

天津大学

《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

本次计算机网络实践需要在 UDP 上实现 TCP 的基本功能。阅读学习 TCP 的 RFC 文档,在原有框架的基础上完善包括但不限于 TCP 标准的 `listen`、`connect`、`accept` 一系列接口,以实现连接管理、可靠数据传输、流量控制及拥塞控制、形成基本的 TCP。

首先是利用已知框架来构建 TCP 连接的初始化与终止流程,需要细致处理客户端与服务端间的交互时序,依据 TCP 三次握手与四次挥手的标准流程,为不同连接状态设计恰当的数据包响应策略。

在可靠数据传输方面,在发送与接收的窗口管理机制中引入动态变量,用以精细调控发送窗口的大小,从而有效管理发送与接收缓冲区的使用。此外,将实现超时重传机制以增强传输的可靠性,并探索快速重传技术以进一步优化性能。

对于流量控制,采用接收窗口大小作为发送的约束条件,确保数据传输的顺畅而不致过载。

拥塞控制方面,设计代码机制以完成出现超时等异常情况时触发拥塞状态的转换,并动态调整拥塞窗口大小及慢启动阈值,适应网络条件的动态变化,保持数据传输的稳定与高效。

经过自动平台测试与手动调试,每个阶段任务均按时完成并不断完善,并进行了性能测试与结果分析,收获良多。

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

3.1 总体设计

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：

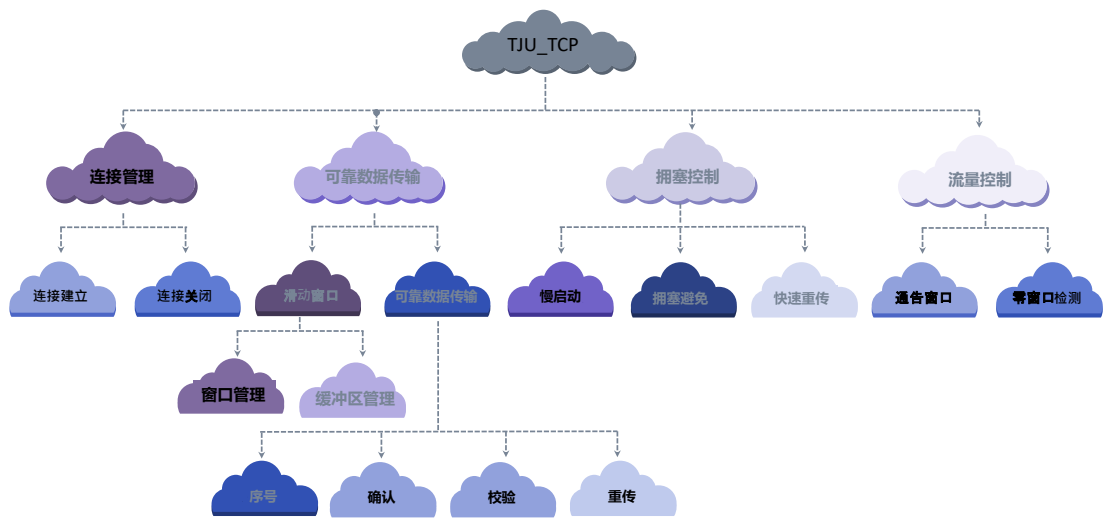


图 3-1 功能模块关系

总体而言，本实验的核心构建于连接维护机制与高效数据传输策略之上。具体而言，通过实现 `tju_listen`、`tju_accept`、`tju_connect`、`tju_send`、`tju_recv`、`tju_close` 等核心功能，完善了连接管理与可靠数据传输部分。在数据包处理模块 `tju_handle_packet` 中，细致地区分了 TCP 传输中发送端与接收端的不同运作状态，并据此实现了灵活的状态迁移与相应的数据包应答机制。尤其值得关注的是，为发送端特别设计了超时事件与三次重复 ACK 事件的应对逻辑。

基于前述的连接与传输机制，引入了流量调控与拥塞避免功能。在 `tju_send` 函数执行数据包发送时，发送端会根据当前的通告窗口尺寸动态决定是否采取缓冲策略。同时，接收端的反馈信息成为发送端调整拥塞状态、拥塞窗口及通告窗口大小的依据，这些窗口尺寸的调整又直接关联到发送窗口的即时变化，这也优化整体的网络传输性能。

3.2 连接管理的设计

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。

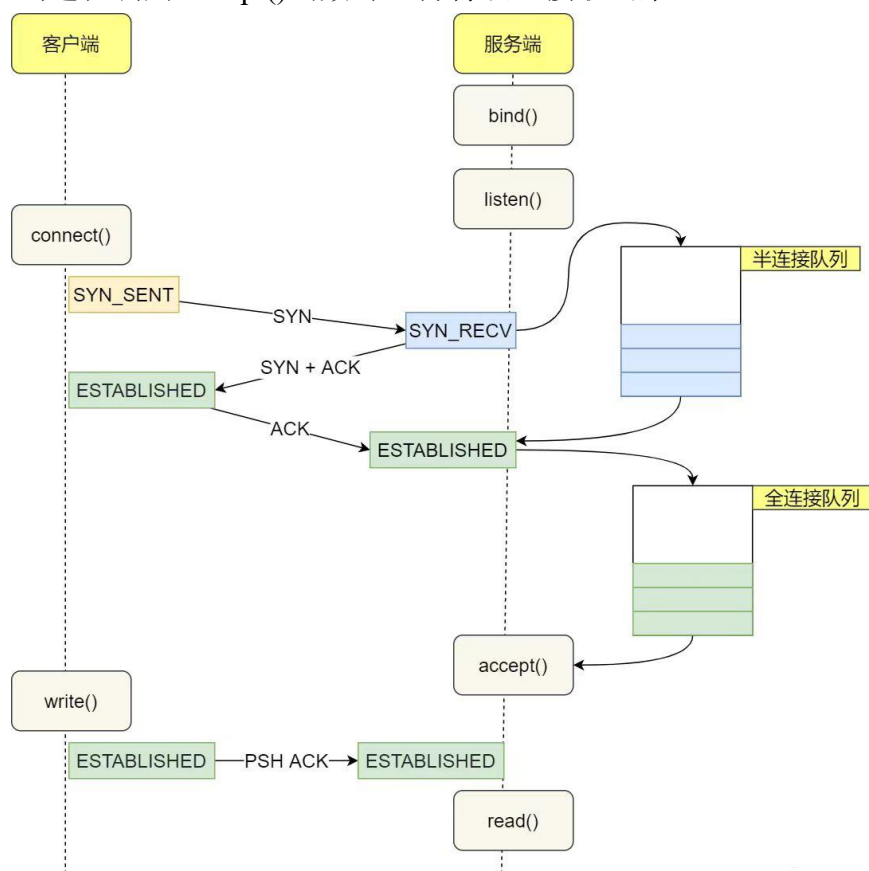


图 3-2 三次握手流程

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。

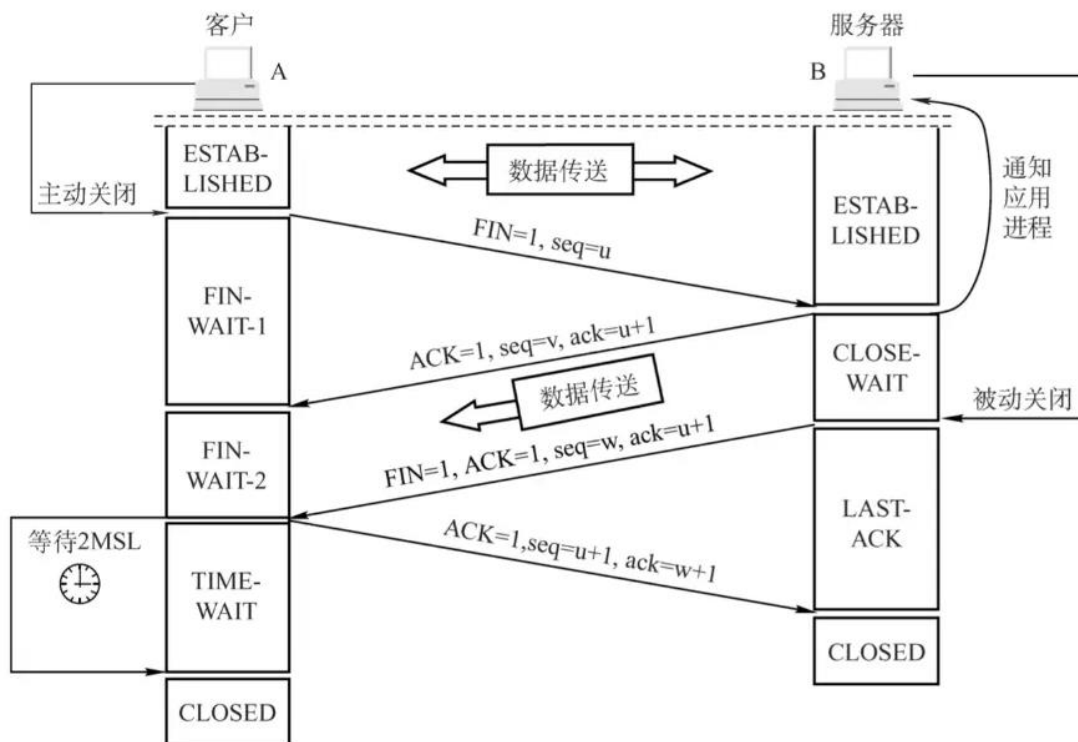


图 3-3 四次挥手流程

(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_SOCKET]	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_SOCKET]	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 {	结构体
uint32_t ip;	IP 地址
uint16_t port;	端口号
}tju_sock_addr;	

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：

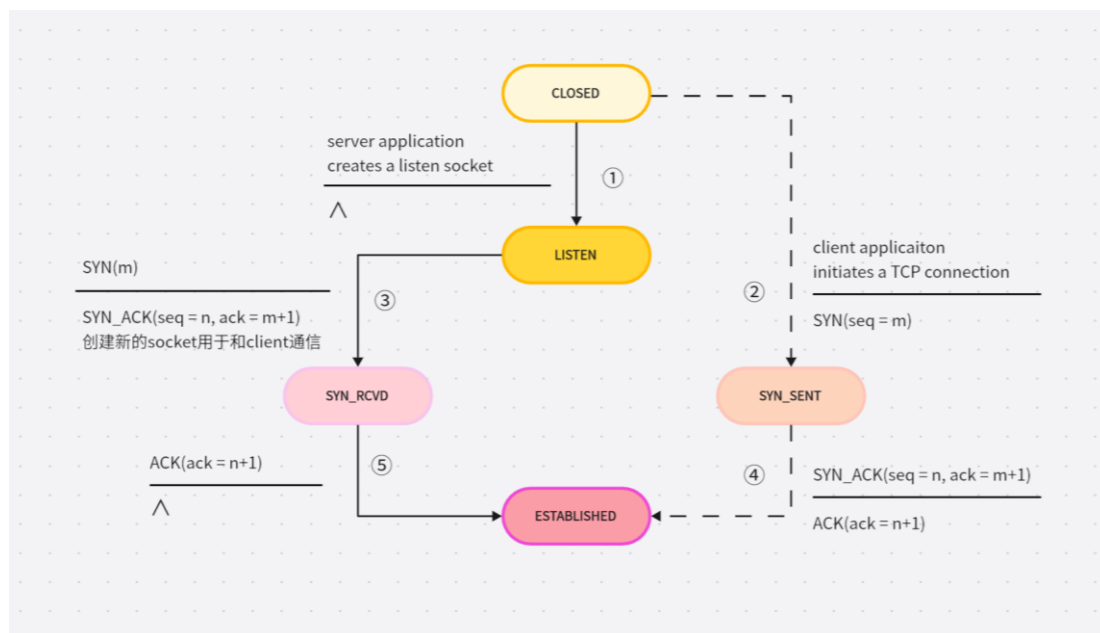


图 3-4 连接建立的状态机

(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。

a) 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。

b) 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。

c) 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。

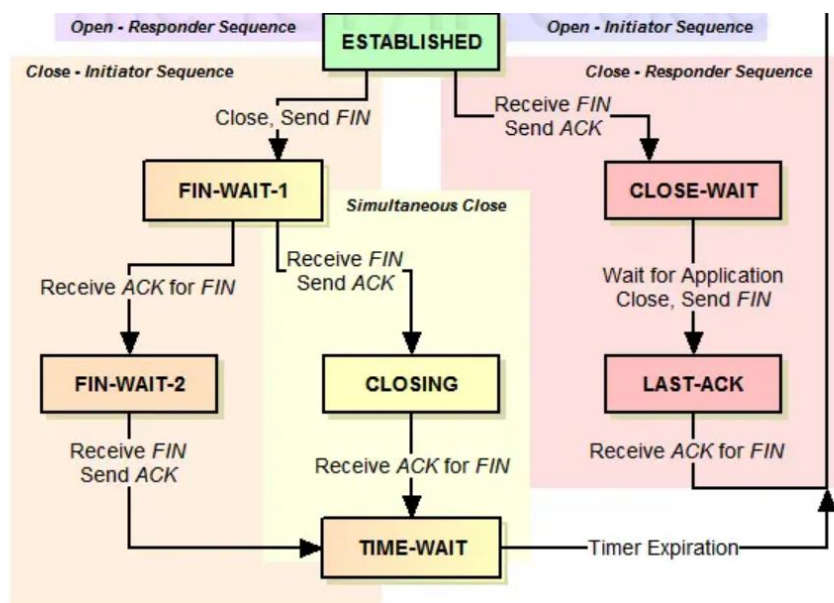


图 3-5 连接关闭总流程

详细阐述如下：

- 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。
 - 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。
 - 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。
 - 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
- 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。

- b) 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。
- c) 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。
- d) 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。

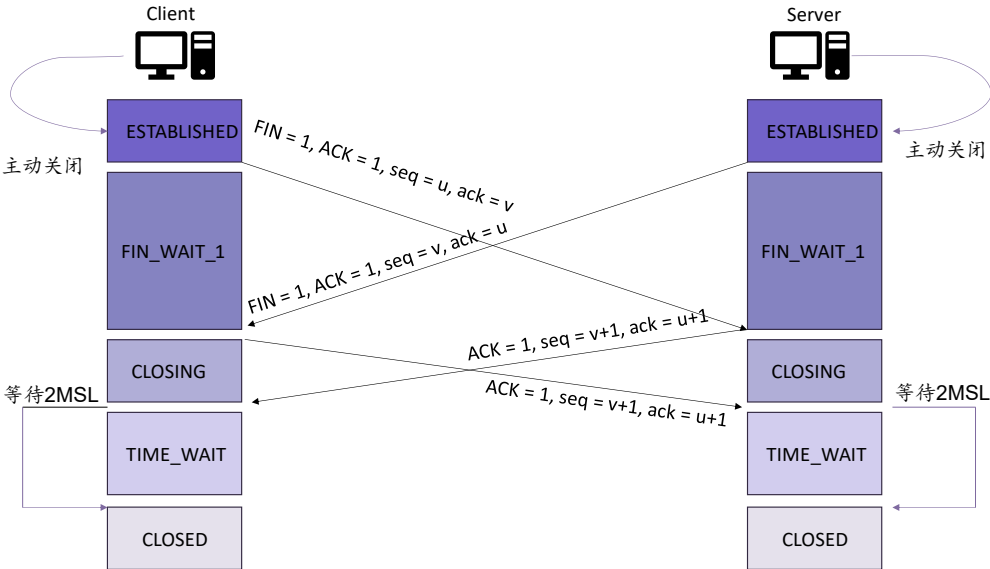


图 3-6 双方同时关闭连接

3.3 可靠数据传输的设计

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 {	结构体
sender_window_t* wnd_send;	发送窗口
receiver_window_t* wnd_rcv;	接收窗口
} window_t;	

2. 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 rcv_lock;	接收数据锁
char* received_buf;	接收数据缓存区
int received_len;	接收数据缓存长度
pthread_cond_t wait_cond;	可以被用来唤醒 rcv 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;	

3. 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 {	结构体
uint16_t window_size;	发送窗口大小
uint32_t base;	发送窗口后沿(最先发出去没 ack 的包号)
uint32_t nextseq;	发送窗口前沿(下一个要发的包的序号)
uint32_t estimated_rtt;	预计 RTT
uint32_t nextack;	下一个进入发送窗口的包的 ack
int ack_cnt;	对冗余 ACK 进行计数
pthread_mutex_t ack_cnt_lock;	锁
struct timeval send_time;	当前发出的最小序号的包的发送时间
char* retrans[2];	用于存储需要重传的包
int clk;	对当前发出的序号最小的包开始计时
uint16_t rwnd;	接收窗口的余量大小

} sender_window_t;	发送窗口
--------------------	------

4. 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 {	结构体
char * outoforder[MAX_LEN];	收窗口缓冲区
uint32_t expect_seq;	预期 seq, 判定是否收到应收到的 seq
} receiver_window_t;	接收窗口

3.3.2 协议规则

1. 滑动窗口与接收缓冲区的设计

(1) 接收方

- 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`。
- 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`。
- 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。

(2) 发送方

- 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`。
- 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。
- 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。
- 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。

- e) 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：

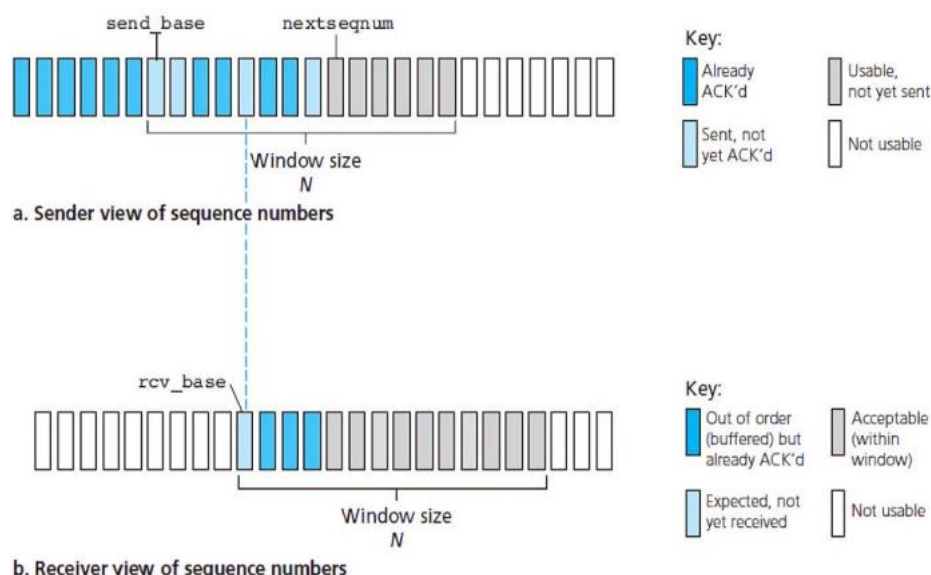


图 3-7 滑动窗口的工作示意图

具体来说：

- (1) $[0, \text{base}-1]$ 中的序列号对应已经传输和确认的数据包；
- (2) $[\text{base}, \text{nextseq}-1]$ 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 $[\text{nextseq}, \text{base}+N-1]$ 中的序列号可用于可以立即发送的数据包。
- (4) 大于或等于 $\text{base}+N$ 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：

$$sRTT = x * (1/sRTT) + (1 - x) * RTT$$

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

$$SRTT_{更新} = (1 - t) * SRTT + t * RTT_{测}$$

$$RTTVL = (1 - h) * RTTVL + h * |RTT_{测} - RTTVL|$$

最终，

$$RTO = SRTT + 4 * RTTVL$$

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。

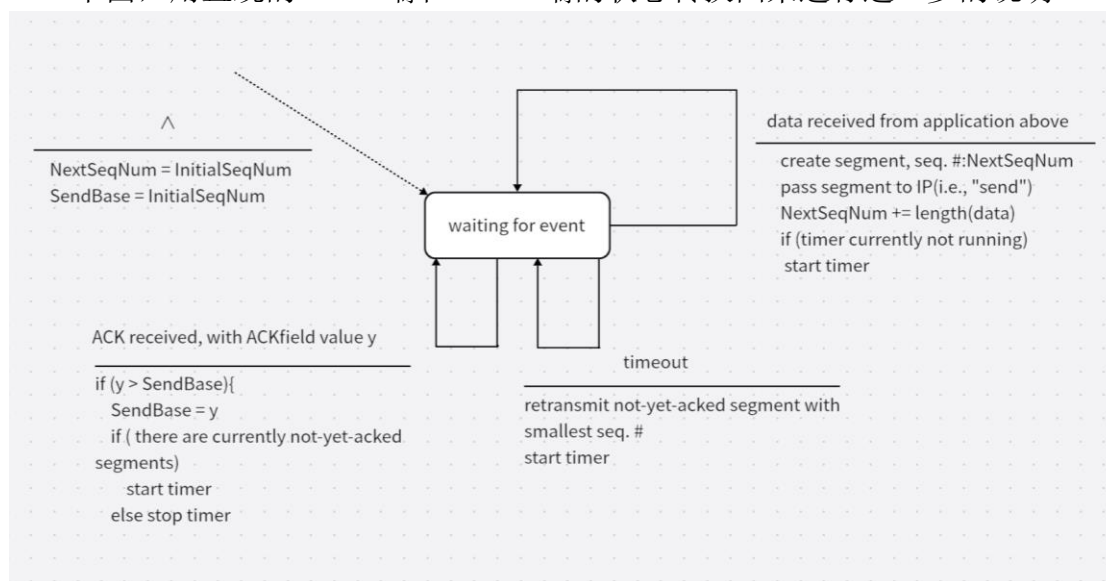


图 3-8 client 端的 FSM 图

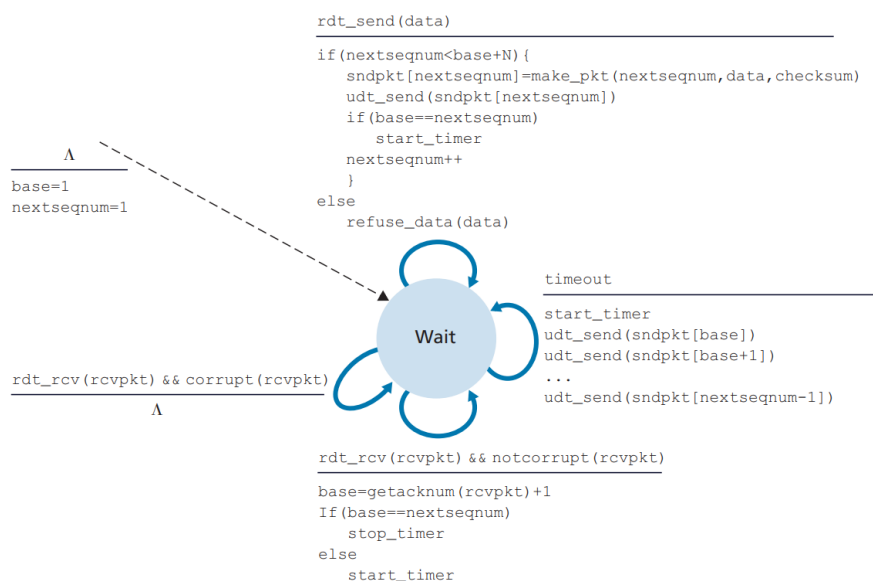


图 3-9 server 端的 FSM 图

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。

2. 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

3.5.1 原理

拥塞控制的原理在于识别网络状态：当超时现象发生时，网络通常正经历显著的拥塞；而冗余 ACK 信号的出现，则可能预示着网络中存在轻度的拥塞迹象。为了有效管理这种拥塞，TCP 协议的发送端采用拥塞控制策略，即通过维护一个称为拥塞窗口（congwin）的变量，来动态调整已发送但尚未获得确认的数据量上限。该机制允许发送端粗略地调控其向网络注入数据的速率，该速率大致可估算为拥塞窗口大小除以往返时间（RTT），即 $\text{rate} \approx \text{congwin}/\text{RTT}$ （字节/秒）。

此外，为了实现更为精细的拥塞与流量双重管理，TCP 还引入了发送窗口（swnd）的概念，它取拥塞窗口（cwnd）与接收窗口（rwnd）中的较小值。因此，这一设计确保了发送端在控制发送速率时，不仅考虑到了网络拥塞状况，还兼顾了接收端的处理能力，即发送端发送但尚未确认的数据量不得超过接收窗口的限制，从而同时满足了拥塞控制和流量控制的需求。

3.5.2 主要数据结构

下面展示新增主要数据结构。

表 8 拥塞控制部分新增主要数据结构

数据结构	含义或作用
Rwnd	接收窗口
swnd	发送窗口
Cwnd	拥塞窗口
ssthresh	慢启动阈值：若慢启动阶段 cwnd 达到这个值，将转换到拥塞避免模式
congestion_status	拥塞控制状态
SLOW_START	慢启动的状态值
CONGESTION_AVOIDANCE	拥塞避免的状态值
FAST_RECOVERY	快速恢复的状态值

发送方每收到来自接收方的一个数据包时，通过通告窗口读取 rwnd、根据当前的 cwnd 调整发送窗口的大小当前发送窗口大小为 $\min(\text{rwnd}, \text{cwnd})$ 。

3.5.3 协议规则

拥塞控制有三种状态：慢启动、拥塞避免、快速恢复。下面逐一介绍其特性。

1. 慢启动

在 TCP 连接初次建立后，系统会进入此阶段，此时拥塞窗口(cwnd)被初始化为 1 个最大报文段长度 (MSS)，因此发送速率起初非常缓慢，大约为 MSS/RTT 。然而，随着传输的进行，系统采用了一种加速策略，即每当收到一个确认应答 (ACK) 时，cwnd 的值就增加 1。这种增长方式使得在每个往返时间 (RTT) 周期内，cwnd 的值呈现指数级上升 (加倍)，从而逐步提升发送速率。

然而，一旦网络中出现数据包丢失 (如超时事件发生)，TCP 协议会迅速作出响应，通过调整策略来减缓发送速度，防止拥塞加剧。具体做法是：将慢启动阈值 (sssthresh) 设置为当前 cwnd 值的一半，并将 cwnd 重置为 1 个 MSS，重新开始慢启动过程。但这次的增长不再是无限制的，当 cwnd 的值增长到与 sssthresh 相等时，低速启动阶段结束，随后 TCP 会进入拥塞避免阶段，采用更为保守的速率增长策略来维护网络的稳定状态。

2. 拥塞避免

当 TCP 连接进入拥塞避免阶段后，拥塞窗口 (cwnd) 的增长速率会显著放缓，转变为平稳的线性增长模式。在此阶段，每经过一个往返时间 (RTT)，仅将 cwnd 的值增加一个最大报文段长度 (MSS)。这种调整旨在避免网络再次陷入拥塞状态，同时保持一定的数据传输效率。

3. 快速恢复

在 TCP 连接的慢启动或拥塞避免阶段中，若检测到连续三次接收到冗余 ACK 事件，系统会触发快速响应。此时，TCP 会首先调整慢启动阈值 (sssthresh) 为当前拥塞窗口 (cwnd) 的一半，并可将 cwnd 更新为 sssthresh 加上一个额外的固定值 (通常是 3，但也可不加)，随后进入快速恢复状态。

在快速恢复阶段，TCP 会根据网络的实际反馈来动态调整发送行为。如果在此阶段内未能接收到新的 ACK，TCP 将维持 cwnd 的线性增长趋势，继续谨慎地增加发送速率。然而，一旦接收到新的确认应答，则可说明网络状况有所改善，TCP 将退出快速恢复状态，并重新进入拥塞避免阶段，以更加稳健的方式管理数据传输，防止再次发生拥塞。

接着，用拥塞控制的 FSM 图来进一步阐述。其实不管处于哪种状态，只要出现 timeout 事件，立即进行以下操作： $sssthresh = cwnd/2$ ， $cwnd = 1 * MSS$ ，并转入慢启动阶段。此外，只要出现 3 次冗余 ACK，TCP 就会设置 $sssthresh = cwnd/2$ ， $cwnd = sssthresh + 3$ ，再转入快速恢复阶段。如图 3-10 所示：

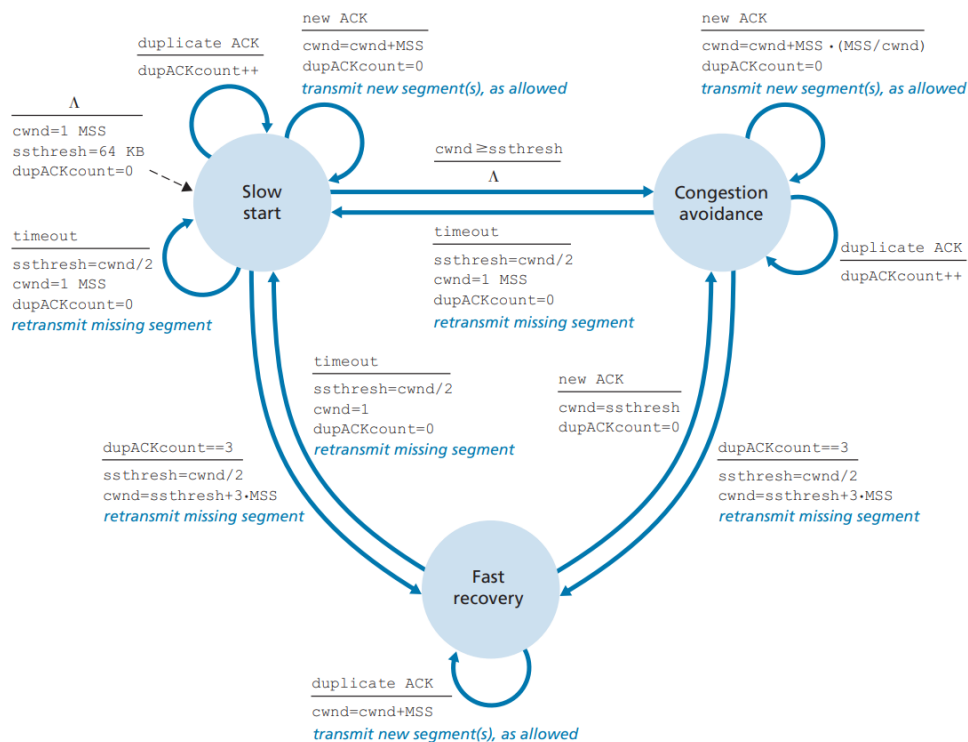


图 3-10 拥塞控制状态转换图

四、协议实现

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_SOCKET;i++)
        if(acceptqueue[i]!=NULL)
            // 将接受连接放入已建立连接表中
            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
```

(2) 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
```

(3) 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：

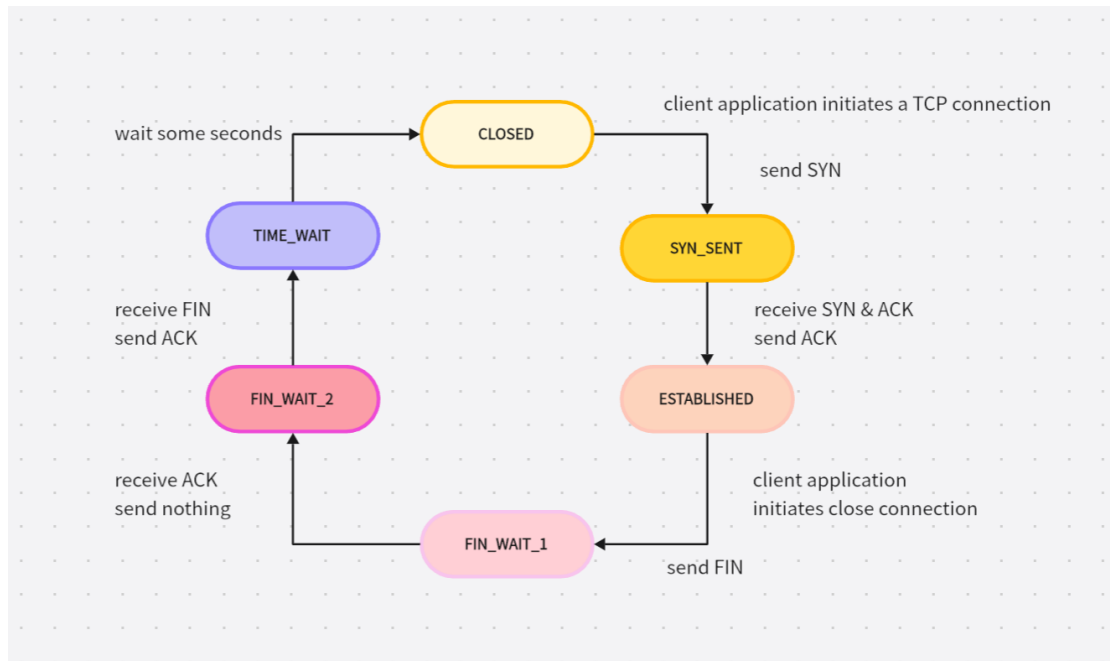


图 4-1 Client 端状态转换

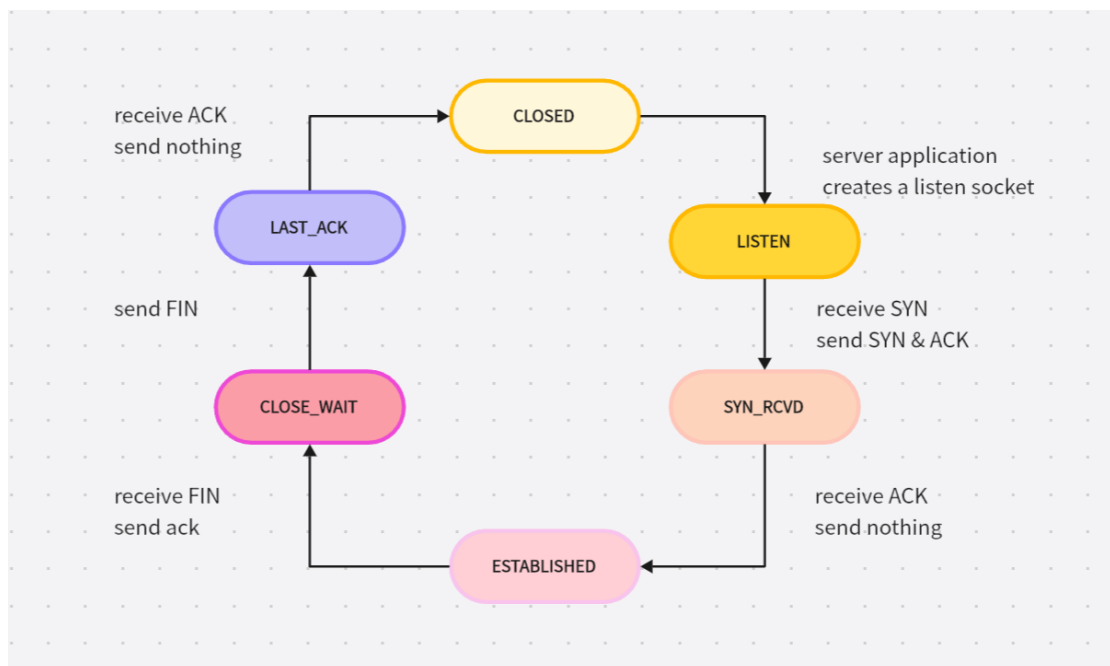


图 4-2 server 端状态转换

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_rcv->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_rcv->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_rcv->expect_seq;i< sock->window.wnd_rcv->
    LastByteRcvd; i++){
        if(sock->data_mark[i] == 1) used_sum++;
    }
    temp_rwnd = TCP_RECVWN_SIZE - (sock->window.wnd_rcv->expect_seq -
    sock->window.wnd_rcv->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

4.4.1 常规情况

```
If: sock->window.wnd_send->congestion_status == SLOW_START:
    // 慢启动
    sock->window.wnd_send->cwnd += MAX_DLEN;
    if: cwnd >= ssthresh
        congestion_status = CONGESTION_AVOIDANCE;
```

```
Else If: congestion_status == CONGESTION_AVOIDANCE:
    // 拥塞避免
    // cwnd 指数型增长
    cwnd+=MAX_DLEN*MAX_DLEN/ cwnd;
```

```
Else:
    // 快速恢复
    Cwnd = ssthresh;
    congestion_status = CONGESTION_AVOIDANCE;
```

4.4.2 冗余 ACK

对于冗余 ACK 的处理需针对当前拥塞状态进行不同的操作。

1. 拥塞状态为 SLOW_START
进行以下操作：
 - (1) $ssthresh = cwnd/2$
 - (2) $cwnd += 3 * MSS$
 - (3) 进入 FAST_RECOVERY
2. 拥塞状态为 CONGESTION_AVOIDANCE
 - (1) $ssthresh = cwnd/2$
 - (2) $cwnd += 3 * MSS$
 - (3) 进入 FAST_RECOVERY
3. 拥塞状态为 FAST_RECOVERY
 - (1) 重传该丢失数据包

三种情况的伪代码如下：

```
If: congestion_status = SLOW_START
```

```
    // 慢启动
    ssthresh = cwnd/2;
    cwnd = ssthresh + 3 * MAX_DLEN;
    congestion_status = FAST_RECOVERY;
```

```
Else If: congestion_status = CONGESTION_AVOIDANCE
```

```
    // 拥塞避免
    ssthresh = cwnd/2;
    cwnd = ssthresh + 3 * MAX_DLEN;
    congestion_status = FAST_RECOVERY;
```

```
Else:
```

```
    // 快速重传
    // 重传数据包
    sendToLayer3(retransmit_pkt, DEFAULT_HEADER_LEN + send_again_len);
```

4.4.3 超时

当 Timer 线程检测到超时事件的发生时，会执行以下操作，伪代码如下：

```
// 超时
```

```
    ssthresh = cwnd/2;
    cwnd = 1 * MSS;
    // 重传数据包
```

```
sendToLayer3(retransmit_pkt, DEFAULT_HEADER_LEN + send_again_len);
congestion_status = SLOW_START;
```

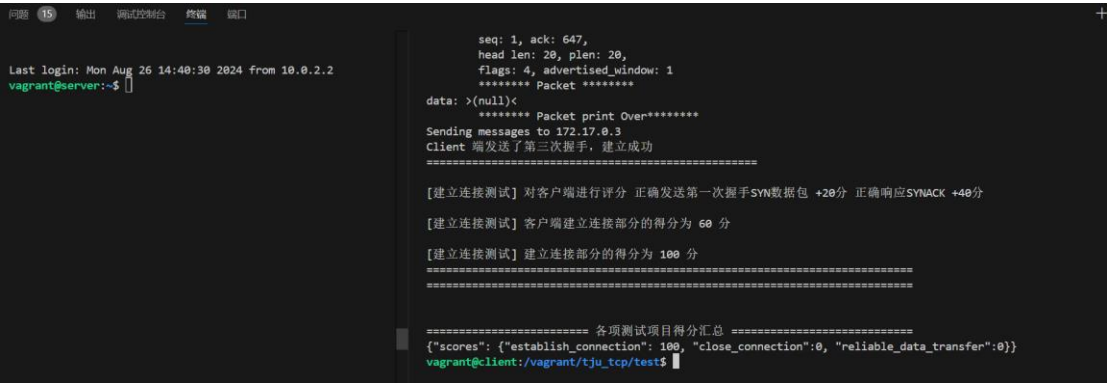
此外，为在完成拥塞控制任务的同时对流量控制部分进行优化，采用条件编译，优化代码，更为方便，也同时复习了遗忘的知识。

五、实验结果及分析

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

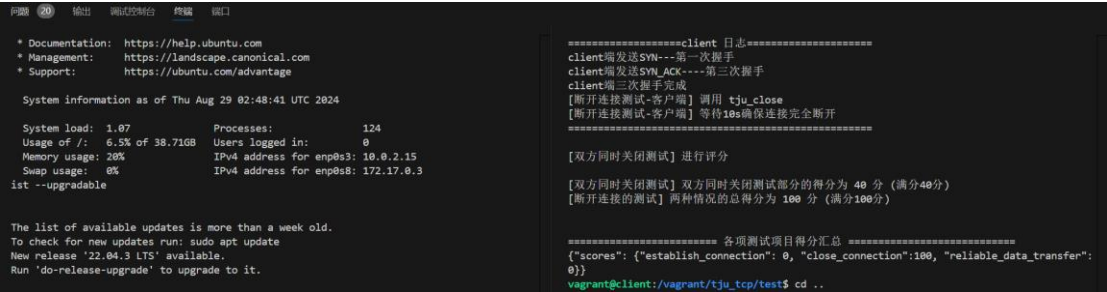
本地测试 establish 和 close 均为 100 分：



```
Last login: Mon Aug 26 14:48:30 2024 from 10.0.2.2
vagrant@server:~$

seq: 1, ack: 647,
head len: 20, plen: 20,
flags: 4, advertised_window: 1
***** Packet *****
data: >(null)<
***** Packet print Over*****
Sending messages to 172.17.0.3
Client 端发送了第三次握手，建立成功
=====
[建立连接测试] 对客户端进行评分 正确发送第一次握手SYN数据包 +20分 正确响应SYNACK +40分
[建立连接测试] 客户端建立连接部分的得分为 60 分
[建立连接测试] 建立连接部分的得分为 100 分
=====
===== 各项测试项目得分汇总 =====
{"scores": {"establish_connection": 100, "close_connection": 0, "reliable_data_transfer": 0}}
vagrant@client:/vagrant/tju_tcp/test$
```

图 5-1 连接建立本地结果



```
* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage

System information as of Thu Aug 29 02:48:41 UTC 2024

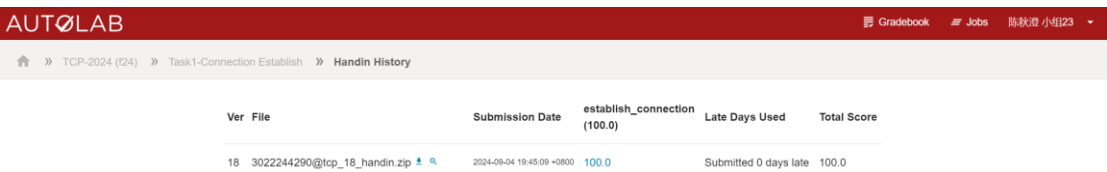
System load: 1.07 Processes: 124
Usage of /: 6.5% of 38.71GB Users logged in: 0
Memory usage: 20% IPv4 address for enp0s3: 10.0.2.15
Swap usage: 0% IPv4 address for enp0s8: 172.17.0.3
ist --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update
New release '22.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

=====client 日志=====
client端发送SYN---第一次握手
client端发送SYN_ACK----第三次握手
client端三次握手完成
[断开连接测试-客户端] 调用 tju_close
[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)
===== 各项测试项目得分汇总 =====
{"scores": {"establish_connection": 0, "close_connection": 100, "reliable_data_transfer": 0}}
vagrant@client:/vagrant/tju_tcp/test$ cd ..
```

图 5-2 连接关闭本地结果

平台测试：



AUTOLAB					Gradebook	Jobs	陈秋澄 小组23
TCP-2024 (t24) » Task1-Connection Establish » Handin History							
Ver	File	Submission Date	establish_connection (100.0)	Late Days Used	Total Score		
18	3022244290@tcp_18_handin.zip	2024-09-04 19:45:09 +0800	100.0	Submitted 0 days late	100.0		

图 5-3 连接建立平台结果

AUTOLAB						Gradebook	Jobs	陈秋澄 小组23
TCP-2024 (t24) » Task1-Connection Close » Handin History								
Ver	File	Submission Date	close_connection (100.0)	Late Days Used	Total Score			
43	3022244290@tcp_43_handin.zip	2024-09-04 19:44:59 +0800	100.0	Submitted 0 days late	100.0			

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：

```
===== 可靠数据传输的测试 =====
[数据传输测试] 开启服务端和客户端 等待6s建立连接

[数据传输测试] 设置双方的网络通讯速率 丢包率 和延迟

[数据传输测试] 等待90s进行数据传输

[数据传输测试] 计时器到时 关闭双端

[数据传输测试] 检查数据传输是否完整

[数据传输测试] 进行评分 共发送50MB数据 每成功1MB得2分

[数据传输测试] 可靠数据传输得分为 100.00 分 (满分100分)

===== 各项测试项目得分汇总 =====
{"scores": {"reliable_data_transfer":100.00}}
vagrant@client:/vagrant/tju_tcp/test$
```

图 5-5 可靠数据传输本地结果

平台测试 100 分：

AUTOLAB						Gradebook	Jobs	陈秋澄 小组23
TCP-2024 (t24) » Task2-Reliable Data Transfer » Handin History								
Ver	File	Submission Date	reliable_data_transfer (100.0)	Late Days Used	Total Score			
2	3022244290@tcp_2_handin.zip	2024-09-07 10:25:26 +0800	100.0	Submitted 0 days late	100.0			
1	3022244290@tcp_1_handin.zip	2024-09-07 10:12:48 +0800	100.0	Submitted 0 days late	100.0			

Page loaded in 0.02998471 seconds

图 5-6 可靠数据传输平台结果

接下来展示往返时延变化图：

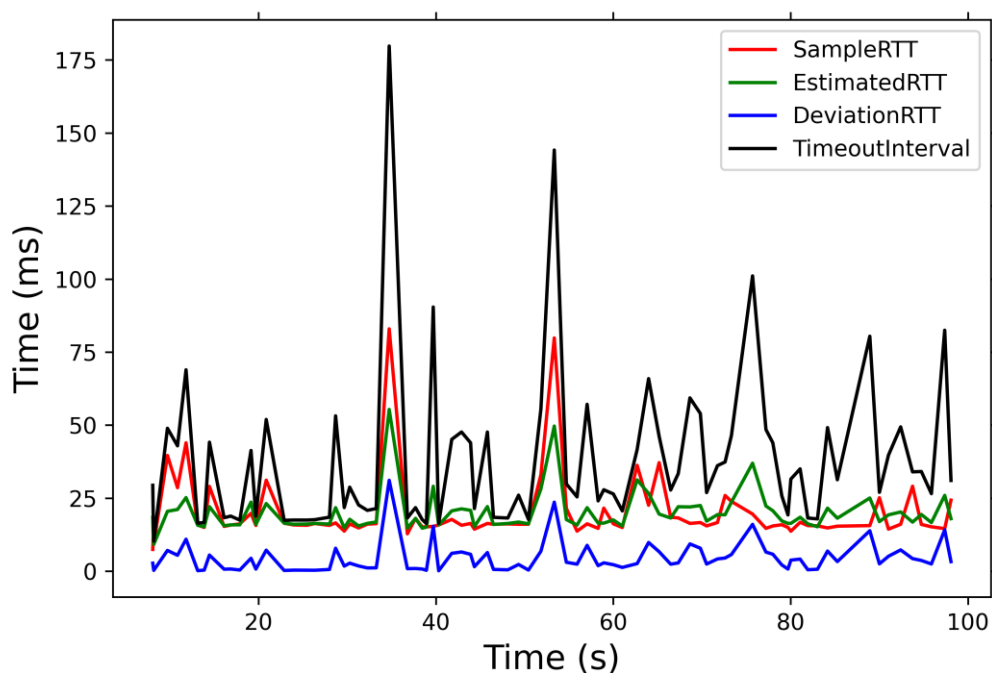


图 5-7 往返时延变化图

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。

由往返时延变化图分析：

- (1) 在数据传输的初始阶段，总 RTO（即 TimeoutInterval）设定得相对较高，主要是由于此时的 sampleRTT 值偏大。此外，在数据传输过程中，RTO 会经历一定的波动，很可能是网络链路上的延迟增加或数据包丢失引起的，进而导致时间上的不稳定性。
- (2) EstimatedRTT 与 SampleRTT 的变化趋势呈现出高度一致性，而 TimeoutInterval 则根据这两者的动态变化进行相应的调整。这种调整机制确保了网络传输的灵活性和适应性，以合理应对网络条件的变化。
- (3) DevRTT 作为 SampleRTT 与 EstimatedRTT 之间差异的指数加权移动平均值，反映出 SampleRTT 的较大波动。而 TimeoutInterval 的计算公式（ $\text{EstimatedRTT} + 4 * \text{DevRTT}$ ）则进一步放大了这种波动，并成为影响 TimeoutInterval 变化的主导因素。

5.3 流量控制的功能测试与结果分析

1. 结果图

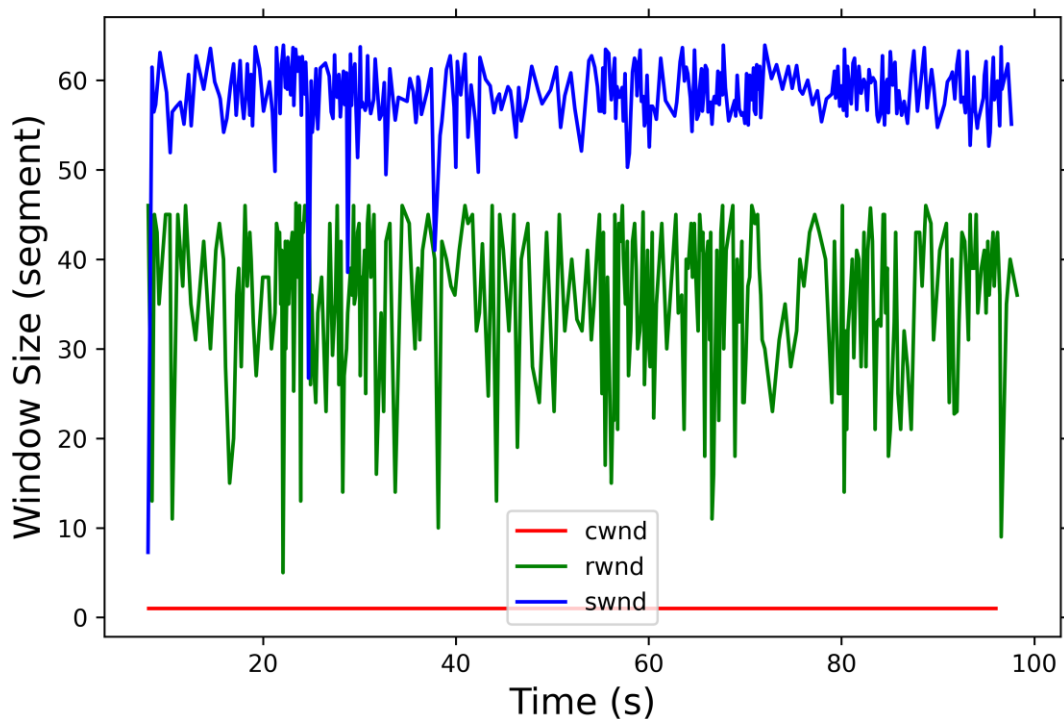


图 5-8 窗口变化图（实现流量控制）

2. 结果分析

结合所学以及上图分析可知：

- （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

1. 结果图

测试的网络环境为默认的带宽 100Mbps、延迟 300ms、延迟波动 50、丢包率 10%，得出如下发包序号的变化曲线：

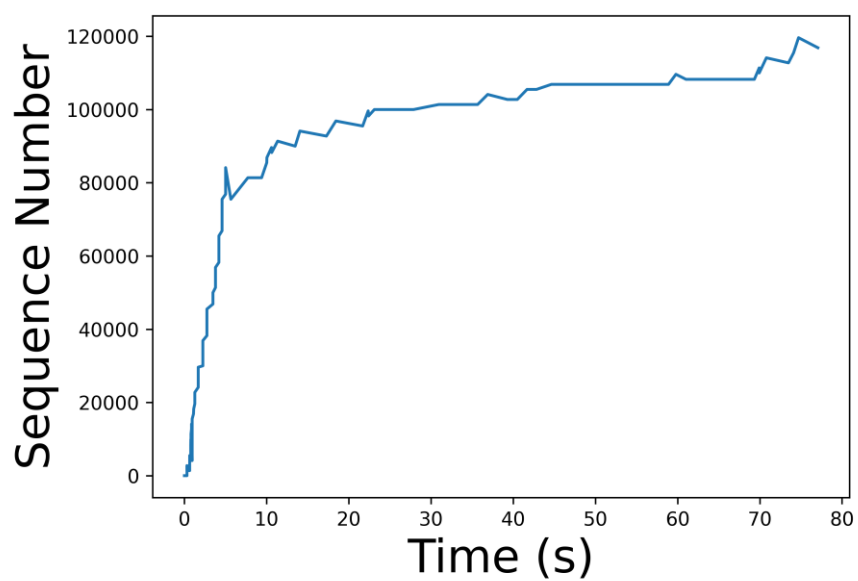


图 5-9 发包序号的变化曲线

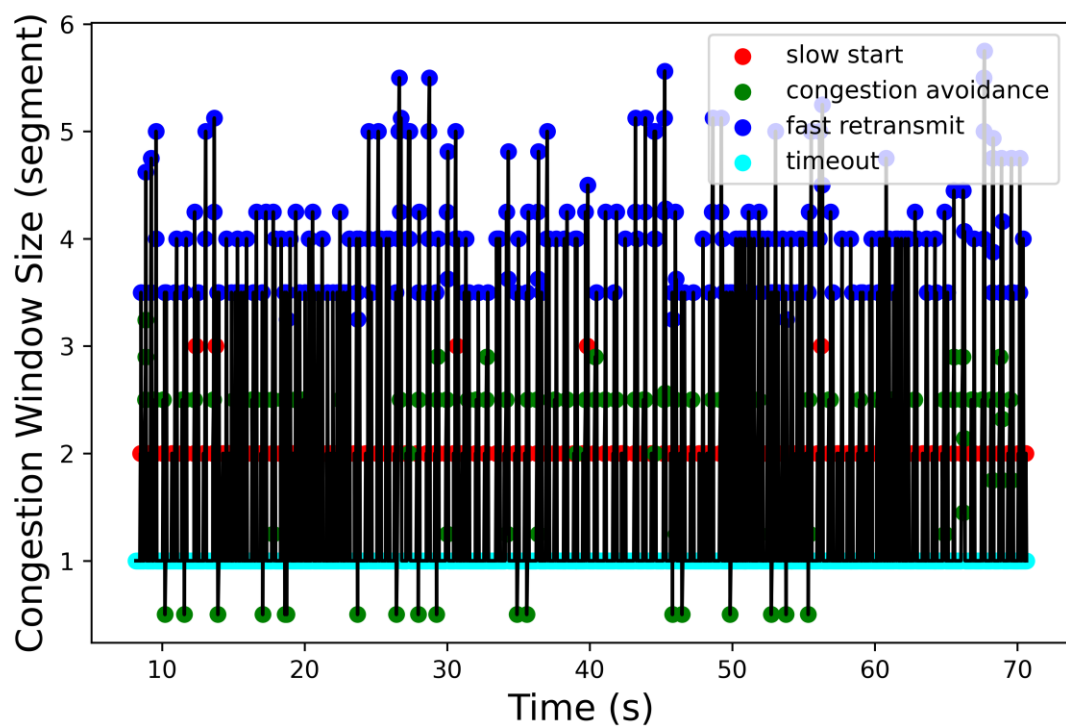


图 5-10 拥塞窗口大小与时间关系

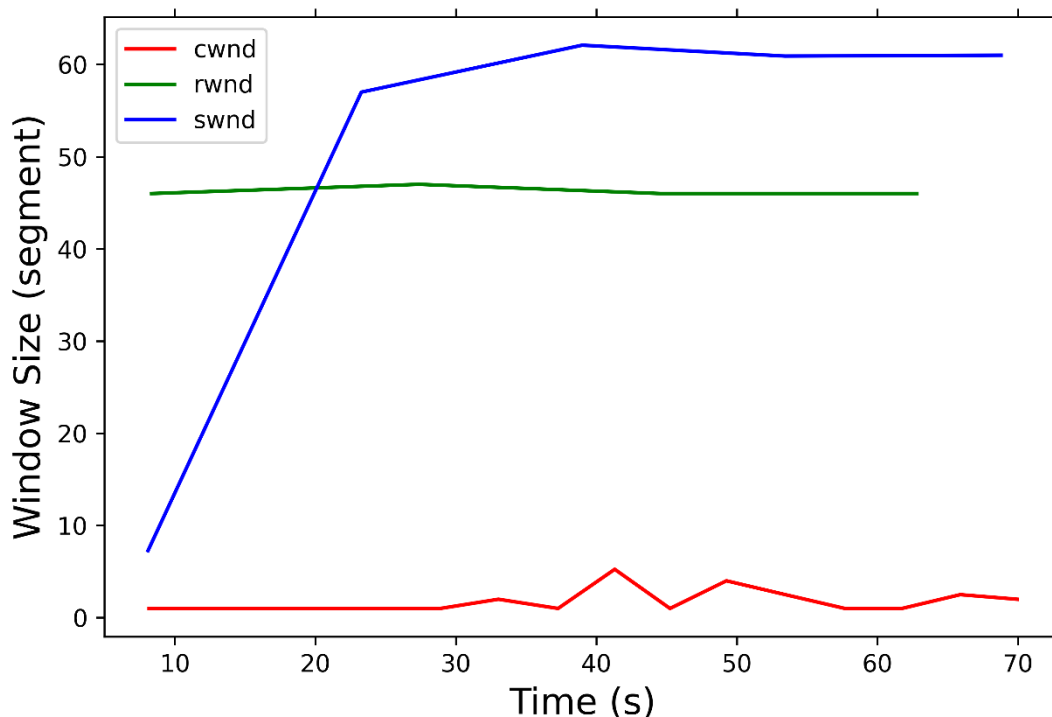


图 5-11 窗口大小变化图（实现拥塞控制）

2. 实验结果分析

通过观察图 5-9 “Sequence Number-Time” 图，可以明显看到到起始阶段为慢启动模式，数据包发送速率迅速攀升，展现出指数级增长的趋势，但随后迅速过渡至拥塞避免阶段，并伴随有数据包重传的现象（通过序列号的非连续增加得以体现）。

此外，图 5-10 “拥塞窗口大小与时间关系图” 描绘了拥塞状态转变的历程：从短暂的慢启动阶段起步，直至进入拥塞避免阶段；在网络条件不佳时，可能因超时直接回退到慢启动；而在网络状况良好时，则可能先因接收到冗余 ACK 而进入快速恢复阶段，其后亦有可能遭遇超时事件。进一步分析图 5-11 “窗口大小” 变化，可见在拥塞控制及流量管理机制下，拥塞控制窗口 cwnd 的较小值对发送窗口 swnd 的大小构成了限制，从而影响了数据传输的效率。

横向对比发现将参数修改为带宽 100Mbps、延迟 300ms、延迟波动 50、丢包率 50%，快速重传对应的窗口值整体偏低，慢启动对应的窗口值在整个区间几乎保持不变。

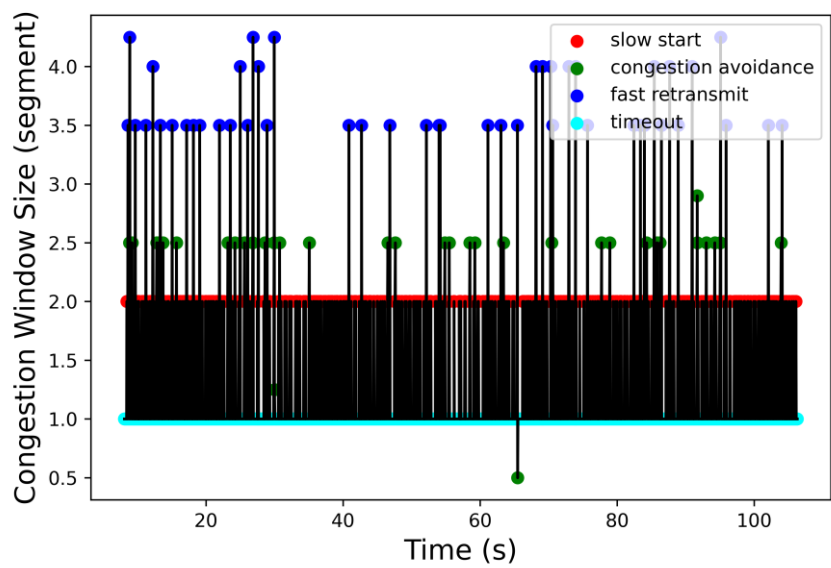


图 5-12 拥塞窗口大小与时间关系（参数有变）

5.5 TCP 协议性能测试与结果分析

1. 发送端窗口大小对吞吐率的影响

下面展示其中的一张图片：

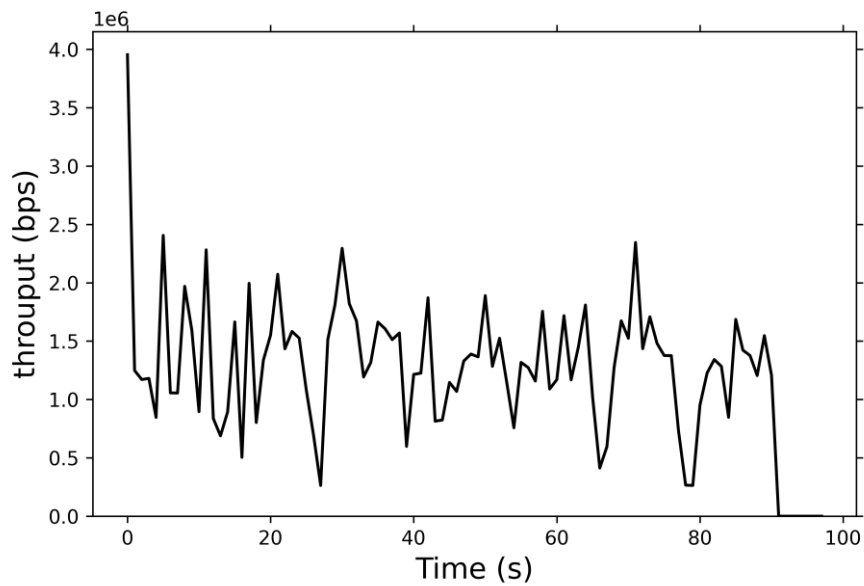


图 5-13 SeqNum 与时间关系变化曲线

在相同条件下发送相同的 50MB 数据量时，平均吞吐率随窗口大小的变化图如下图所示：

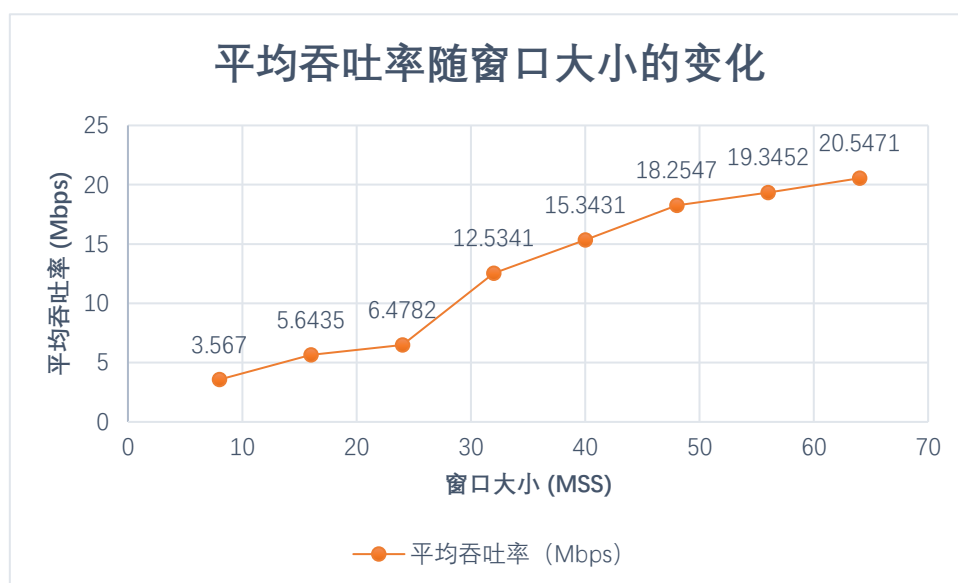


图 5-14 平均吞吐率随窗口大小的变化曲线

结论：不难看出看到吞吐率随窗口大小增大也逐渐增大，除窗口大小由 16MSS 增至 32MSS 时斜率较大，其余情况增速平稳。

2. 丢包率对吞吐率的影响

在相同条件下发送相同的 50MB 数据量时，平均吞吐率随丢包率的变化图如下图所示：

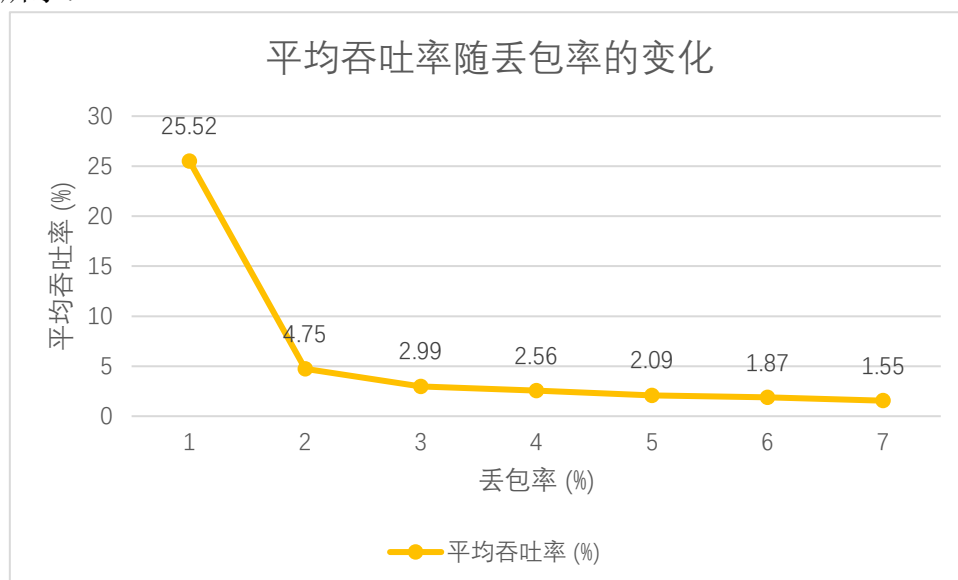


图 5-15 平均吞吐率随丢包率的变化曲线

结论：可以看到吞吐率随丢包率升高先断崖式下降，然后降速明显放缓，以近乎水平的趋势平稳下降。

六、总结

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：

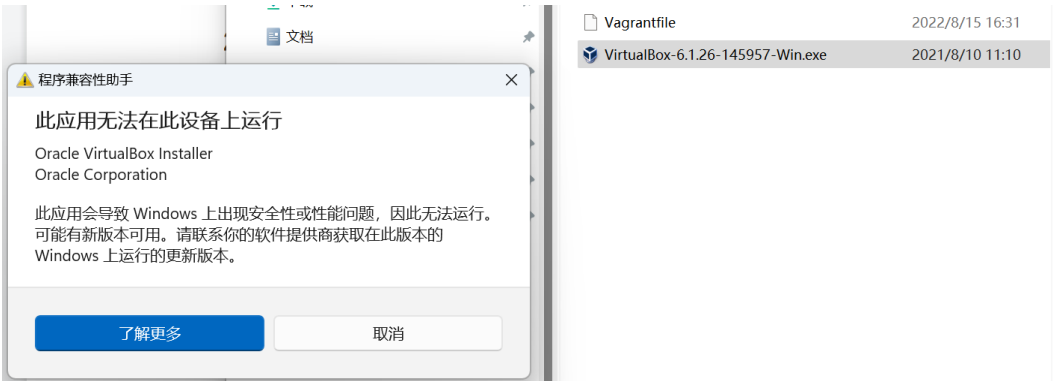


图 6-1 第一周问题 1

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：

```
PS D:\networkproject> vagrant up
Bringing machine 'client' up with 'virtualbox' provider...
Bringing machine 'server' up with 'virtualbox' provider...
==> client: Importing base box 'ubuntu/netproj'...
==> client: Matching MAC address for NAT networking...
==> client: Setting the name of the VM: networkproject_client_1724681322435_50580
Vagrant is currently configured to create VirtualBox synced folders with
the "SharedFoldersEnableSymlinksCreate" option enabled. If the Vagrant
guest is not trusted, you may want to disable this option. For more
information on this option, please refer to the VirtualBox manual:
https://www.virtualbox.org/manual/ch04.html#sharedfolders
This option can be disabled globally with an environment variable:
VAGRANT_DISABLE_VBOXSYMLINKCREATE=1
or on a per folder basis within the Vagrantfile:
config.vm.synced_folder '/host/path', '/guest/path', SharedFoldersEnableSymlinksCreate: false
==> client: Clearing any previously set network interfaces...
==> client: Preparing network interfaces based on configuration...
client: Adapter 1: nat
client: Adapter 2: intnet
==> client: Forwarding ports...
client: 22 (guest) => 2222 (host) (adapter 1)
==> client: Booting VM...
There was an error while executing 'VBoxManage', a CLI used by Vagrant
for controlling VirtualBox. The command and stderr is shown below.

Command: ["startvm", "5eccfd8b-7dee-47b4-a151-48e92988be07", "--type", "headless"]
Stderr: VBoxManage.exe: error: Failed to get device handle and/or partition ID for 0000000001f68b80 (hPartitionDevice=000000000000a9d, Last=0xc0000002/1) (VERR_NEW_VM_CREATE_FAIL
ED)
VBoxManage.exe: error: Details: code E_FAIL (0x80004085), component ConsoleWrap, interface IConsole
PS D:\networkproject>
```

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。

```
PS C:\Users\13904> bcdedit /set hypervisorlaunchtype off
操作成功完成。
PS C:\Users\13904> |
```

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：

```
==> client: Booting VM...
There was an error while executing `VBoxManage`, a CLI used by Vagrant
for controlling VirtualBox. The command and stderr is shown below.

Command: ["startvm", "5eccfd8b-7dee-47b4-a151-48e92988be07", "--type", "headless"]

Stderr: VBoxManage.exe: error: The VM session was closed before any attempt to power it on
VBoxManage.exe: error: Details: code E_FAIL (0x80004005), component SessionMachine, interface ISession
PS D:\networkproject\tju_tcp> |
```

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

```
src/tju_tcp.c: In function 'tju_handle_packet':
src/tju_tcp.c:287:9: error: a label can only be part of a statement and a declaration is
not a statement
 287 |         char* packet_FIN_ACK4=create_packet_buf(get_dst(pkt),get_src(pkt),get_se
q(pkt),get_seq(pkt)+1,\
    |         ^~~~~
make: *** [Makefile:19: build/tju_tcp.o] Error 1
Traceback (most recent call last):
  File "test.py", line 420, in <module>
  File "test.py", line 399, in main
  File "test.py", line 74, in compile_source_files
  File "<decorator-gen-3>", line 2, in run
  File "fabric/connection.py", line 30, in opens
  File "fabric/connection.py", line 723, in run
  File "invoke/context.py", line 102, in _run
  File "invoke/runners.py", line 380, in run
  File "invoke/runners.py", line 442, in _run_body
  File "invoke/runners.py", line 509, in _finish
invoke.exceptions.UnexpectedExit: Encountered a bad command exit code!

Command: 'cd /vagrant/tju_tcp && make'

Exit code: 2

Stdout: already printed

Stderr: already printed
```

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 `switch` 语句块），所以如果在某个 `case` 下面声明变量的话，对象的作用域是在两个花括号之间，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结构的分析与理解。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。

6.1.3 第三周

1. 关于任务间的影响

第三周遇到的问题主要是实现拥塞控制后，可靠数据传输的测试分数变低且不稳定。与同学讨论后，得知应观察 `log` 日志，寻找推迟原因。

2. 关于绘图程序的疑惑

按照指导书运行代码后，出现如下报错，起初未仔细阅读报错提示，认为是 `trace` 文件过大导致的抓包困难（实际上 `trace` 文件确实有 20 万行的内容），尝试删除部分文件内容等方式仍未解决。

```

vagrant@client:/vagrant/tju_tcp/test$ python3 /vagrant/tju_tcp/test/gen_graph_win.py /va
grant/tju_tcp/test/client.event.trace
正在使用 /vagrant/tju_tcp/test/client.event.trace Trace文件绘图
Traceback (most recent call last):
  File "/vagrant/tju_tcp/test/gen_graph_win.py", line 144, in <module>
    SEND_dic, RECV_dic, CWND_dic, RWND_dic, SWND_dic, RTTS_dic, DELV_dic = read_trace(FI
LE_TO_READ)
  File "/vagrant/tju_tcp/test/gen_graph_win.py", line 83, in read_trace
    info_list = [item.split(':')[1] for item in info_list]
  File "/vagrant/tju_tcp/test/gen_graph_win.py", line 83, in <listcomp>
    info_list = [item.split(':')[1] for item in info_list]
IndexError: list index out of range

```

图 6-6 第三周的问题 2

静下心来观察发现原因是自动终止数据传输后，最后一条日志文件可能会不完整，导致绘图的索引“对不上”。于是删除最后一条不完整的记录后重新执行指令，成功绘图。

3. 报错

执行画图程序，突然出现以下信息，无法绘图。

```

socket.timeout

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/vagrant/tju_tcp/test/test_congestion.py", line 244, in <module>
    main()
  File "/vagrant/tju_tcp/test/test_congestion.py", line 85, in main
    rst = conn.run("cd /vagrant/tju_tcp && make", timeout=10)
  File "<decorator-gen-3>", line 2, in run
  File "/usr/local/lib/python3.8/dist-packages/fabric/connection.py", line 29, in opens
    self.open()
  File "/usr/local/lib/python3.8/dist-packages/fabric/connection.py", line 636, in open
    self.client.connect(**kwargs)
  File "/usr/local/lib/python3.8/dist-packages/paramiko/client.py", line 406, in connect
    t.start_client(timeout=timeout)
  File "/usr/local/lib/python3.8/dist-packages/paramiko/transport.py", line 660, in star
    t_client
    raise e
  File "/usr/local/lib/python3.8/dist-packages/paramiko/transport.py", line 2039, in run
    self._check_banner()
  File "/usr/local/lib/python3.8/dist-packages/paramiko/transport.py", line 2215, in _ch
    eck_banner
    raise SSHException(
paramiko.ssh_exception.SSHException: Error reading SSH protocol banner
vagrant@client:/vagrant/tju_tcp/test$

```

图 6-7 第三周的问题 3

解决方法：vagrant halt 后重新 vagrant up，并进行后续连接，问题解决，成功绘图。

4. 环境问题

无法进行 vagrant up 操作，尝试重新开启终端窗口等方式均无法解决，最后重启、更新系统，问题得以解决。

```
==> client: Booting VM...
There was an error while executing `VBoxManage`, a CLI used by Vagrant
for controlling VirtualBox. The command and stderr is shown below.

Command: ["startvm", "5eccfd8b-7dee-47b4-a151-48e92988be07", "--type", "headless"]

Stderr: VBoxManage.exe: error: The VM session was closed before any attempt to power it on
VBoxManage.exe: error: Details: code E_FAIL (0x80004005), component SessionMachine, interface ISession
PS D:\software\kubernetes\day_4\src> vagrant up
```

图 6-8 第三周的问题 4

6.2 收获、体会及建议

通过本次课程实践，我复习了解了许多平时不常用的 C 语言函数：比如 `sprintf` 函数、`fprintf` 函数、`fflush` 函数等等。

初见代码时毫无头绪，便将逐个文件、每个函数仔细阅读并加以注释，强化理解，但是代码很长、文件很多，真的花费了很久才理清代码逻辑与函数功能。接着就是翻看老师的 PPT 和计算机网络教材、复习可靠数据传输等 TCP 相关知识点。接下来遇到的问题主要是环境的不稳定和不知道怎么将原理转化成代码加以实现，经历了试错、疑惑、讨论，每个任务都成功按时完成。

虽然没有与同学组队，但在本次课程设计中通过不断阅读相关文档、自己多次尝试与学习、请教助教老师、与同学讨论，我逐步实现了实验要求，在其中不仅增进了对 TCP 协议等知识的理解，还增加了学习能力和抗压能力。很幸运能有这样一个机会，将课上所学的理论知识与实践相结合，在深入理解计算机网络及协议的同时，提高了代码能力和文档写作水平。

当然，不管是在这次的课程实验还是在过往的学习中，数不清自己经历过多少次在花了很长时间配置环境或编写代码后，进行测试后电脑屏幕上显示的却是冰冷的报错信息的情景。我也总是因此感觉心灰意冷，感慨计算机为什么总是给出模式化的报错信息，而不能帮我精准定位到出现错误的代码位置并且给出修改方案；也时常感叹自己掌握的知识还是太少了，还是需要很多的学习和练习；但是我也发现现在针对一些简单的错误比如调用函数时函数名称多或少了一个字母，计算机已经可以实现我“精准定位到出现错误的代码位置并且给出修改方案”的畅想了。但不管怎样，我都将继续学习掌握更多的专业知识，多多动手实践，提高能力水平，希望未来有机会可以为广阔的知识海洋贡献属于自己的崭新的一滴水。