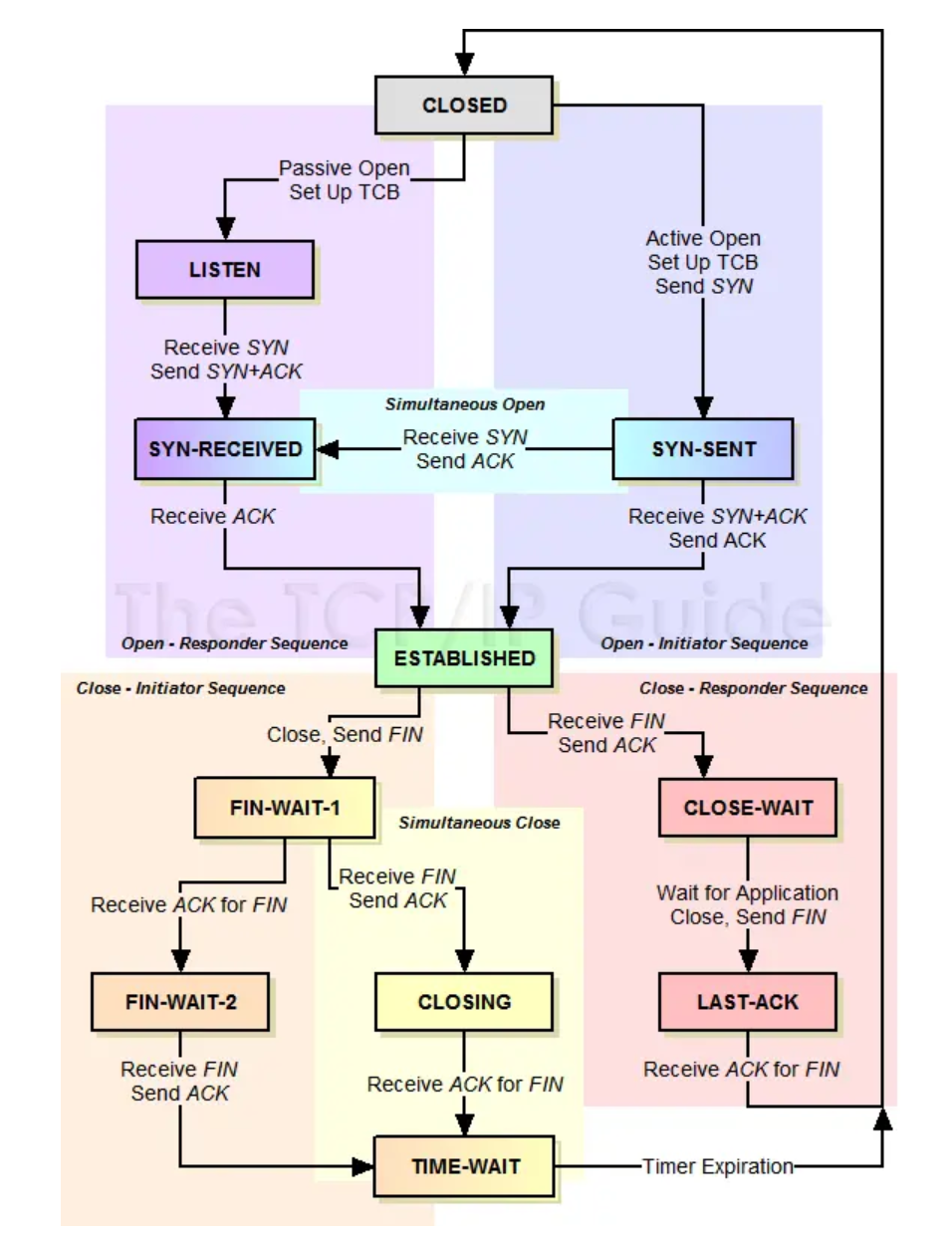


图3-5 连接关闭总流程

详细阐述如下：

1. 首轮信号（第一次挥手）：

客户端启动断开连接的流程，发送一个包含FIN标志（FIN=1）的报文段（序列号seq=u）。该操作使客户端状态转变为FIN\_WAIT\_1，同时停止数据发送，主动终止TCP连接。在四次挥手流程中，ACK报文负责确认接收，而SYN报文用于同步连接状态。

1. 服务端的响应确认（第二次挥手）：

服务端在接收到FIN报文后，会回复一个ACK报文，其序列号被设置为客户端序列号的下一个值（ack=u+1），以此确认已接收到客户端的断开请求。此时，服务端进入CLOSE\_WAIT状态，TCP连接进入半关闭状态，即仅允许从服务端向客户端的数据传输。客户端在收到此确认后，其状态转变为FIN\_WAIT\_2，准备接收来自服务端的断开信号。

1. 服务端的断开请求（第三次挥手）：

若服务端也决定关闭连接，它将模仿客户端的首次操作，但会同时设置FIN和ACK标志（FIN=1, ACK=1），并指定一个序列号（seq=w）。此时，服务端状态转变为LAST\_ACK（最终确认），等待客户端的最终确认。

1. 客户端的最终确认与等待（第四次挥手）：

客户端在接收到来自服务端的FIN|ACK报文后，会发送一个ACK报文作为回应，其序列号设置为服务端发送的FIN报文序列号的下一个值（ack=w+1）。此时，客户端进入TIME\_WAIT状态，通常等待2MSL（即两倍的最大报文生存时间），以确保服务端收到其ACK报文后已安全关闭连接。一旦计时结束，客户端也会进入CLOSED状态。这一过程中，TCP连接在客户端保持活动状态，直至计时结束。

（2）双方同时断开连接

1. 双方同时发起连接终止信号：

客户端与服务器几乎同时决定结束它们之间的通信会话，于是几乎同步发出FIN报文，各自随即进入FIN\_WAIT\_1的等待状态，预示连接的断开。

1. 相互确认对方的终止请求：

鉴于双方几乎同时发出FIN报文，这些报文在网络上交错传递。一旦接收到对方的FIN报文，双方都会迅速响应，通过发送ACK确认报文来告知对方其关闭请求已被成功接收；促使双方的状态从FIN\_WAIT\_1转变为一个更为接近终止的状态，可视为CLOSING状态。

1. 进入等待确保阶段：

完成ACK报文的交换后，为确保所有传输的数据包都能得到妥善处理，包括确认接收到的FIN报文，双方都将进入TIME\_WAIT状态。这一阶段通常持续2MSL的长度，让网络中可能存在的残留数据包有足够的时间得以处理，从而防止因ACK报文丢失而引发的连接状态不一致问题。

1. 彻底断开连接：

经过TIME\_WAIT状态设定的时间（即2MSL）后，双方确信所有必要的通信都已妥善处理，便会同时进入CLOSE状态，标志着它们之间的连接已经完全且安全地终止。

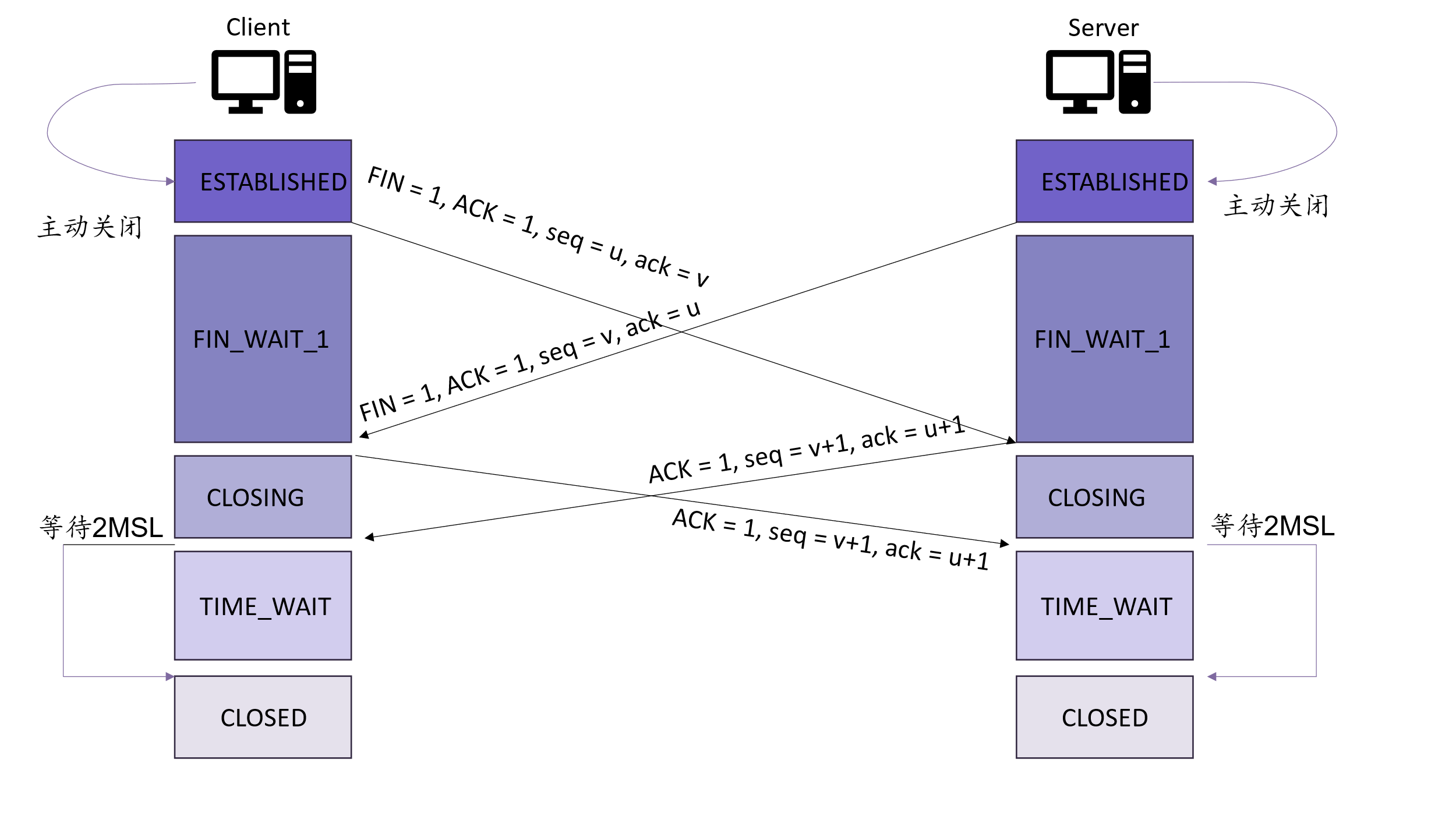


图3-6 双方同时关闭连接

## **3.3 可靠数据传输的设计**

说明TCP可靠数据传输的主要数据结构和协议规则。用FSM图表示主要工作流程。

说明发送端和接收端缓冲区的管理，滑动窗口的设计方法。

### 3.3.1主要数据结构

下面，依次介绍本章节用到的主要数据结构：

1. TCP 窗口：每个建立了连接的TCP都包括发送和接受两个窗口

**表4 TCP窗口结构体**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| typedef struct {  sender\_window\_t\* wnd\_send;  receiver\_window\_t\* wnd\_recv;  } window\_t; | 结构体  发送窗口  接收窗口 |

1. TJU\_TCP 结构体，用于保存TJU\_TCP用到的各种数据：

**表5 TJU\_TCP结构体**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| typedef struct { | 结构体 |
| int state; | TCP的状态 |
| tju\_sock\_addr bind\_addr; | 存放bind和listen时该socket绑定的IP和端口 |
| tju\_sock\_addr established\_local\_addr; | 存放建立连接后 本机的 IP和端口 |
| tju\_sock\_addr established\_remote\_addr; | 存放建立连接后 连接对方的 IP和端口 |
| pthread\_mutex\_t send\_lock; | 发送数据锁 |
| char\* sending\_buf; | 发送数据缓存区 |
| int sending\_len; | 发送数据缓存长度 |
| pthread\_mutex\_t recv\_lock; | 接收数据锁 |
| char\* received\_buf; | 接收数据缓存区 |
| int received\_len;  pthread\_cond\_t wait\_cond; | 接收数据缓存长度  可以被用来唤醒recv函数调用时等待的线程 |
| window\_t window; | 发送和接受窗口 |
| int close\_same; | 关闭标志初始化为0 |
| } tju\_tcp\_t; |  |

1. 发送窗口的结构体：

**表6 发送窗口的结构体**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| typedef struct {      uint16\_t window\_size;      uint32\_t base;      uint32\_t nextseq;      uint32\_t estimated\_rtt;      uint32\_t nextack;      int ack\_cnt;      pthread\_mutex\_t ack\_cnt\_lock;      struct timeval send\_time;      char\* retran[2];  int clk;  uint16\_t rwnd;  } sender\_window\_t; | 结构体  发送窗口大小  发送窗口后沿（最先发出去没ack的包号）  发送窗口前沿(下一个要发的包的序号)  预计RTT  下一个进入发送窗口的包的ack  对冗余ACK进行计数  锁  当前发出的最小序号的包的发送时间  用于存储需要重传的包  对当前发出的序号最小的包开始计时  接收窗口的余量大小  发送窗口 |

1. 接收窗口的结构体：

**表7 接收窗口结构体**

|  |  |
| --- | --- |
| **数据结构** | **含义或作用** |
| typedef struct {  char \* outoforder[MAX\_LEN];  uint32\_t expect\_seq;  } receiver\_window\_t; | 结构体  收窗口缓冲区  预期seq，判定是否收到应收到的seq  接收窗口 |

### 3.3.2协议规则

1. 滑动窗口与缓冲区的设计

1. 接收方
2. 有序接收的数据包：系统首先会将数据存储在接收缓冲区中。随后，系统会检查该数据分组是否已完全存在于缓存中，确认数据的完整性。接着，更新期望序列号expect\_seq，反映最新的接收状态。并计算当前可用的接收缓存空间，向发送方发送一个确认信号ACK，其中包含当前的expect\_seq。
3. 接收方收到期待值之后的数据包：若接收到的数据包序列号大于当前expect\_seq，系统会同样将它们存储在接收缓冲区中，但暂时不更新expect\_seq值，因为还有更早的数据包待接收。此时，系统计算可接收缓存的当前状态，并发送一个包含当前expect\_seq的冗余ACK，以通知发送方这些数据包已被接收但尚未按其顺序处理，同时提示发送方更新其接收窗口大小rwnd。
4. 接收方收到期待值之前的数据包：当接收到的数据包序列号小于当前expect\_seq时，这些数据包被视为超前到达。由于它们不符合当前的接收序列，系统会选择直接丢弃这些数据包，不进行进一步处理，以维护数据的有序性和完整性。
5. 发送方
6. 在发送方处理接收到的每个数据包时，根据ACK值的校验结果，动态调整base pointer的位置。此外，从报文头部中提取接收方通告的窗口大小advertise\_window，并据此更新发送方的接收窗口大小rwnd。
7. 每当发送方准备发送一个新的数据包时，它会递增下一个序列号nextseq的值，递增的量等于该数据包中包含的字节数，同时确保这一操作不会使nextseq超出当前发送缓冲区的容量限制。
8. 如果发送方收到了一个ACK，且该ACK的序列号大于当前的base，则一段数据已被成功接收。此时，发送方会更新其rwnd值，并将基准指针直接推进至ACK所指示的序列号，利用TCP的累积确认特性优化性能。
9. 若发送方接收到的ACK的序列号恰好等于当前的base，则为一个重复确认即冗余ACK。在此情况下，发送方会记录这些重复确认的次数。当此类确认累积达到特定阈值（如三次），根据TCP的快速重传算法，发送方将不等待超时，而是立即重传那些尚未收到确认的数据包。
10. 对于任何序列号小于当前base的ACK数据包，发送方将忽略这些数据包，因为它们指向的数据段已经被认为是旧数据或不再相关，直接进行丢弃处理。

下面，展示滑动窗口的工作示意图及不同序列号范围对应的数据包状态：

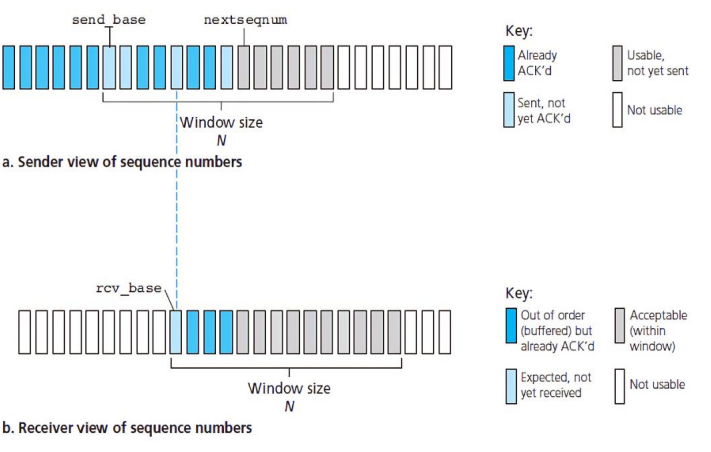


图3-7 滑动窗口的工作示意图

具体来说：

1. [0,base-1]中的序列号对应已经传输和确认的数据包；;
2. [base, nextseq-1] 对应已发送但尚未确认的数据包。
3. 如果数据从上层到达，则[nextseq,base+N-1] 中的序列号可用于可以立即发送的数据包。
4. 大于或等于 base+N 的序列号直到当前在流水线中的未确认数据包被确认后才能使用。

2. 其他标志位及重传机制的设计的说明

3.2.2节已经对ACK，FIN等标志位以及ack，seq等序列号的含义与关系做出了详细的解释，这里结合可靠数据传输进行进一步说明。

1. 序列号字段seq：TCP报文首部的序号字段值指本报文所发送的数据的第一个字节的序号。TCP 连接中的数据被视为一个虽无特定结构但保持严格顺序的字节序列，而序列标识则是基于这一连续字节流构建的，而非直接应用于报文段本身。
2. 确认号字段ack：指期待收到对方下一个报文段的第一个数据字节的序号。TCP 默认采用累积应答机制，即确认已无误接收至第一个缺失字节之前的所有数据。
3. 重传机制与超时：TCP 报文段的重传可由超时与冗余应答触发。TCP 仅为最久未获确认的数据包设置定时器，一旦超时，则仅重发该数据包，此过程称为超时重传。另一方面，若收到多个重复的确认（如连续三个对同一字节位置的ACK），则触发快速重传机制。
4. 遵循RFC文档，得知：

RTT：往返时间，指数据发送至其确认接收之间的时间差。

RTO：超时重传时间，是决定在何种时间点上执行重传的关键参数。

为获得更平滑的RTT估计值（SRTT），采用平滑因子x（推荐值介于0.8至0.9之间）进行计算：

进一步，RTO的计算涉及SRTT与RTT偏差（RTTVAL）的综合考量，其中t取1/8，h取1/4，具体公式为：

最终，

初始RTO建议设为1秒，遇超时则加倍，但每次接收到新报文并更新SRTT后，均按上述公式重新计算RTO。

下面，用直观的client端和server端的状态转换图来进行进一步的说明。

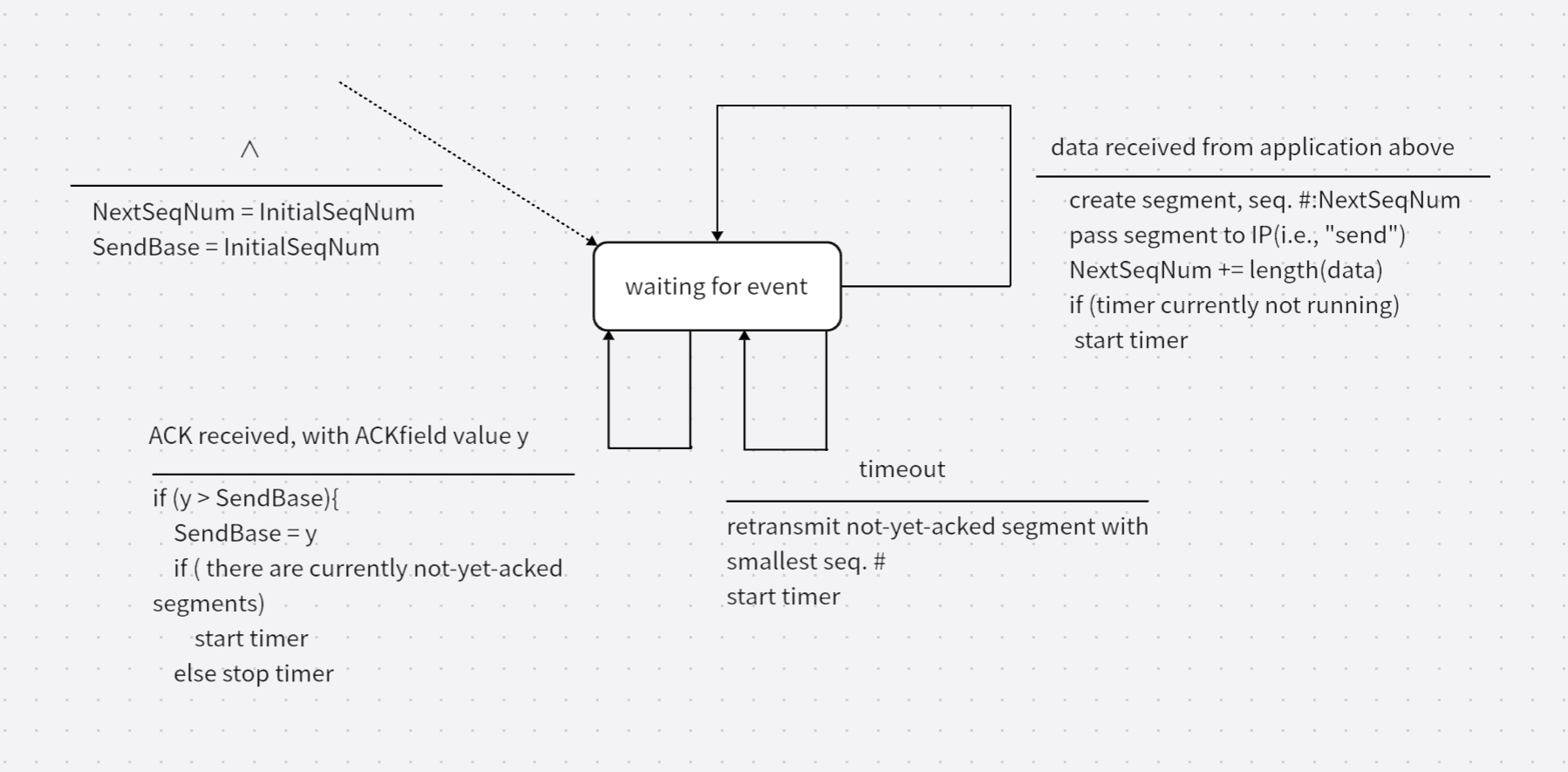


图3-8 client端的FSM图

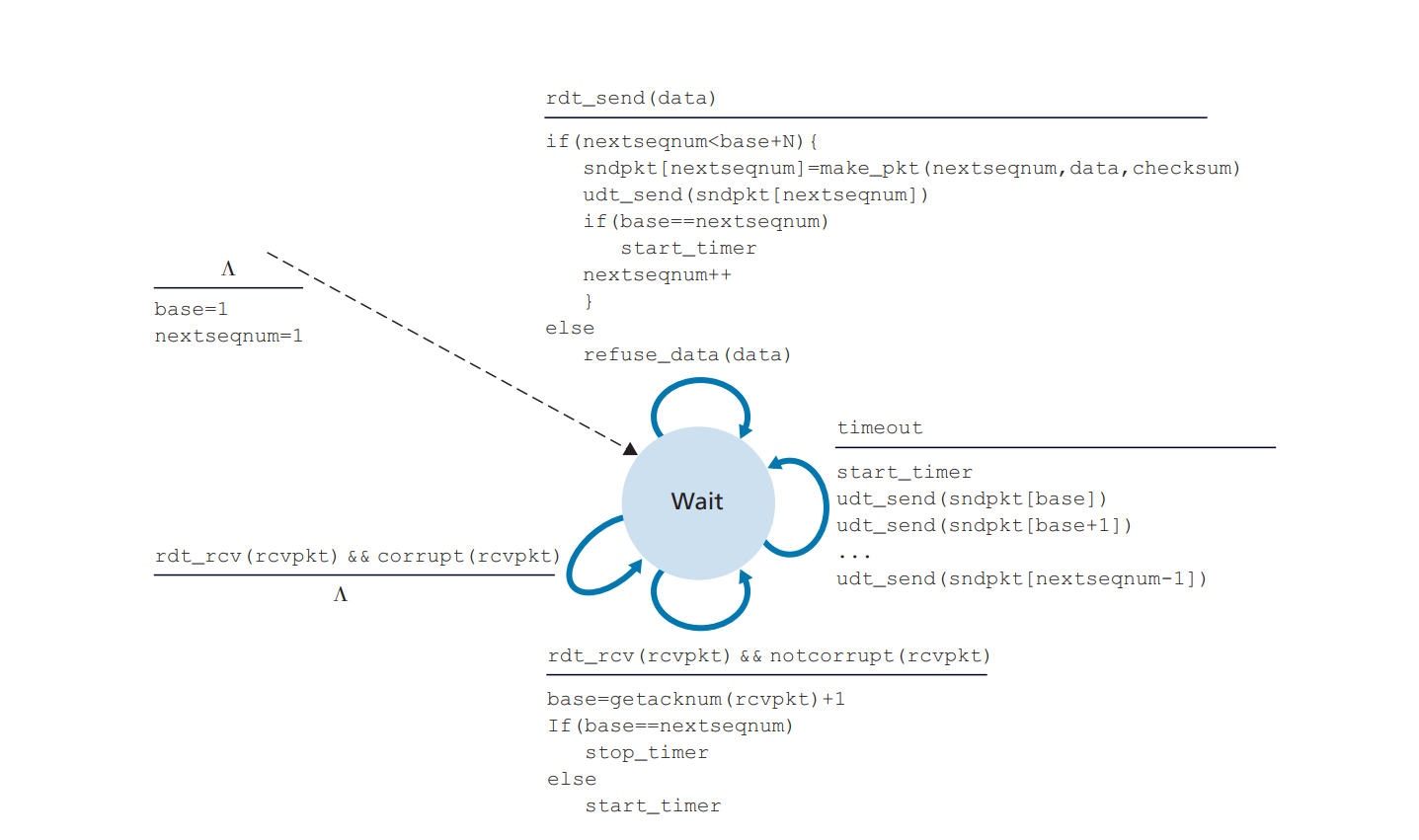


图3-9 server端的FSM图

## **3.4 流量控制的设计**

下面讲述流量控制的原理以及设计方法。

1. 滑动窗口通知机制

在数据传输过程中，为了优化发送效率并确保接收方处理能力不被过载，我们实施了一种基于接收方反馈的滑动窗口控制策略。具体而言，每当接收端的缓冲区容量发生变化时，它会通过特定字段（即窗口大小字段）向发送方报告其当前可用的缓冲空间大小，这一数值被称为“通告窗口”（advertised window size）。此机制的核心作用在于：

1. 自适应调节：若接收端应用层处理速度滞后，缓冲区内的数据未及时读取，则通告窗口的大小会相应缩减，以此作为发送速率上限的指示。
2. 零窗口处理：当通告窗口减至零时，发送方将暂停发送新数据，直至接收到窗口更新的通知。这一设计确保了即便在极端情况下，如接收端暂时无法接收更多数据，发送端也能有效避免数据溢出。

考虑到窗口大小通常以16位表示，我们设定了RWND\_SIZE的最大值为65535。在实际编程实现中，我在tju\_send函数中嵌入了循环检查逻辑，确保发送的未确认数据量不超过当前的rwnd值，从而实现了基于接收端缓冲状态的动态流量控制。若条件不满足，发送操作将保持阻塞状态，直到窗口大小允许继续发送。

1. 零窗口问题的探测与解决

TCP协议中，接收方在缓存满额时会向发送方报告rwnd=0，但若此后接收方无其他数据或确认信息需发送，即便其缓存空间已释放，也不会主动通知发送方。这则可能导致发送方长期处于等待状态，无法有效利用新增的接收空间。

为解决这一问题，我们引入了“零窗口探测”（Zero Window Probe, ZWP）机制：

1. 主动探测：在检测到通告窗口为零后，发送方会周期性地发送仅含少量数据（通常为单字节）的探测包，以尝试唤醒潜在的接收空间变化。
2. 响应与调整：接收方在收到这些探测包后会回复ACK，并附带最新的窗口大小信息。这一反馈机制使发送方能够及时了解接收端的状态变化。
3. 持续探测：零窗口探测将持续进行，直至接收方的窗口中出现可用空间，从而恢复正常的数据传输流程。

## **3.5拥塞控制的设计**

说明拥塞控制的原理、主要数据结构和协议规则。用FSM图表示主要工作流程。

# 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 连接管理的实现**

### 4.1.1 连接建立的实现

1. 实现connect和accept接口

|  |
| --- |
| tju\_accept(listen\_sock) |
| if全连接队列中无sock，则阻塞；  else  队列变更(-1)；  for(int i=0;i<MAX\_SOCK;i++)          if(acceptqueue[i]!=NULL)              // 将接受的连接放入已建立的连接表中  established\_socks[i] = acceptqueue[i];              acceptqueue[i]=NULL;  // 设置新连接为当前连接              new\_conn = established\_socks[i];              break; |

|  |
| --- |
| tju\_connect(sock, target\_addr) |
| 给sock绑定本地和目标ip址及端口  将建立了连接的socket放入内核 已建立连接哈希表中  // 发送SYN（第一次挥手）  Create packet;  sendToLayer3(syn, LEN);      sock->state = SYN\_SENT;  阻塞等待直到 socket 状态变为ESTABLISHED |

2. 状态转换的实现

连接建立针对三种状态：

1. socket的状态为LISTEN，由server端处理：

|  |  |
| --- | --- |
|  | sock->state == LISTEN**：** |
| 1 | if 收到不是SYN或者目的地不是server：忽视它； |
| 2 | else分配new\_sock建立连接 |
| 3 | 更改状态为SYN\_RECV； |
| 4 | 把new\_conn放到半连接队列； |
| 5 | 将建立了连接的 socket 放入内核全连接哈希表中； |
| 6 | server 端发送 syn+ack 包，进行第二次挥手。 |
| 7 | 此外，若数据包标志位为SYN\_FLAG\_MASK且当前状态SYN\_RECV： |
| 8 | 进行序号6的操作 |

1. socket的状态为SYN\_RECV，由client端处理：

|  |
| --- |
| sock->state == SYN\_RECV，server等待client的ACK： |
| 将半连接删除，放入全连接队列  状态变为ESTABLISHED  等待tju\_accept调用 |

1. socket的状态为SYN\_SENT，由client端处理：

|  |
| --- |
| sock->state == SYN\_SENT，client等待第二次握手的SYN|ACK **：** |
| if(dst!=sock->established\_local\_addr.port||flags!=ACK\_FLAG\_MASK+SYN\_FLAG\_MASK)  不是期待的SYN|ACK，忽略该数据包，由发送线程进行重发  将半连接删除，全连接放入  状态变为ESTABLISHED  返回ACK报文答复，client确认TCP建立成功 |

### 4.1.2 连接关闭的实现

从代码实现的角度考虑，我们不需要在意当前情况是两方同时发起关闭还是双方先后关闭。可以分析状态转换图、根据 socket 状态以及接收到的报文进行状态的转换以及处理：

1. tju\_close函数的实现

|  |
| --- |
| tju\_close (sock) |
| ①Client端调用close（）  当发送缓冲区内还有数据没发完，则阻塞；  // 发送 FIN  sendToLayer3(msg ,DEFAULT\_HEADER\_LEN);  sock->state = FIN\_WAIT\_1;  ②server端调用close（）  sleep(1);  这种情况为双方同时关闭  若此时状态不是CLOSED，则阻塞  释放sock |

2. 状态转换的实现

（1）socket的状态为ESTABLISHED

|  |  |
| --- | --- |
| 1 | If：get\_flags(pkt) == FIN\_FLAG\_MASK |
| 2 | 开始进行第二次挥手 |
| 3 | 发送包 |
| 4 | 变更状态，socket->CLOSE\_WAIT |
| 5 | server端在一段时间后发送 FIN |
| 6 | 开始进行第三次挥手 |
| 7 | 发送包 |
| 8 | 变更状态，socket->LAST\_ACK |

（2）socket的状态为FIN\_WAIT\_1

|  |  |
| --- | --- |
| 1 | If：flag==ACK\_FLAG\_MASK |
| 2 | 变更状态，sock->state = FIN\_WAIT\_2； |
| 3 | If: flags == ACK\_FLAG\_MASK +FIN\_FLAG\_MASK |
| 4 | 发送ack数据包 |
| 5 | 变更状态，sock->state = CLOSING; |

（3）socket的状态为FIN\_WAIT\_2

|  |  |
| --- | --- |
| 1 | If:dst!=sock->established\_local\_addr.port||flags!= ACK\_FLAG\_MASK+FIN\_FLAG\_MASK |
| 2 | 该数据包不是第四次挥手期待的FIN\_ACK，忽略 |
| 3 | If: flags == ACK\_FLAG\_MASK+FIN\_FLAG\_MASK |
| 4 | sendToLayer3(ack\_pkt ,DEFAULT\_HEADER\_LEN); |
| 5 | sock->state = TIME\_WAIT; |
| 6 | 一段时间后，sock->state = CLOSED; |

（4）socket的状态为CLOSING

|  |  |
| --- | --- |
| 1 | If：sock->state == CLOSING && get\_flags(pkt) == ACK\_FLAG\_MASK |
| 2 | sock->state = TIME\_WAIT; |
| 3 | 一段时间后，sock->state = CLOSED; |

（5）socket的状态为LAST\_ACK

|  |  |
| --- | --- |
| 1 | If：sock->state == LAST\_ACK && flags != ACK\_FLAG\_MASK |
| 2 | sock->state = CLOSED; |

下面，再用状态转换图表明server端和client端的状态变化：

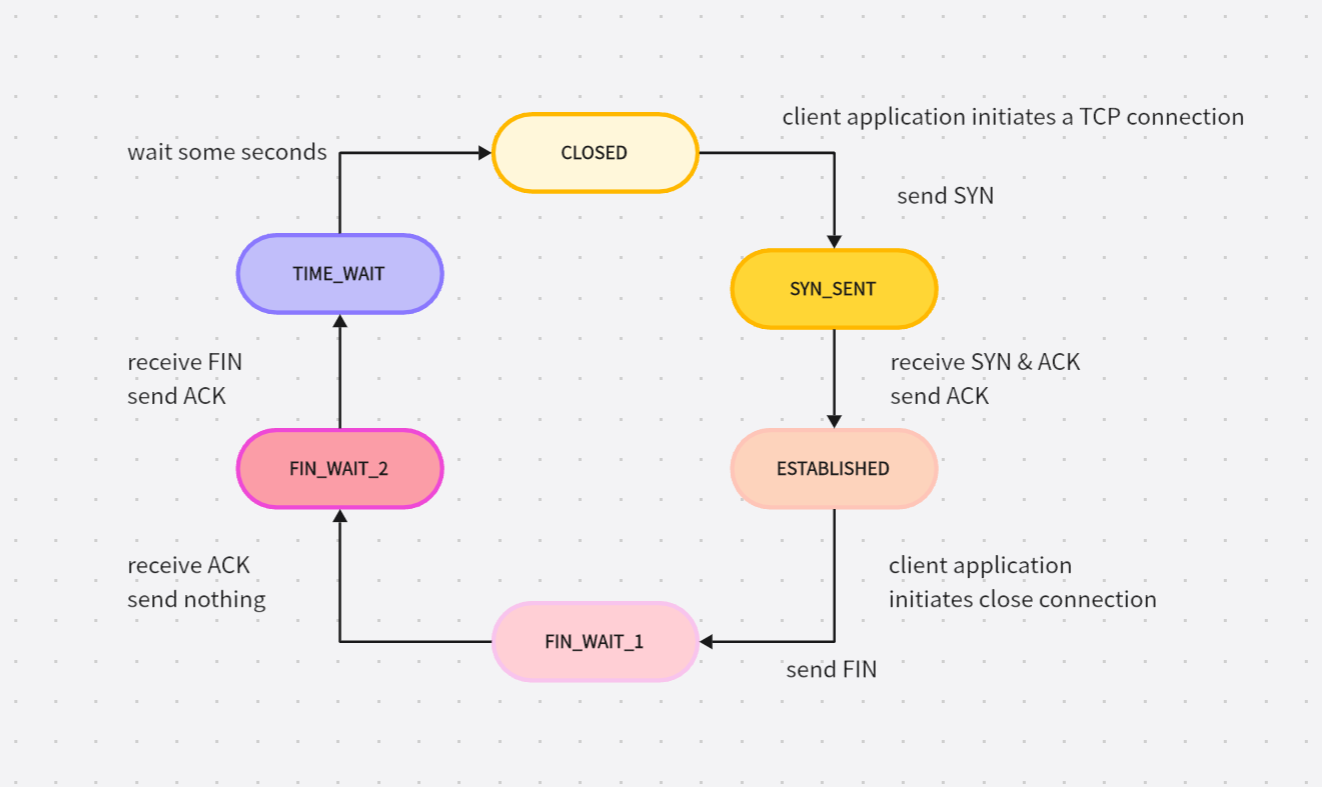


图4-1 Client端状态转换

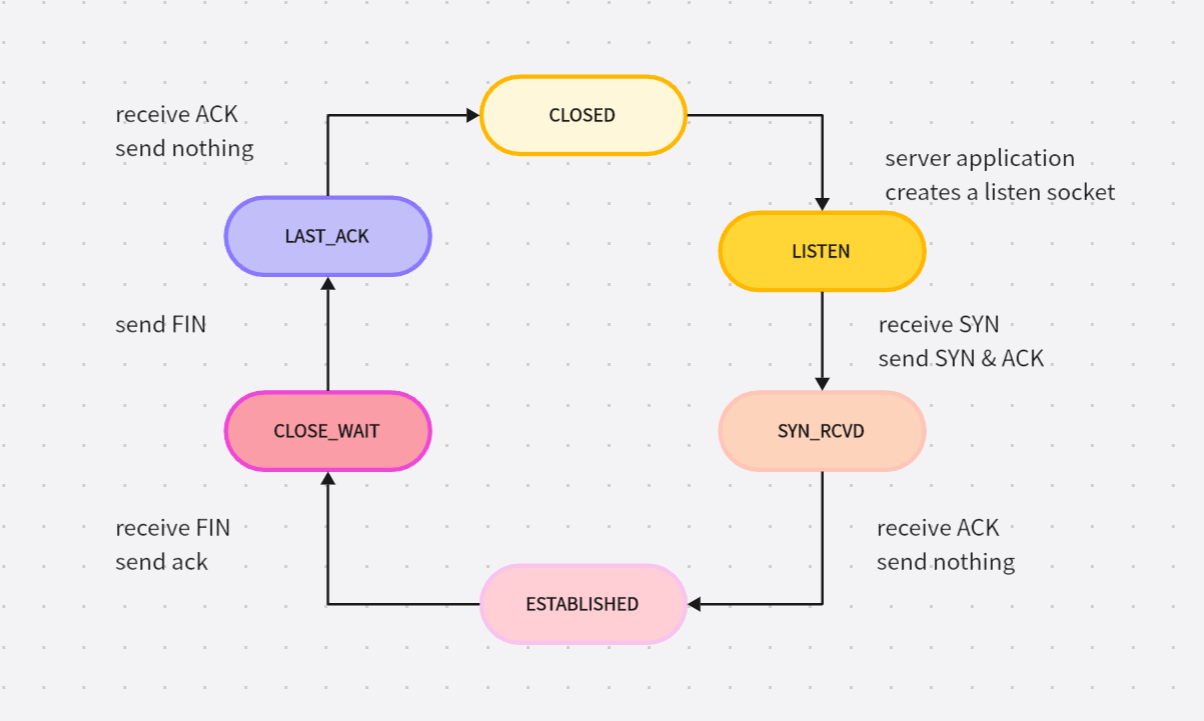


图4-2 server端状态转换

## **4.2 可靠传输的实现**

本人将缓冲区管理与滑动窗口的实现融入到原有连接管理的实现上，尽可能少新建函数。下面主要介绍需要用于实现可靠数据传输的函数伪代码及逻辑。其中，tju\_handle\_packet函数主要展现了可靠数据传输部分接收与发送缓冲区的管理以及滑动窗口的变更。

1. 对基础函数的补充与修改

（1）更改tju\_connect，在上周的基础上，在函数的末尾创建可以实现可能发生的超时重传的一个线程。

|  |
| --- |
| tju\_connect(sock, target\_addr) |
| …  pthread\_t id = 1500;  int rst = pthread\_create(&id, NULL, startTimer, (void\*)sock);  if (rst<0)       // ERROR open time thread       exit(-1); |

其中，startTimer用于超时重传，将在后面介绍。

（2）tju\_send函数

该函数的功能是将客户端发来的数据缓储存到客户端的发送缓冲区中。

|  |
| --- |
| tju\_send (sock, buffer, len) |
| If: 即将发送的数据长度<防止IP层分片的最大包内数据长度MAX\_DLEN  CreatePKTandsend(sock,buffer,len); // 直接打包发送  Else:  将数据按长度MAX\_DLEN分段，循环调用CreatePKTandsend发出 |

（3）tju\_recv函数

|  |
| --- |
| tju\_recv (sock, buffer, len) |
| 接收缓冲区为空时先阻塞  加锁  从到达的数据包中读取不大于len长度的数据  if : 接收缓冲区数据长度 > 要读的数据长度  将读的数据复制进buffer  修改接收缓冲区长度，记录实际所读的数据长度read\_len  接收缓冲区指针指向未读数据，释放已读数据所占空间  Else:  将读的数据复制进buffer，释放整个接收缓冲区，修改缓冲区长度为0  解锁  Return要读的数据长度 |

（4）补充tju\_handle\_packet函数连接建立（ESTABLISHED）后的部分，实现缓冲区的管理，与窗口的变更。

|  |
| --- |
| Case ESTABLISHED: |
| ①server  If: get\_flags(pkt)==ACK\_FLAG\_MASK(收到普通的包)  If: 序号小于期望的序号，即get\_seq(pkt)< sock->window.wnd\_recv-> expect\_seq  丢弃这个包，发送一个ack告知发送端期望序号  If: 序号等于期望的序号  接收这个包，放入缓冲区  If: 接收缓冲区已满  丢弃分组  If: 失序报文缓冲区为空  直接放入顺序缓存  Else(失序报文缓冲区有报文)  将失序缓冲区与该包序号连续的报文取出发送到顺序缓冲区  接下来恢复无序缓冲区，使得其依然从游标0开始存储报文  If: 序号大于期望的序号  放入失序缓冲区  If：get\_flags(pkt) == FIN\_FLAG\_MASK  开始进行第二次挥手  发送包，变更状态，socket->CLOSE\_WAIT  server端在一段时间后发送 FIN  开始进行第三次挥手  发送包，变更状态，socket->LAST\_ACK  ②client  If: get\_flags(pkt)==ACK\_FLAG\_MASK  If: get\_ack(pkt)==sock->window.wnd\_send->base  收到冗余ACK，计数ack\_cnt++  // 变更窗口大小  sock->window.wnd\_send->rwnd = get\_advertised\_window(pkt);  若计数=3，进入快速重传, ack\_cnt=0  // 开启新一轮计数  gettimeofday(&sock->window.wnd\_send->send\_time,NULL);  // 收到的ack大于等于最小未被确认的包的序号，累积确认  newseq = get\_ack(pkt);  更新发送窗口下沿为收到包的ack值  上锁  If：发送缓冲区只有一个包没确认  清空发送缓冲区  Else：清空已确认包所占空间  解锁  // 计算RTO  If：sock->window.wnd\_send->base==sock->window.wnd\_send->nextseq  结束超时重传的计时  else：  给序号最小的包开始计时  把最小包缓存下来用于重传  ack计数位ack\_cnt清0;  更新通告窗口值 |

2. 新增函数，用于超时重传

|  |
| --- |
| startTimer() |
| while(1)  if(sock->window.wnd\_send->clk==FALSE)  // 等待时钟启动  else  gettimeofday(&nowtime,NULL);  // 计算当前时间与上次发送时间差  If: Time>RTO //时间大于估计RTT  RTO加倍  重传存储的最小包  更新发送时间 |

## **4.3 流量控制的实现**

本人采用sock->window.wnd\_recv->rwnd作为动态更新的指标，在tju\_send函数执行期间，若rwnd空间不足以容纳待发送的数据包，则发送操作将保持等待状态，直至rwnd扩容至足以接收数据包为止。具体而言，这一过程通过以下循环实现：

|  |
| --- |
| while (sock->window.wnd\_send->nextseq - sock->window.wnd\_send->base + len >= sock->window.wnd\_send->rwnd); |

这样的设计巧妙地绕过了传统的零窗口检测机制。简而言之，只要rwnd作为发送或接收窗口结构体的一部分，且这些窗口结构体内置于TCP连接结构体之中，客户端便无需依赖发送零探测数据包来询问服务器的rwnd大小，因为rwnd作为连接状态的一部分，其值会根据应用层的数据读取情况由服务器实时更新，客户端则能直接读取到最新的rwnd值。

此外，TCP协议标准中确实存在专门用于记录rwnd大小的字段。若出于某些特定需求必须实现零窗口探测机制，我们可以在tju\_send函数的起始处添加一段额外的阻塞逻辑：

|  |
| --- |
| while (sock->window.wnd\_send->rwnd < len)  sendToLayer3(zeroWindowProbePacket);  // 这里使用了一个新的标识符，GET\_NEW\_RWND，以请求服务器更新rwnd值 |

随后，在tju\_handle\_packet函数中，我们需处理两种情形：一是当服务器接收到GET\_NEW\_RWND请求时，计算并返回当前的rwnd值给客户端；二是客户端接收到来自服务器的rwnd更新后，进行相应的值更新操作。通过上述步骤，我们可以灵活实现零窗口探测的功能。

相关伪代码如下：

|  |
| --- |
| case GET\_NEW\_RWND: |
| // 此时发送端认为rwnd已经不够接收了，所以持续发0个字节的探测报文，从而使对方收到RETURN\_NEW\_RWND，以此来确认新的rwnd值  接收缓存区加锁  int used\_sum = 0;  for(int i=sock->window.wnd\_recv->expect\_seq;i< sock->window.wnd\_recv-> LastByteRcvd; i++){  if(sock->data\_mark[i] == 1) used\_sum++;  }  temp\_rwnd = TCP\_RECVWN\_SIZE - (sock->window.wnd\_recv->expect\_seq - sock->window.wnd\_recv->LastByteRead) - used\_sum;  向对方发送ACK，并通知对方更新rwnd  解锁 |

|  |
| --- |
| case RETURN\_NEW\_RWND: |
| //发送方收到更新rwnd的报文段，然后更新自己的rwnd  发送缓存区加锁  sock->window.wnd\_send->rwnd = get\_advertised\_window(pkt);  解锁 |

## **4.4 拥塞控制的实现**

# 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 连接管理的功能测试与结果分析**

1. 本地测试与平台测评

本地测试establish和close均为100分：

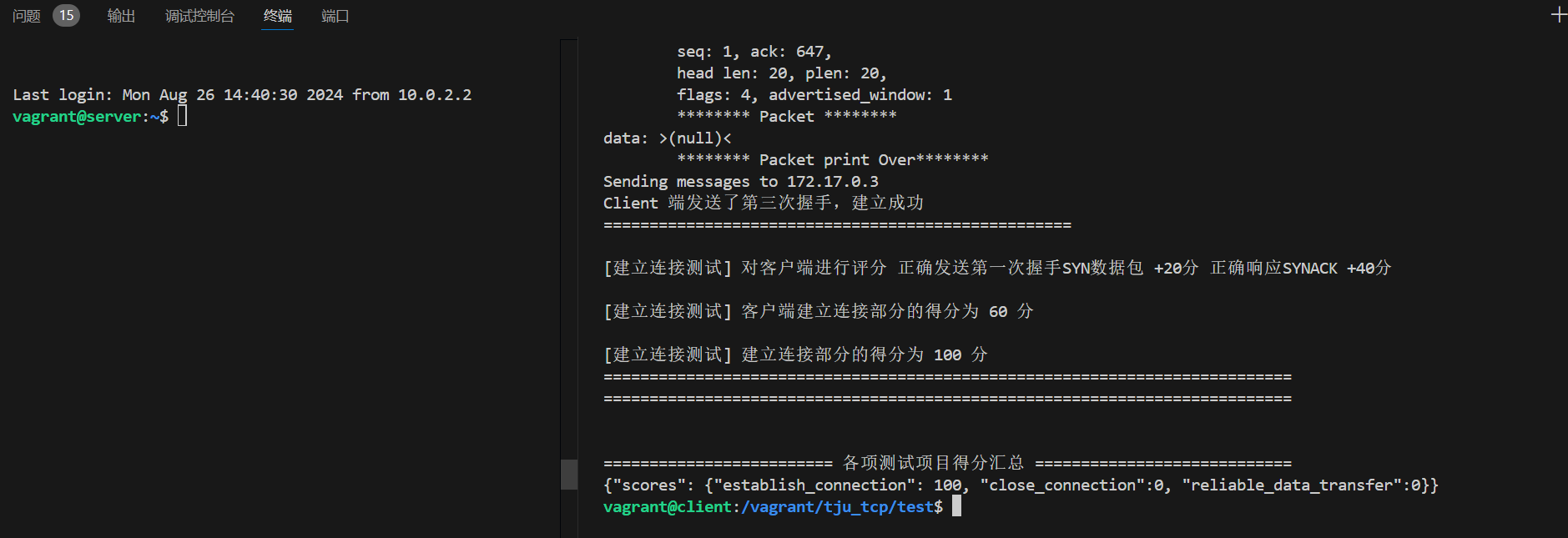


图5-1 连接建立本地结果

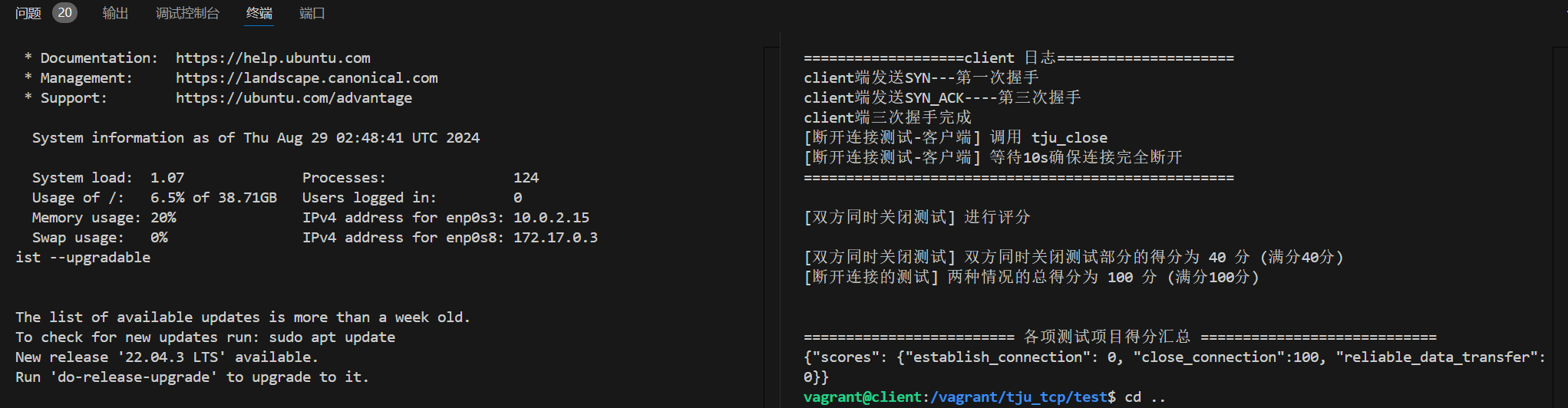


图5-2 连接关闭本地结果

平台测试：

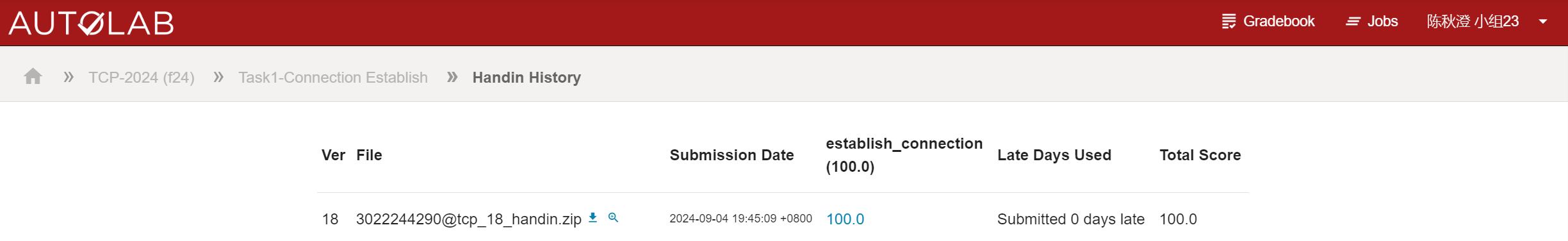


图5-3 连接建立平台结果

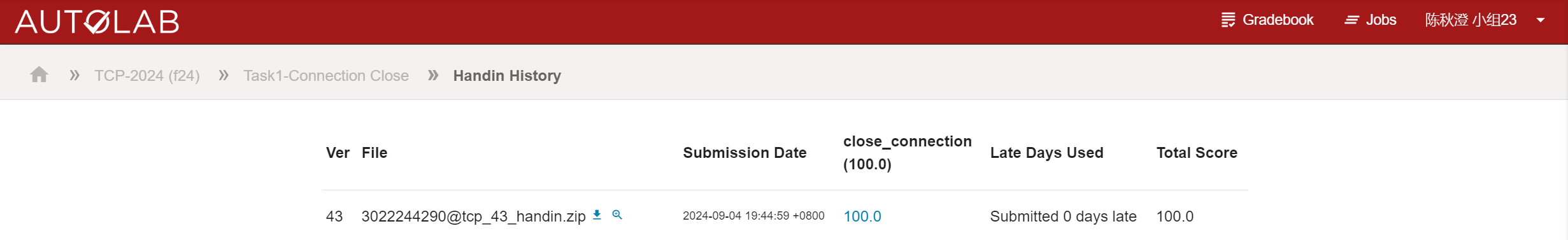


图5-4 连接关闭平台结果

2. 实验结果分析

通过测评结果，已完成TCP连接建立与关闭的模块，并实现日志打印，将状态实时打印，成功实现第一周的连接管理任务。

## **5.2 可靠传输的功能测试与结果分析**

1. 本地测试与平台测评

本地测试rdt为100分：

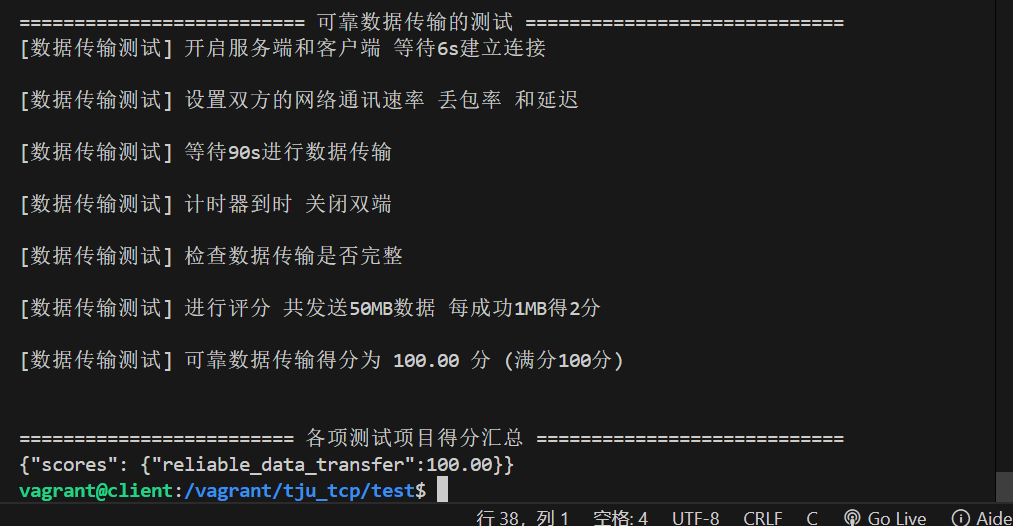


图5-5 可靠数据传输本地结果

平台测试100分：

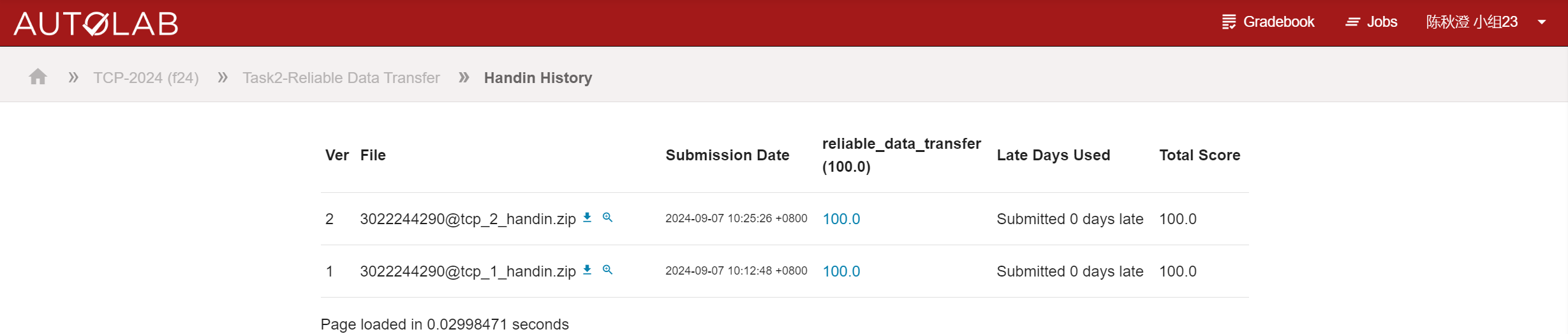


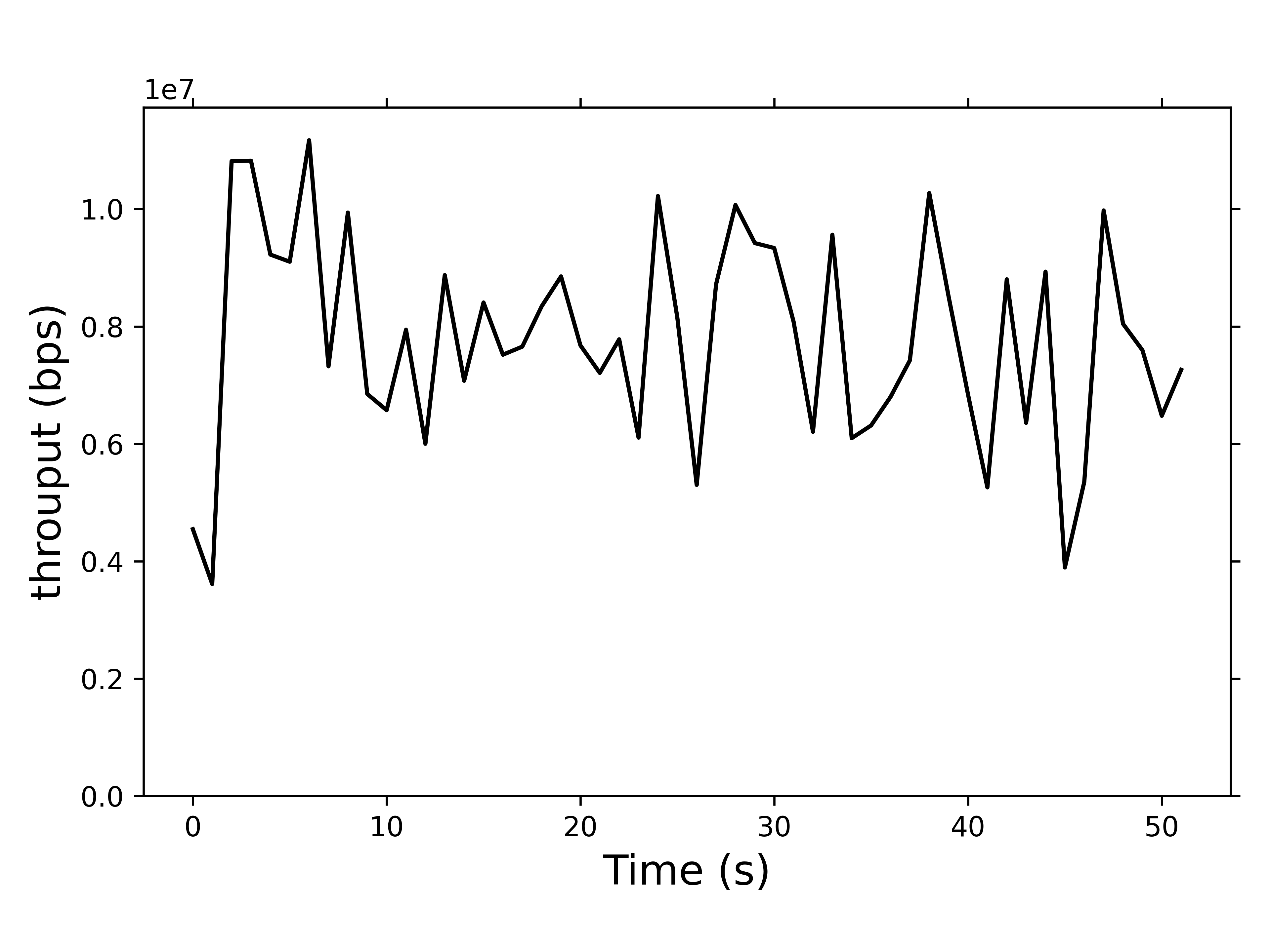
图5-6 可靠数据传输平台结果

2. 实验结果分析

通过测评结果，已完成TCP可靠数据传输的模块，并实现日志打印，将状态实时打印，成功实现第二周的可靠数据传输任务。

## **5.3 流量控制的功能测试与结果分析**

1. 结果图



2. 结果分析

结合所学可知：

1. 随着数据的流入，接收窗口的值会不断的减小；
2. 发送窗口的值随着接收窗口的值变化而相应的调整变化，且为滞后调整，因为收到 ACK 包后才会进行调整；
3. 当接收方处理数据过后，接收窗口的值会有一定的回升。

## **5.4 拥塞控制的功能测试与结果分析**

## **5.5 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。

## **6.1 问题与解决**

### 6.1.1 第一周

1. 软件安装

起初，本人的win11系统无法安装这种情况给定6.1.26版本的virtualbox，双击可执行文件后弹出如下指示：

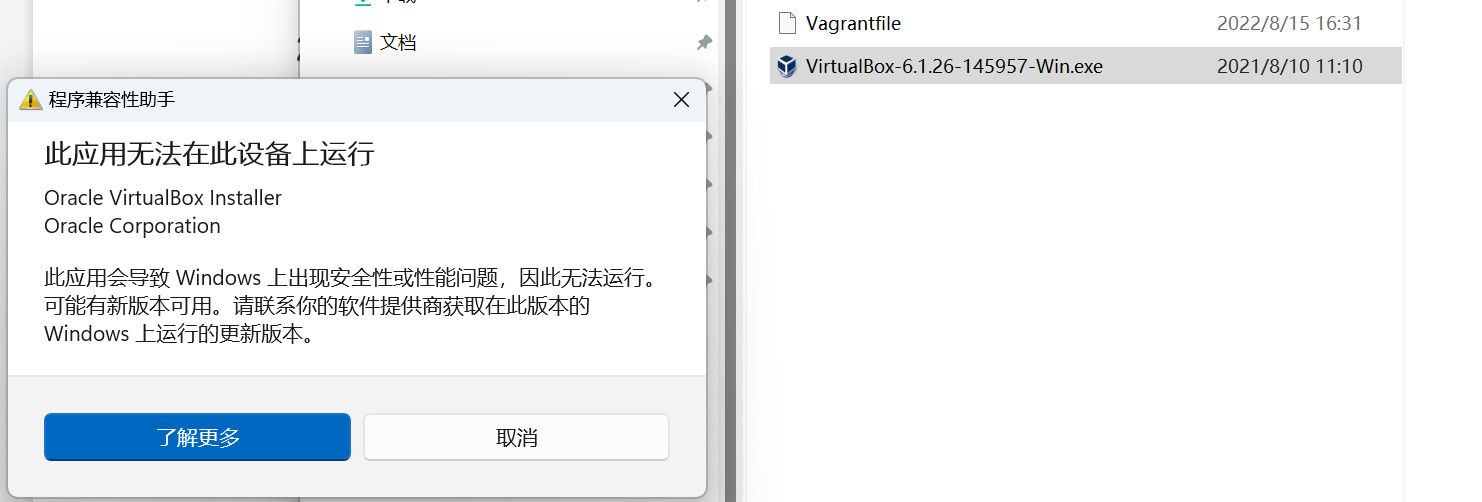


图6-1 第一周问题1

经查阅资料，在Windows设置中关闭“内存完整性”，重启设备即可继续安装。

2. 环境配置

在环境配置过程中，在添加box安装vagrant插件后试图运行vagrant up以开启虚拟机，但出现如下报错：

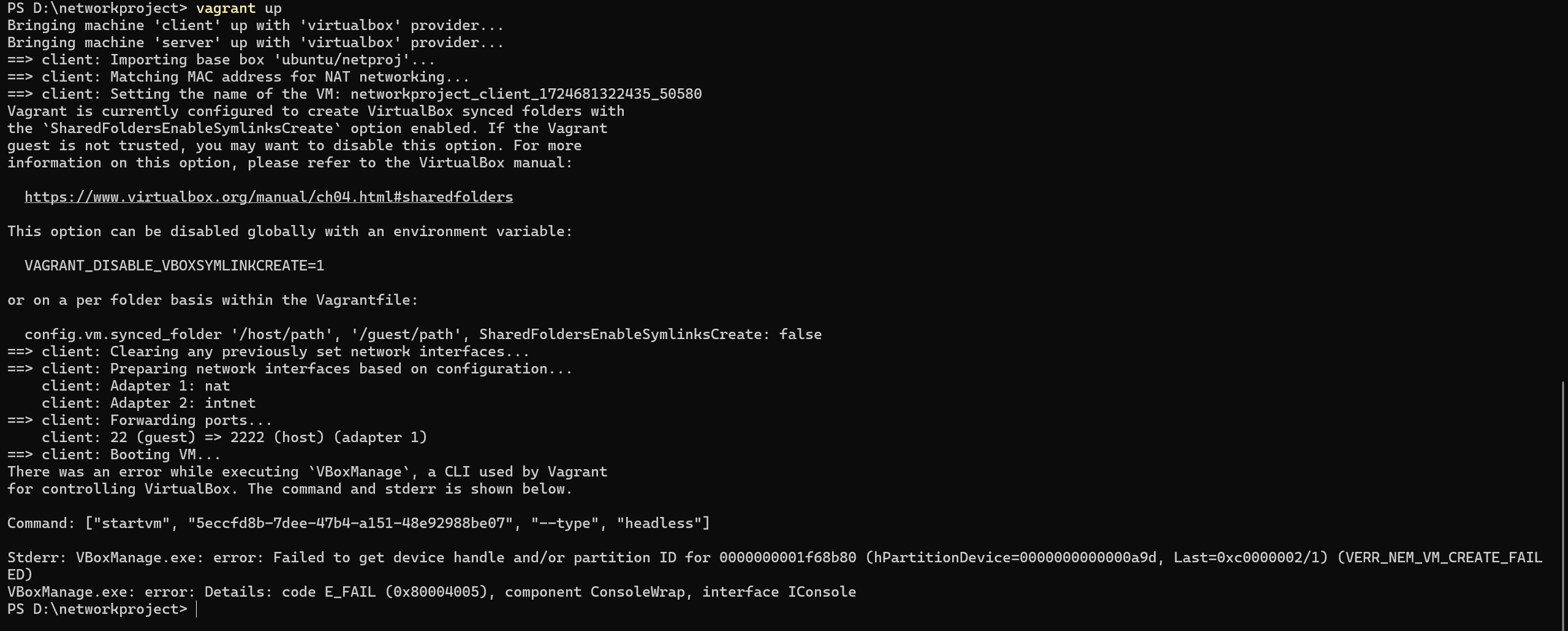


图6-2 第一周问题2

上网查找后发现没有相近问题的解决措施；幸运的是，经过尝试，用管理员身份开启终端，运行以下指令以关闭上学期计算机网络课程实践用到的Hyper-v，重启电脑后，该问题成功解决。

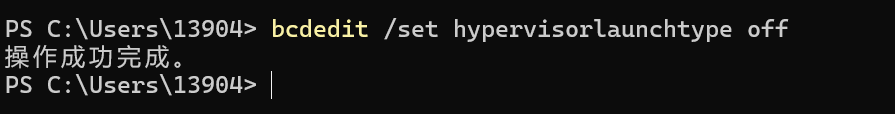


图6-3 第一周问题2的解决方法

3. 环境错误

误修改Vagrantfile文件中client端和server端的IP地址后会报错如下：

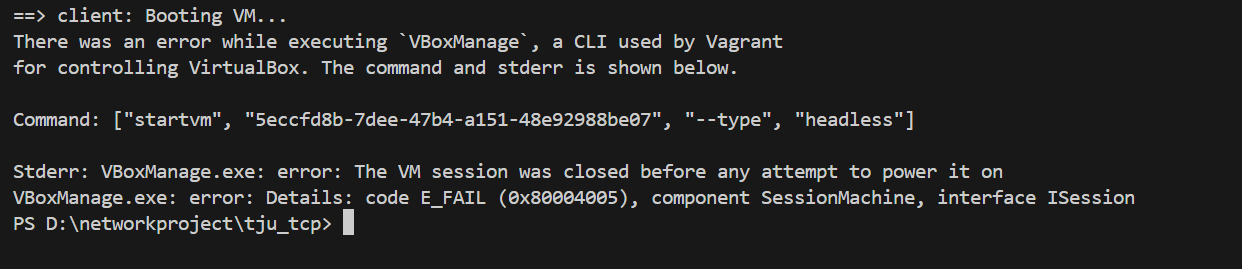


图6-4 第一周问题3的解决方法

但修改回原来的地址后，重新尝试运行vagrant up仍报错。

解决：打开任务管理器，关闭VirtualBox相关进程，重新运行指令，成功修复。

同时，明白了本地自动测试包中设置的虚拟机ip为：客户端172.17.0.2，服务端172.17.0.3。所以在使用本地自动测试包前，需在项目目录下的Vagrantfile文件以及tju\_tcp/src目录下的kernel.c文件中对client端和server端的ip地址进行修改。而基础框架其他文件中例如client.c和server.c中也有相关ip设置，不涉及本地自动测试。

4. 其他问题

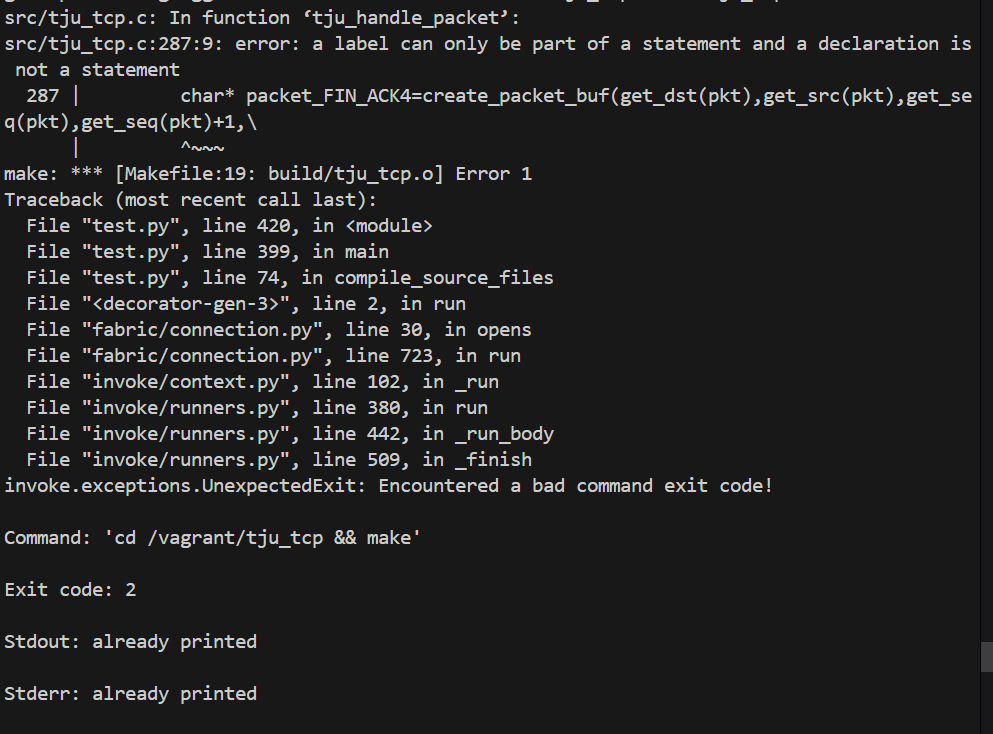


图6-5 第一周问题4的解决方法

原因是我在case之后进行变量的声明而没有将case下的语句以一个大括号包括。

分析：由于switch的几个case语句在同一个作用域（因为case 语句只是标签，它们共属于一个swtich语句块），所以如果在某个case下面声明变量的话，对象的作用域是在俩个花括号之间，也就是整个switch语句，其他的case语句也能看到，这样就可能出现错误。

解决：我们可以在case后面的语句加上大括号，明确声明的变量的作用域是仅在本case之中。

5. 其他问题

第一周遇到的问题主要是代码架构的分析与理解。这一周花了许多的时间梳理代码结构，理解任务背后的原理，但不清楚如何用代码实现。最后查阅了大量资料，复习C语言指针使用、与结构体等知识，学习了包括gettimeofday、fprintf等多个函数，一点点调试，尝试完成作业。

### 6.1.2 第二周

1. 关于代码自动评测的疑惑

第一周时，我将几个函数进行封装，按照助教老师的说明将测试程序的Makefile修改后将文件重命名为test\_Makefile, 放进打包好的handin.zip的tju\_tcp的目录下，本地测试满分通过，但多次提交到测试平台均显示0分且没有报错提示。但在完成第二周的作业时，我重新进行了完全相同的操作，提交至平台却可以满分通过，于是重新将代码提交到第一周的评测界面中，仍显示0分。目前我还是没明白其中的原因，留待接下来的时间探寻。

2. 指针的使用比较陌生

在调试过程中，出现了Segmentation fault段错误，经检查发现，原来是有空指针。与同学讨论后，将window\_t的两个指针进行初始化，解决了问题。此外，完成作业的过程中还出现了double free释放了两次指针等等问题，还是学习到了许多东西。希望在以后的学习与练习中，可以敢于运用指针，并且将其更熟练地掌握。

## **6.2 收获、体会及建议**