

天津大学

《计算机网络》 课程报告



题目：TCP 在应用层的设计与实现

学 号：3019244160 3020202184

姓 名：程子姝 刘锦帆

学 院：智能与计算学部

专 业：计算机科学与技术

年 级：2020 级

任课教师：石高涛

2022 年 10 月 19 日

目 录

第一章	报告摘要	1
第二章	任务要求	2
第三章	协议设计	3
3.1	总体设计	3
3.2	连接建立的设计	3
3.3	可靠数据传输的设计	4
3.3.1	状态机设计	5
3.3.2	超时重传机制	7
3.3.3	缓冲区与滑动窗口设计	7
3.4	流量控制的设计	8
3.5	连接关闭的设计	8
3.6	拥塞控制的设计	10
第四章	协议实现部分	11
4.1	连接建立的实现	11
4.2	可靠数据传输的实现	12
4.2.1	数据结构	12
4.2.2	实现思想	14
4.2.3	具体实现	14
4.3	流量控制	15
4.4	连接关闭的实现	15
4.5	拥塞控制的实现	16

第五章	实验结果及分析	17
5.1	连接建立的功能测试与结果分析	17
5.2	可靠传输的功能测与结果分析	17
5.2.1	RTO 动态调整	17
5.2.2	可靠数据传输	17
5.3	流量控制的功能测试与结果分析	19
5.4	连接关闭的功能测试与结果分析	20
5.5	拥塞控制的功能测试与结果分析	20
5.6	TCP 协议性能测试与结果分析	21
第六章	总结	23

一 报告摘要

随着时代的发展，网络技术日新月异，为了能够对网络技术的底层设计哲学有更深层的理解，我们参与并完成了本次关于 TCP 协议在应用层实现的实践任务。我们首先根据 RFC 793 文档设计了基于应用层的实现逻辑。然后使用 C 语言，在统一的 Ubuntu 环境（见图一）下进行了协议实体的代码实现。实现了包括“连接管理”，“可靠数据传输”，“流量控制”，“连接关闭”和“拥塞控制”等功能。并对吞吐率进行了验证，验证了协议实现的正确性。

```
vagrant@client:/vagrant/tju_tcp$ neofetch
  .-/+oossssoo+/-.
   `:+ssssssssssssssssssss+:'-
  -+ssssssssssssssssssssyssss+-.
  .osssssssssssssssssssdMMMNyssso.
  /sssssssssssshdmmNNmmyNMMMHhssssss/
  +ssssssssssshmydMMMMMMMddddyssssssss+
  /ssssssssshNMMMyhyyyyhmNMMMNhssssssss/
  .ssssssssdMMMNhssssssssssshNMMMdssssssss.
  +sssshhhyNMMNysssssssssssyNMMMyssssssss+
  ossyNMMMNyMMhssssssssssssshmmmhssssssso
  ossyNMMMNyMMhssssssssssssshmmmhssssssso
  +sssshhhyNMMNyssssssssssssyNMMMyssssssss+
  .ssssssssdMMMNhssssssssssshNMMMdssssssss.
  /ssssssssshNMMMyhyyyyhdNMMMNhssssssss/
  +ssssssssdmydMMMMMMMddddyssssssss+
  /sssssssssssshdmNNNNmynMMMHhssssssss/
  .osssssssssssssssssssdMMMNyssso.
  -+ssssssssssssssssssyyssss+-.
  `:+ssssssssssssssssss+:'-
  .-/+oossssoo+/-.

vagrant@client:/vagrant/tju_tcp$
```

图 1-1 测试环境信息

协议设计分为六个部分，将在第三章进行重点介绍，设计的重点在于将 RFC 793 文档的文字进行逻辑翻译。程序总共三个线程，使用一个接收线程处理接收到的包裹、发送 ACK 以及将内容递交给上层实体；使用一个发送线程从上层实体接收待发送的包，并进行可靠数据传输；最后一个应用层的用户端主线程。

我们顺利通过了三次线上的测试以及拥塞控制和性能测试等额外测试（拥塞控制并没有来得及提交报告，在本报告中统一进行阐述）。

二 任务要求

使用 UDP 协议，根据 RFC 793 文档的描述，实现一个基于应用层的 TJU_TCP 协议。

TCP 提供客户与服务器之间的连接，TCP 客户端先与某个给定服务器建立连接，然后通过该 Socket 连接与服务器进行数据交换，最终终止连接。同时在传输过程中，需要进行流量的控制，避免发送端过快导致对方无法正确接收。最后还需要处理拥塞控制，避免因为发送过快而导致丢包不断。因此包括以下功能：

1. 三次握手：建立连接
2. 可靠的数据传输功能：滑动窗口，超时重发，选择重发等策略，ACK 协议
3. 四次挥手：断开连接
4. 流量控制：关于 advertise_window 的使用
5. 拥塞控制：流量控制，慢启动，拥塞避免，快速重传等

在完成代码实验后，需要依次通过总共 3 个 autolab 的线上测试（如图2-1），以及根据 Trace 文件进行吞吐量、拥塞控制窗口的检测进行报告的撰写和现场展示。

establish_connection (100.0)	Late Days Used	Total Score
100.0	Submitted 0 days late	100.0
Submitted at: 2022-09-17 19:18:00 +0800		
a) 第一周结果		
reliable_data_transfer (100.0)	Late Days Used	Total Score
100.0	Submitted 0 days late	100.0
Submitted at: 2022-10-02 16:22:04 +0800		
b) 第二周结果		
3 479404627@qq.com_3_handin.zip	2022-10-11 21:36:24 +0800	100.0
		Submitted 2 days late 100.0
c) 第三周结果		

图 2-1 三周的结果

三 协议设计

3.1 总体设计

为实现 TCP 的全部协议内容，我们将协议拆分成 5 个部分：连接建立、可靠数据传输、流量控制、连接关闭以及拥塞控制。在本章中，将从流程图和状态机的角度进行宏观描述，而代码实现将在下一章中进行详细介绍。

3.2 连接建立的设计

连接建立的流程设计如图3.2所示

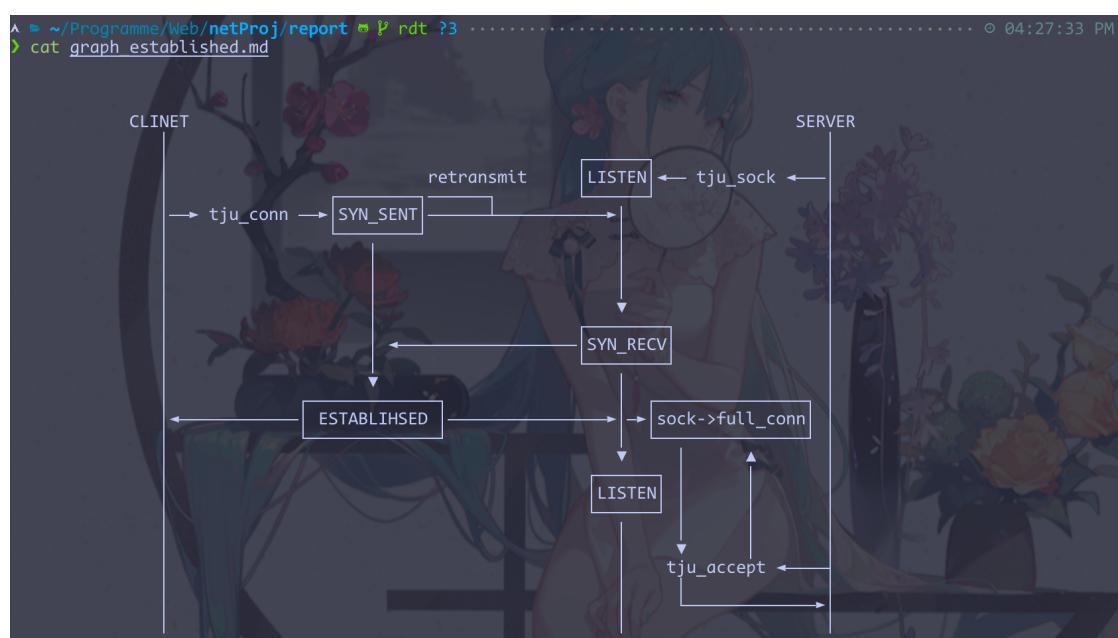


图 3-1 连接建立流程图

由图可知，连接建立应当以 `tju_conn` 和 `tju_accept` 两个 API 供用户使用，（当然，首先 `server` 端需要使用 `tju_sock` 接口获取一个初始的 socket）。

状态机如图3.2所示：

具体流程如下：

1. Server 端使用 tju_sock 接口获取一个处于 LISTEN 状态的 socket
 2. Client 端调用 tju_conn 接口向 Server 端发送报文 1（请求连接），必要时进行超时重发（这个在后期会与 rdt 进行融合），并将自己的状态设置为"SYN_SENT"

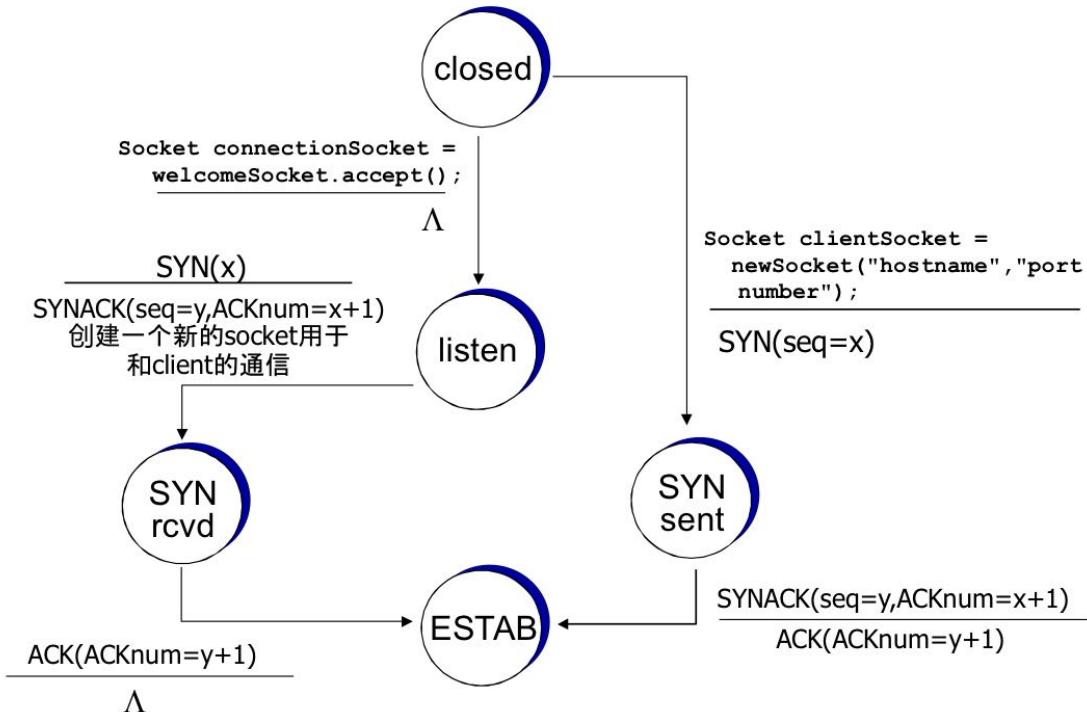


图 3-2 连接建立 FSM

3. Server 端接受到报文 1，确认要连接，发送报文 2，并将自己的状态设置为"SYN_RECV"
4. Client 端接收到报文 2 后，将自己的状态设置为"ESTABLISHED" 并向上层接口返回一个建立连接的 socket。同时，向 Server 发送报文 3 作为答复。
5. Server 端接收到报文 3 后新建一个 socket 并将其放入初始 socket 的全连接队列中。
6. 当 Server 端调用 tju_accept 进行连接接收时，只需从全连接队列中拿出一个即可。

3.3 可靠数据传输的设计

可靠数据传输的流程图如图3.3所示：

在该图中，我们通过 ACK 和超时重传的原理进行环境丢包、接收端丢包的处理。为此我们需要实现一些数据结构：发送端负责超时重传的 timer_list 以及接收端负责接收乱序数据的 disoerder_tree 缓冲区。

在发送端，我们直接采用了 sending_queue 的结构将主线程和发送线程连接，即在 tju_send 函数中，将数据打包成 tju_packet_t 结构，并放入 sending_queue，然后发送线程从 sending_queue 中获取新的发送包裹。将数据以包为单位进行计数和重传，使得整个程序的可读性、可维护性提高。同时，我们设计并实现了自定

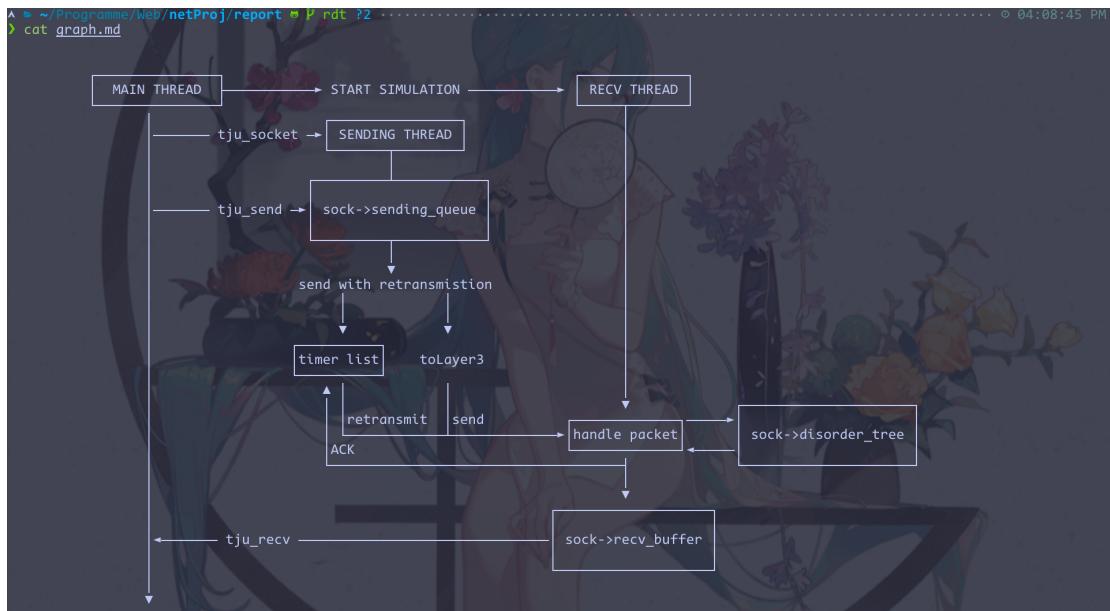


图 3-3 可靠数据传输流程图

义的队列数据结构，在“连接建立”中的全连接队列和“可靠数据传输”的发送队列使用（具体实现见下一章）。

3.3.1 状态机设计

在此，我们给出两个状态机的设计图，结合我们的流程图进行更加详细的设计阐述。

Client 端状态机设计

Client 端状态机如图3.3.1所示。Client 端需要响应三种事件：

1. 来自上层的调用，将数据打包并发送给下一层实体 (toLayer3)，然后新建 TIMER 并注册之
2. 来自 TIMEOUT 通过 TIMER 的信息进行超时重传
3. 收到 ACK，确认 ACK 的状态，根据 ACK 信息选择性地关闭 TIMER

Server 端状态机设计

在 Server 端，我们目前暂不考虑数据出错的情况，即达到的包，都是正确的。在此情况下，我们需要处理一下情况：

1. 到达 SEQ 正确：判断缓冲区大小，选择性的将其放入缓冲区，发送 ACK。
同时在 AVL 中递归地查找连续的正确的包裹
2. 到达 SEQ 大了：将其放入 AVL Tree 并发送 ACK
3. 到达 SEQ 小了：收到过，丢弃

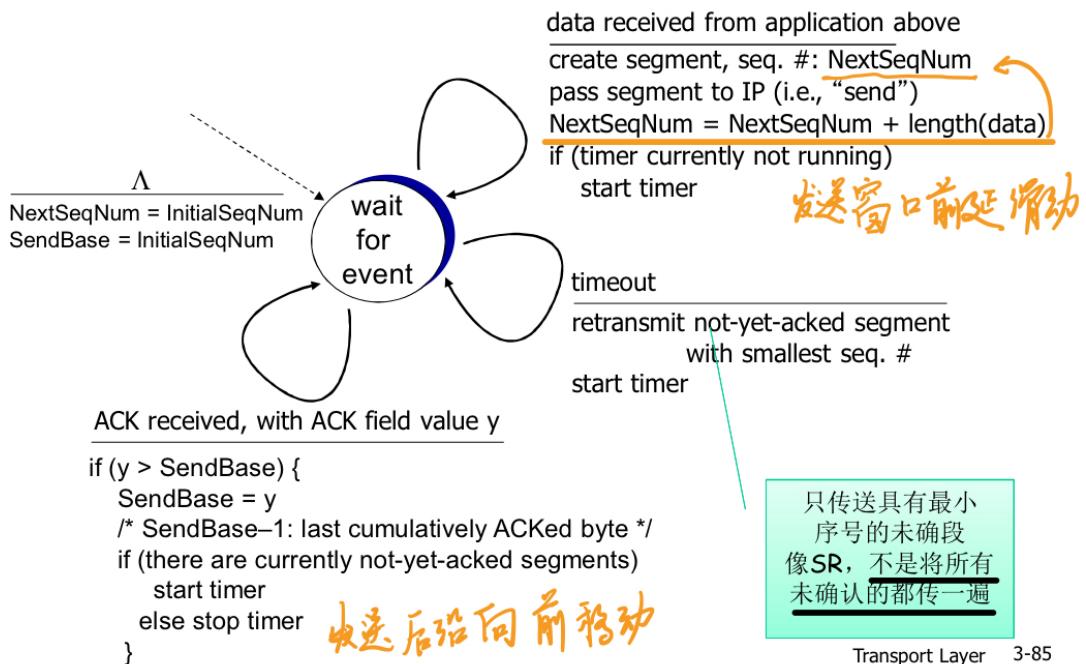


图 3-4 Client FSM

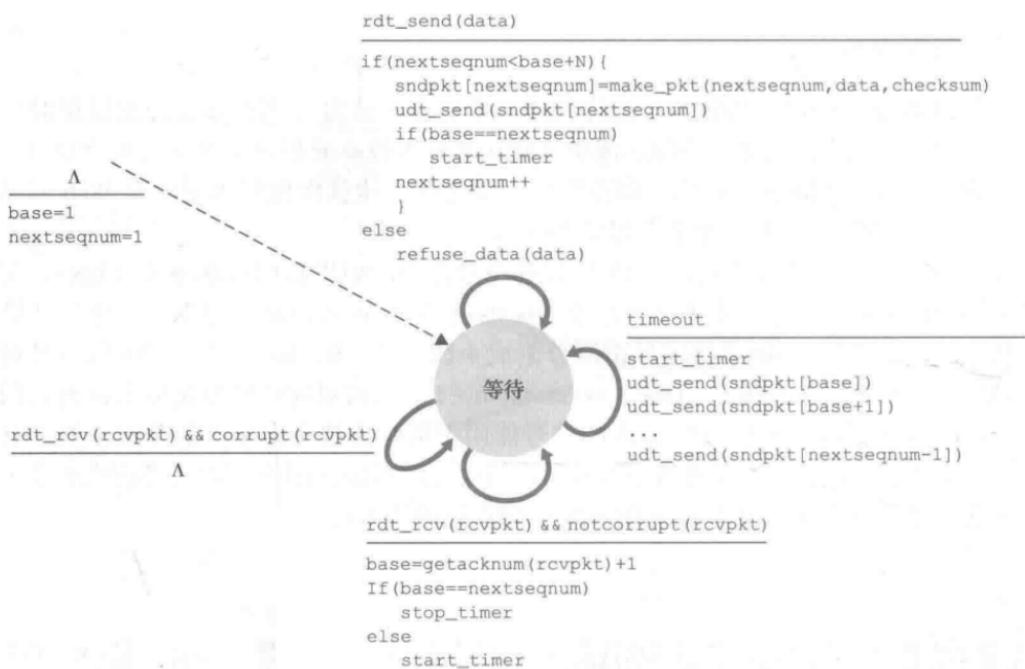


图 3-5 Server FSM

3.3.2 超时重传机制

我们采用的是 ACK 累计确认型的超时重传机制，因此需要在 Server 端建立一个 AVL 类型的缓冲区，乱序接收包裹，顺序拿出包裹。

连接建立时 丢失有三种：第一次握手丢失，服务端返回第二次握手丢失，以及第三次确认丢失。第一、二个由 Client 端设置定时器保护，第三个由 Server 端设置定时器保护，即当收到第三次确认时，Server 端再将信息定时器进行关闭。

发送数据时 丢失有两种：发送数据丢失（Seq），发送 ACK 丢失。二者对于 Client 端的表现均是长时间（RTO）没收到 ACK 信息，于是我们将定时器设置在 Client 端，由 Client 端进行超时重传。而 Server 只对于前者，即发送数据丢失的情况作出反馈，即设置乱序到达，顺序提交的机制，确保因超时而呈现乱序到达的数据得以顺序提交给上层用户。

RTO 计算法则

RTO 是超时重传机制的基础。由于网络变化，RTO 应当是动态调整的。如果 TCP 过早重传，会导致注入多数不必要的报文，阻塞网络。如果过晚重传，则会影响网络使用效率。我们根据 RFC793 的标准，让 Socket 维护了一个 SampleRTT 均值（EstimatedRTT）通过如下公式进行更新：

$$EstimatedRTT := (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT \quad (3-1)$$

除了 RTT，RFC 6298 还定义了 RTT 的偏差 DevRTT 的计算，用于 usuan SampleRTT 和 EstimatedRTT 的偏离程度

$$DevRTT := (1 - \beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT| \quad (3-2)$$

于是，最终的 TRO 计算公式如下：

$$RTO := EstimatedRTT + 4 \times DevRTT \quad (3-3)$$

3.3.3 缓冲区与滑动窗口设计

发送端缓冲区与滑动窗口 发送段共有 sending_queue 和 timer_list 两个数据结构组成发送缓冲区。采用 sending_queue，队列的数据结构，上层将包裹放入 sending_queue 表示要发送，发送线程读取 sending_queue 来进行发送。然后利用 timer_list 的个数确定滑动窗口的大小。即，使用 timer_list 来代表进入滑动窗口的包裹，通过发送将其放入 timer_list，通过 ACK 将其踢出。于是，send-

ing_queue 实际表示在缓冲区且不在滑动窗口的部分。滑动窗口的大小以 segment 计数。

接收端缓冲区 接收端有两个缓冲区，一个是正序到达时直接放入的 recv_buf，共 tju_recv 调用获取。另一个是手动实现的 AVL 树，能够快速插入乱序数据并正序取出，用于存入乱序到达的包裹。

3.4 流量控制的设计

流量控制也是 TCP 标准功能之一，是构成可靠传输的关键步骤，能够与其他 TCP 实体一起，维护整个网络的传输可靠性。如果流量过大，则会导致接收方不断丢弃传输的结果，发送方不断进行重发。故而，Server 端需要将 rwnd 的信息通过 ACK 信息携带给 Client 端，让 Client 能够正确地调整发送速率。

具体来说，我们在 Server 端的 handle_packet 函数中加入对当前接收窗口更新的代码：接收到按顺序达到的包裹、将数据放入 recv_buf 时更新 rwnd 值。同时在 tju_recv 函数中，上层拿走数据时也进行更新。同时，我们在返回 ACK 时，将 rwnd 信息通过 advertise 字段返回给 Client 端。当然，我们需要首先约定一个 Window_Dlen，这样在描述 rwnd 时，只需描述其 segment 的大小。

在 Client 端，我们通过在发送线程中"send with retransmission" 时先对当前 swnd 和对方 rwnd 的大小进行比较，如果大于，则等待 rwnd 变大或 swnd 因为 ACK 而减少时再发送包裹。此处我们因为引入了 timer_list，于是只需要使用 timer_list 的个数作为 swnd 即可。同时，我们需要在接收到 ACK 时对当前 rwnd (对方的) 数值进行更新。

3.5 连接关闭的设计

我们的协议应当给出 API tju_close 让上层进行调用。我们需要明确的是，当且仅当 socket 两端都调用了 tju_close，我们才能认为该 socket 已经被关闭了，可以回收了。否则 socket 的接收功能依旧要工作。且处于当前重传 buffer 中的包裹必须全部完成。

连接关闭的 FSM 图如图3.5所示：

在此我们对关闭时的两种情况进行描述：

一方发起关闭

其实由于 TCP 协议的设计时的对等性考虑，并没有 Client 或 Server 端的区分，在此，我们为了方便进行讨论，使用 Client 端表示优先调用 tju_close 的一方，Server 作为后调用 tju_close 的一方。

Client 端首先调用 tju_close 后，将状态转换为"FIN_+WAIT_1"，并将一

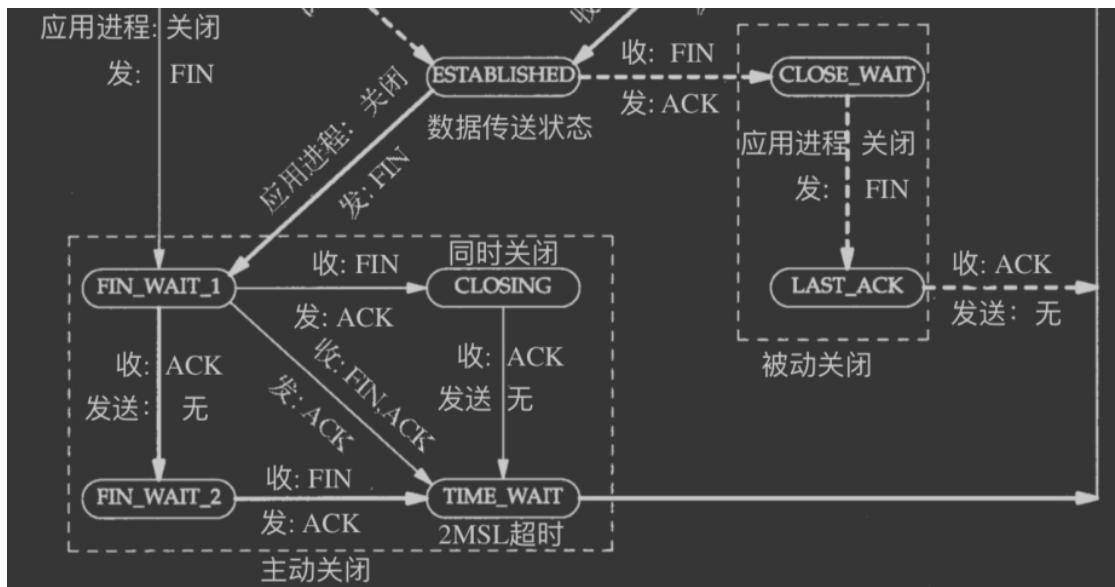


图 3-6 连接关闭 FSM

一个 FIN 包放入自己的 sending_queue 中，等待发送线程进行超时重传型的发送。同时，拒绝上层继续向 sending_queue 中放入包裹，但需要将此时 sending_queue 以及 timer_list 中的包裹全部安全送达。等待对方的 ACK 使得自己能够进入 "FIN_WAIT_2" 状态。然后继续等待，直到对方发送了 FIN 包、我方完成所有超时重传任务时，我方状态变为 CLOSED，并销毁 socket。

Server 端接收到 FIN 包时，将自己的状态变为 CLOSE_WAIT 并返回一个 ACK，但仍然保持接收和发送功能。直到我方调用 tju_close 进入关闭程序：将己方状态转换为 "LAST_ACK" 并拒绝上层继续向 sending_queue 中放入任何包裹，但需要将已经在 sending_queue 和 timer_list 中的包裹进行重传型的发送，安全送达对岸，即使对面已经宣布关闭。在接受到对方发送的关于 FIN 包裹的 ACK 时，我们才能关闭己方 socket。因为如果对方能够对 FIN 包裹进行 ACK，由于我们采用的是累计确认，则此时代表 FIN 包裹发送之前所有的包裹都已经收到。完成了对上层的承诺，可以关闭了。

两方同时关闭

此时我们认为是 Server 端在没有接收 FIN 包，且 Client 端已经发送的情况下发送了 Server 端的 FIN 包裹。

此时 Server 和 Client 端在接收到 FIN 包裹时，其状态都是 "FIN_WAIT_1"，由于尚未接收到 ACK 而直接接收到 FIN，故而将状态转变为 "TIME_WAIT" 并在两个超时后直接关闭连接。由于此时接收到了 FIN（累计确认）且自己已经

发送 FIN，故而表明我们没有继续发送包裹，且对方亦然，所有包裹都已经发送、接收完毕，该 socket 可以关闭。

3.6 拥塞控制的设计

在拥塞控制中，我们采用一个 cwnd 的结构进行控制，在发送线程每次从 sending_queue 读取包裹时，首先需要根据当前的 rwnd 更新滑动窗口的大小，在包含拥塞控制的设计中，我们使用加入对 cwnd 的考量，即选取当前 cwnd 和 rwnd 中较小的一个作为当前 swnd 的值。然后根据 swnd 来判断是否在该轮循环中进行数据的发送。

cwnd 的更新，为此，我们创建一个 cwnd 的数据结构，包括当前 cwnd 的状态 cwnd_STATE 和窗口大小 cwnd，上限 sshresh，并在以下事件结点进行更新：

1. 新的 ACK：按照状态进行更新，同时按照情况将状态从慢启动换为拥塞避免
2. 出现超时：将 cwnd 降为 $1, sshresh$ 降为 $cwnd / 2 + 1$ ，同时设置状态为拥塞避免
3. 出现三个连续 ack：将 cwnd 将为 $1/2 + 3$ ，sshresh 将为 $cwnd / 2 + 1$ ，同时将状态设置为慢启动

拥塞控制的 FSM 如图3.6所示

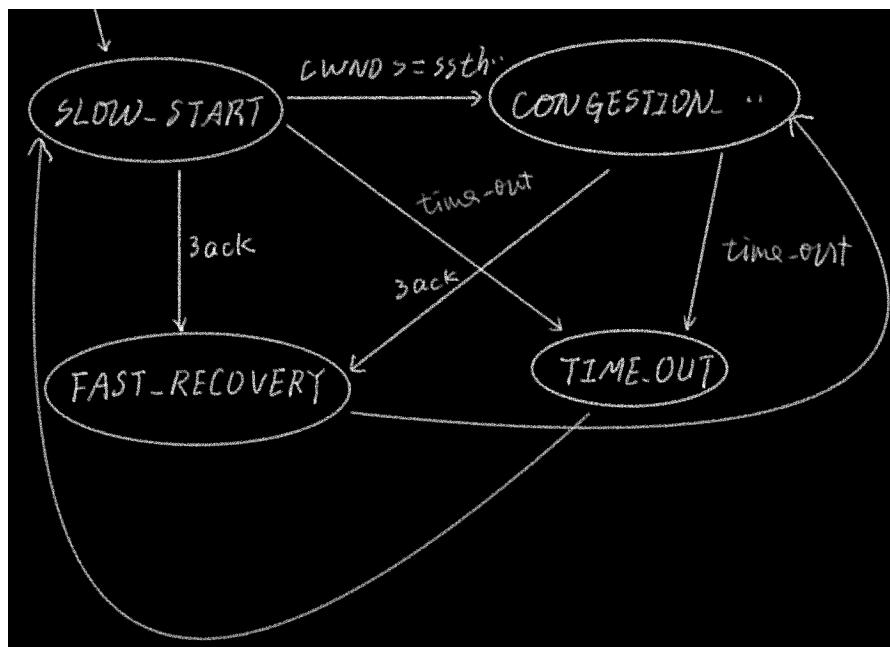


图 3-7 拥塞控制 FSM

四 协议实现部分

首先，从宏观上阐释：我们 TCP 实现部分采用了两个线程进行服务：发送线程和接收线程。

发送线程 负责从数据结构 sending_queue 中取出并按需发送报文、设置时钟，检查时钟进行超时重传。在用户调用 tju_socket 生成 socket 时创建。首先发送线程会到达 ACK 时预订消除的时钟进行删除，然后判断是否有超时时钟，执行重传。最后进行正常报文的发送。

接收线程 负责从 socket 中获取对方发送的报文、返回 ACK 并处理不按需到达的报文，确保放入 recv_buf 中的数据是正确且有序的。在接收线程收到包裹后，调用 tju_handle_packet 进行处理。

tju_handle_packet 处理报文的主要函数，通过 switch，根据当前 socket 的状态选择合适分支进行报文的处理。

4.1 连接建立的实现

要实现连接的建立，我们实现了如图3.2的流程图。实现了 tju_conn 和 tju_accept 供 client 以及 server 端进行调用的 API

tju_conn Client 端通过该 API 对 Server 端发起连接请求。而后进入等待循环，直到自己 socket 的状态变为 Established 再返回。而 socket 状态的改变则由接收线程处理。

tju_accept Server 端调用该 API，从处于 LISTEN 状态的 socket 的 full_connection 列表中获取一个全连接的 socket 即可。

接收线程-三次握手 在 tju_handle_packet 中，关于连接建立有如下三种情况：

1. LISTEN: 此状态为 Server 端。接收到 SYN 报文后，Server 端将状态变为 SYN_RECV 并发出 SYN | ACK 报文，同时将其放入半连接队列中，发送线程需要对半连接中的 sock 进行重发
2. SYN_SENT: 此状态为 Client 刚发送完 SYN 包裹，从连接的角度来说是 Client 端。此时收到包裹应为 SYN | ACK，否则拒收，由发送线程进行重

发。接收正确报文后，socket 状态调整为 ESTABLISHED，同时返回一个 ACK 报文答复。

3. SYN_RECV：此状态为 Server 端发送完 SYN | ACK 报文，等待 client 的 ACK。得到 ACK 后，将其半连接的 sock 拿出，放入全连接队列，等待 tju_accept 进行调用。

4.2 可靠数据传输的实现

4.2.1 数据结构

为了更好地实现可靠数据传输，我们额外实现了三种数据结构进行更优化的处理。分别是 timer_list，queue 和 tree。

timer_list 是对超时重发的处理，本质上是一个队列，其实现的头文件如图4.2.1所示。对于每个 timer 我们记录了它创建的时间和创建时根据 socket RTO 计算出的 timeout 时间，以及它所代表的 tju_packet_t 和重传时的回调函数。我们采用 list 链表的形式进行整个线性表的遍历。当然，我们需要对其进行加锁，避免出错。

```
t/s/tju_tcp.c | t/s/tran.c | t/i/timer_helper.h
43 #ifndef __LIST_H__
42 #define __LIST_H__
41 /* 定义一个 LIST 类型（链表实现） */
40 #include "global.h"
39 #include "debug.h"
38 #include "queue.h"
37
36 //TODO: Not yet Finish IMPLEMENTATION
35 #define SEC2NANO(x) ((uint64_t)(x * 1000000000))
34 #define NANO2SEC(x) ((float)((float)(x)/1000000000.0))
33 #define NANO2TIMESPEC(nano) ({struct timespec}{.tv_sec = (time_t)(nano / 1000000000), .tv_nsec = (long)(nano % 1000000000)})
32 #define TIMESPEC2NANO(timespec) ((uint64_t)(timespec.tv_sec * 1000000000 + timespec.tv_nsec))
31
38 typedef struct timer_event{
39     struct timespec *create_at;
40     struct timespec *timeout_at;
41     tju_packet_t *args;
42     void * (*callback)(void *, void *); // 调用函数传递的参数
43     void * (*func); // 调用的函数
44 }timer_event;
45
46 typedef struct timer_node{
47     struct timer_event *event;
48     struct timer_node *next;
49     int id;
50 }timer_node;
51
52 typedef struct timer_list{
53     timer_node *head;
54     timer_node *tail;
55     int size;
56     int total;
57     pthread_mutex_t lock;
58     myQueue *queue;
59 }timer_list;
60
61 struct timer_list* init_timer_list();
62 uint32_t set_timer(struct timer_list *list, uint32_t sec, uint64_t nano_sec, void *(*callback)(void *, void *), void *args);
63 int check_timer(tju_tcp_t *sock);
64 uint32_t Set_timer_without_mutex(struct timer_list *list, uint32_t sec, uint64_t nano_sec, void *(*callback)(void *, void *), void *args);
65 int destroy_timer(tju_tcp_t *sock, uint32_t id);
66 void get_list_size(struct timer_list *list);
67 void print_timers(timer_list *list);
68 uint64_t get_create_time(timer_list *list, int id);
69
70#endif // __LIST_H__
```

图 4-1 timer_list 数据结构

queue 是对包括发送缓冲区、全连接队列以及清除队列（异步清除 timer）的具体实现。头文件如图4.2.1所示。可以看到，我们使用 void* 表示当前结点装载的

数据，这样实现的好处就是高扩展性，能够用来装载任何自定义的数据结构。缺点当然是使用不当会导致调用队列错乱。

```
t/s/tju_tcp.c | t/s/tran.c | t/i/timer_helper.h | t/i/queue.h+
24 #ifndef __QUEUE_H__
25 #define __QUEUE_H__
26 #include "global.h"
27 #include "debug.h"
28 #include <stdio.h>
29 #define MAX_QUEUE_SIZE 2*INIT_WINDOW_SIZE
30
31 typedef struct myNode{
32     void *data;
33     struct myNode *next;
34 }myNode;
35
36 typedef struct myQueue{
37     int size;
38     struct myNode *head;
39     struct myNode *tail;
40     pthread_mutex_t q_lock;
41 }myQueue;
42
43 myQueue* init_q();
44 int push_q(myQueue* q, void *data);
45 void *pop_q(myQueue* q);
46 int is_empty_q(myQueue *q);
47 int is_full_q(myQueue* q);
48
49#endif // __QUEUE_H__
```

图 4-2 queue 数据结构头文件

```
t/i/tree.h+
1 #ifndef __TREE_H__
2 #define __TREE_H__
3 // Define a binary tree with <Key, Value> Pair search Function
4 #include "global.h"
5 #include "debug.h"
6 // Create Node
7 typedef struct treeNode{
8     int key;
9     void *value;
10    struct treeNode*left;
11    struct treeNode*right;
12    int height;
13 }treeNode;
14 typedef struct myTree{
15     struct treeNode *root;
16     uint32_t size;
17 }myTree;
18 tju_packet_t* get_value(struct myTree *root, int key);
19 void free_tree(myTree* root);
20 void insert_key_value(myTree *node, int key, void *value);
21 void print_tree(myTree *root);
22 struct myTree* init_tree();
23 void remove_blow(struct treeNode* root, int key);
24#endif // __TREE_H__
```

图 4-3 tree 数据结构头文件

tree 是对接收缓冲区的具体实现，在此我们采用了 AVL 的数据结构（自平衡搜索二叉树，aka 小根堆）。因为我们的应用场景是：存入乱序到达的包裹，并按照正序进行取出。使用 AVL 可将二者操作时间复杂度降到 $\log_2(N)$ 。图4.2.1是 tree 的头文件。

4.2.2 实现思想

可靠数据传输的实现较为复杂。首先我们从发送线程的角度考虑，需要对发送的每个报文进行超时超时重传的检测。同时在接收到 ACK 后需要累计地将这些超时重传标签丢掉。

首先是在调用 `tju_send` 时，按照原来的做法，是使用 `send_buf` 进行发送。我们考虑到这个调用的流程逻辑本质上就是一种队列的思想：先从 `tju_send` 进入发送队列的报文先发送。于是我们复用了半/全连接队列时创建的 `QUEUE` 数据结构，让 `tju_send` 将报文放入 `sending_queue` 中。再让发送线程通过 `sending_queue` 取出需要发送的报文即可。

然后我们来处理超时重传：建立一个 `timer_list` 结构，用于存放已经发送且未得到 ACK 的报文。在发送线程中，我们首先对这个 `timer_list` 进行检查：按照顺序依次检查每个报文创建的时间和当前时间对比，如果有超时，则进行重发，否则直接退出。由于这个 `timer_list` 特性，天然按照报文创建的时间进行排序，所以可以在第一次遇到不用重发时退出。

然后我们从接收写成的角度进行考虑：我们会接收到三种类型的报文：`seq` 小于、等于以及大于 `expected_seq`。所以我们的思路也就很明朗了：小于时将其丢弃，大于时将其放入之前定义的 AVL 树中，等于时将其放入 `recv_buf` 并更新 `expected_seq`，并在 AVL 树中按照 `seq` 的大小查找是否有乱序达到的报文。当然，三种情况都需要发送 ACK 回去通知收到。

4.2.3 具体实现

Tju_Send 实现 在 `Tju_Send` 中，首先需要判断上层内容的大小，若过大，则需要进行切片，以免在下层被切割。然后在进入阻塞状态，等待发送窗口出现空闲。然后交给发送窗口，同时调用发送函数，在 `rwnd` 的情况下将发送缓冲区的内容进行发送（此处后期需要调整为线程）

Tju_Recv 实现 在 `Tju_Recv` 中，首先进行阻塞等待，直到其他线程给接收缓冲区放入内容。待有内容后，再将内容通过 `Memcpy` 提交给上层调用的实体。

Socket 数据结构

- 给 `window wnd_send` 增加了 `rwnd` 和 `cwnd` 等控制发送的信息

- 给 window wnd_recv 增加了 AVL Tree 等数据结构，用于存放乱序到达的数据包
- 给 window wnd_send 增加了 rto 等字段，动态控制超时事件
- 给 window wnd_recv 增加了 expe_seq 等字段，判断接收到的数据顺序正确与否

超时重传机制

超时重传定时器 实现了 Timer_Helper 子系统，通过设置 set_timer() 函数来新建一个 Timer，Delete_timer() 来终止一个 Timer。使用了一个线程不断检查是否有超时的 Timer，并进行重传和重置。使用 Mutex Lock 进行存储区域的一致性，使得在增加、删除和检查时只能有一个线程存在。

特别的，我们使用了链表的数据结构进行 Timer 的增加、删除和检查，链表能够高效地进行增加删除。同时，设置了 event 数据类型来处理当前 Timer 的超时操作（可以通过传入不同的函数和变量达到不同类型 Timer 的统一调用）。

RTO 调整 由于我们每个需要重传的报文都对应一个 TIMER，所以我们能通过发送时的 Created_At 和删除时的系统事件算出当前 Timer 提供的 sampleRTT 并在每次进行 Timer 删除时，通过公式 3.3.2 进行 RTO 的更新。

值得注意的是，这里依然需要大量使用 Mutex Locker 来控制：在更新 RTO 时，不能创建新的 Timer，直到 RTO 更新完成。

4.3 流量控制

接收方发送 ACK 时，将自己缓冲区大小放入 advertised_wind 字段。发送方在接收到 ACK 时，需要提取 rwnd 值。在设置发送窗口的大小时，我们需要考虑 rwnd（发送窗口大小），cwnd（拥塞控制），以及自己的大小。

当得到 rwnd == 0 的情况时，发送方需要发送 1 比特的试探报文进行发送窗口的试探，同时需要设置超时机制进行不断试探。

4.4 连接关闭的实现

在实现连接关闭之前，我们需要统一说明一下目前 tju_handle_packet 中的 socket 状态种类，以便在后续的流程中解释更加清晰。

连接建立 LISTEN, SYN_SENT, SYN_RECV, ESTABLISHED

连接关闭 FIN_WAIT_1, FIN_WAIT_2, CLOSE_WAIT, CLOSING, LAST_ACK, TIME_WAIT

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一方先一方后。我们根据绘制的 FSM 图可以得出结论：我们根据 socket 状态以及接收到的报文进行状态的转换以及处理：

1. 处于 **ESTABLISHED** 且发起 **close** 发送 FIN 状态转化为 **FIN_WAIT_1**
2. 处于 **FIN_WAIT_1** 且收到 FIN 的 ACK 状态转化为 **TIME_WAIT** 超时后变为 **CLOSED**
3. 处于 **ESTABLISHED** 收到 FIN 状态变为 **CLOSE_WAIT** 并返回 ACK
4. 处于 **CLOSE_WAIT** 且发起 **close** 状态转为 **LAST_ACK** 且等待 ACK
5. 处于 **LAST_ACK** 且收到 ACK 状态转为 **CLOSED** 并关闭

实现时需要注意的是，只要 socket 状态不是 **CLOSED**，都要继续接收新的报文并回复 ACK。

4.5 拥塞控制的实现

拥塞控制的实现核心在于两个事件触发拥塞窗口进行变化，且发送窗口随之而变。两个事件分别是：3 个 ACK，以及超时重传。

3 个 ACK 检测 在 sock 中的 cwnd 数据结构中加入 last_ack 以及 count 的整型数据，在 received_ack 函数中对 last_ack 进行比对并记录 count 是否增加或还原。

超时检测 在 check_timer 中触发超时转换 cwnd 的大小。

需要注意的是需要创建一个 sshresh 对 cwnd 慢启动和拥塞避免进行转换。同时，需要记得在 3 个 ack 后将状态调整为拥塞避免，在超时后将状态矫正为慢启动。

日志记录

实现了日志 trace 模块，在 Server Client 端调用 tju_sock 的时候进行日志的初始化（即定义日志名，创建日志文件等）。然后通过提供的日志格式和事件，在相应事件发生时，调用响应函数进行日志的录入。

五 实验结果及分析

实验测试环境

通过 VirtualBox 虚拟机进行测试，脚本默认丢包率 6%，延迟 6ms。系统环境如图一所示。测试方法是使用脚本和修改后的测试脚本进行运行，分析输出的 log 进行 debug。

5.1 连接建立的功能测试与结果分析

连接建立的 Log 文件如图5-1所示：

```

vagrant@server:/vagrant/tju_tcp$ ./server
vagrant@client:/vagrant/tju_tcp$ ./client

[...]

```

a) server 端

```

[...]

```

b) client 端

```

[...]

```

图 5-1 连接创建 Log

可以看到，Client 端首先创建了第一个 SYN 报文并进行发送，然后由于初次设置的 Timer 并不合理，立马出现了 timeout，进行重传，在接收到 SYN | ACK 后确认建立全连接，并发送 ACK 同时删除一个 timer。Server 端接收到 SYN 报文后进行 SYN | ACK 的返回，最后接收到 ACK 后确定一个 Full Connection 的建立。

5.2 可靠传输的功能测与结果分析

5.2.1 RTO 动态调整

如图5-2 是在 100Mbps 20ms 延迟的情况下观测到的数据，可见，我们的 TimeoutInterval 的确是随事件变化、随 EstimatedRTT 变化的。

如图5-3是 RTO 根据获得的 EstimatedRTT 进行的调整

5.2.2 可靠数据传输

我们通过 Client 端的 Trace (如图5-4) 可以看到接收到连续 ACK 表明接收到正确的信息。更多测试，在后续的性能中进行展示。

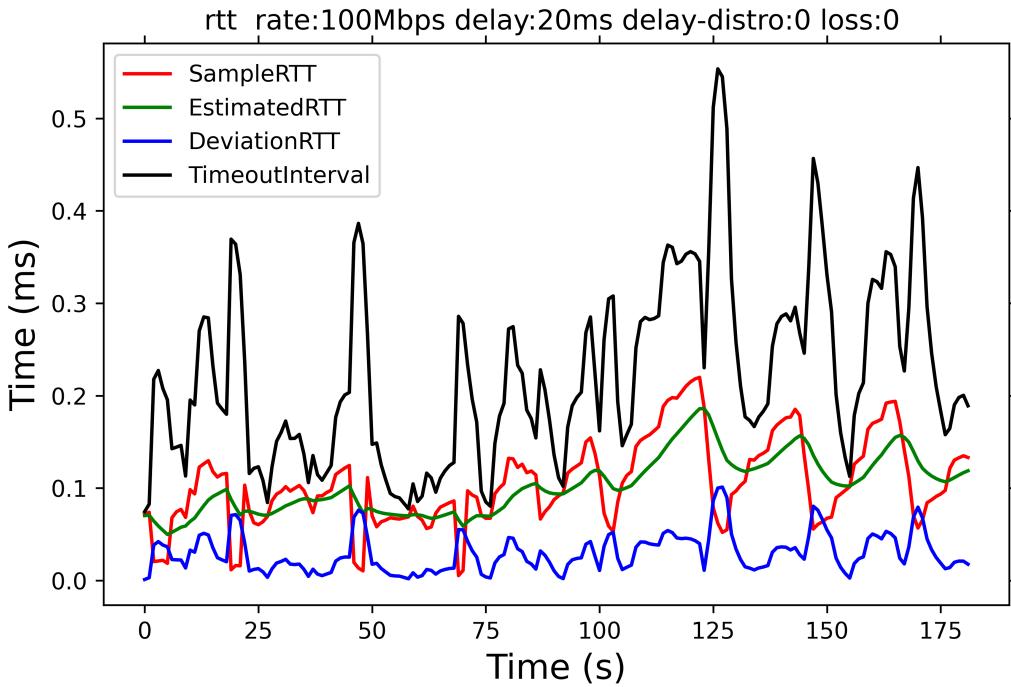


图 5-2 RTT 动态调整图

```

50 [1664877134018] [RTTS] [SampleRTT:0.657533 EstablishedRTT:0.626699 DeviationRtt:0.030946 TimeoutInterval:0.750483]
49 [1664877134019] [RWND] [size:44000]
48 [1664877134019] [RECV] [seq:0 ack:1146417 flag:ACK]
47 [1664877134020] [SEND] [seq:2139961 ack:0 flag:]
46 [1664877134021] [SEND] [seq:2139961 ack:0 flag:]
45 [1664877134021] [SEND] [seq:2141337 ack:0 flag:]
44 [1664877134022] [SEND] [seq:2141337 ack:0 flag:]
43 [1664877134022] [SWND] [size:44000]
42 [1664877134023] [RTTS] [SampleRTT:0.660575 EstablishedRTT:0.630934 DeviationRtt:0.033143 TimeoutInterval:0.763508]
41 [1664877134023] [RWND] [size:44000]
40 [1664877134023] [RECV] [seq:0 ack:1147793 flag:ACK]
39 [1664877134025] [SEND] [seq:2141713 ack:0 flag:]
38 [1664877134027] [SEND] [seq:2141713 ack:0 flag:]
37 [1664877134028] [RTTS] [SampleRTT:0.665238 EstablishedRTT:0.635222 DeviationRtt:0.034014 TimeoutInterval:0.771279]
36 [1664877134028] [RWND] [size:44000]
35 [1664877134029] [RECV] [seq:0 ack:1149169 flag:ACK]
34 [1664877134032] [RTTS] [SampleRTT:0.664263 EstablishedRTT:0.638852 DeviationRtt:0.030284 TimeoutInterval:0.759989]
33 [1664877134033] [RWND] [size:44000]
32 [1664877134034] [RECV] [seq:0 ack:1150545 flag:ACK]
31 [1664877134040] [RTTS] [SampleRTT:0.671368 EstablishedRTT:0.642917 DeviationRtt:0.031958 TimeoutInterval:0.770748]
30 [1664877134040] [RWND] [size:44000]
29 [1664877134041] [RECV] [seq:0 ack:1150921 flag:ACK]
28 [1664877134041] [SEND] [seq:2143089 ack:0 flag:]
27 [1664877134042] [SEND] [seq:2143089 ack:0 flag:]
26 [1664877134042] [SEND] [seq:2144465 ack:0 flag:]
25 [1664877134042] [SEND] [seq:2144465 ack:0 flag:]
24 [1664877134044] [SEND] [seq:2145841 ack:0 flag:]
23 [1664877134045] [SEND] [seq:2145841 ack:0 flag:]
22 [1664877134047] [SEND] [seq:2147217 ack:0 flag:]
21 [1664877134048] [SEND] [seq:2147217 ack:0 flag:]
20 [1664877134050] [RTTS] [SampleRTT:0.681771 EstablishedRTT:0.647773 DeviationRtt:0.037130 TimeoutInterval:0.796295]
19 [1664877134051] [RWND] [size:44000]
18 [1664877134051] [RECV] [seq:0 ack:1232361 flag:ACK]
17 [1664877134051] [SEND] [seq:2148593 ack:0 flag:]
16 [1664877134052] [SEND] [seq:2148593 ack:0 flag:]
15 [1664877134053] [RTTS] [SampleRTT:0.683951 EstablishedRTT:0.652296 DeviationRtt:0.036416 TimeoutInterval:0.797959]
14 [1664877134053] [RWND] [size:44000]
13 [1664877134054] [RECV] [seq:0 ack:1152297 flag:ACK]
12 [1664877134054] [SEND] [seq:2149969 ack:0 flag:]
11 [1664877134055] [SEND] [seq:2149969 ack:0 flag:]
10 [1664877134055] [SEND] [seq:2151345 ack:0 flag:]
9 [1664877134055] [SEND] [seq:2151345 ack:0 flag:]
8 [1664877134056] [SWND] [size:44000]
7 [1664877134057] [SEND] [seq:2151721 ack:0 flag:]
6 [1664877134060] [SEND] [seq:2151721 ack:0 flag:]
5 [1664877134060] [RTTS] [SampleRTT:0.688660 EstablishedRTT:0.656841 DeviationRtt:0.036377 TimeoutInterval:0.802350]
4 [1664877134061] [RWND] [size:44000]
3 [1664877134061] [RECV] [seq:0 ack:1153673 flag:ACK]
2 [1664877134064] [SEND] [seq:2153097 ack:0 flag:]
1 [1664877134064] [SEND] [seq:2153097 ack:0 flag:]
6688 [1664877134068] [RTTS] [SampleRTT:0.696781 EstablishedRTT:0.661834 DeviationRtt:0.039049 TimeoutInterval:0.818029]
1 1664877134069] [RWND] [size:44000]

```

图 5-3 RTT 动态调整 Log

```

23 [1664877130334] [RECV] [seq:0 ack:1377 flag:ACK]
22 [1664877130334] [SEND] [seq:404449 ack:0 flag:;]
21 [1664877130334] [SEND] [seq:404449 ack:0 flag:;]
20 [1664877130334] [RTTS] [SamplerRTT:0.041259 EstablishedRTT:3.837762 DeviationRTt:-4.261212 TimeoutInterval:0.050000]
19 [1664877130334] [RWND] [size:42625]
18 [1664877130334] [RECV] [seq:0 ack:2753 flag:ACK]
17 [1664877130334] [SEND] [seq:405825 ack:0 flag:;]
16 [1664877130334] [SEND] [seq:405825 ack:0 flag:;]
15 [1664877130334] [RTTS] [SamplerRTT:0.041736 EstablishedRTT:3.363258 DeviationRTt:-3.974926 TimeoutInterval:0.050000]
14 [1664877130334] [RWND] [size:41250]
13 [1664877130334] [RECV] [seq:0 ack:4129 flag:ACK]
12 [1664877130339] [SEND] [seq:422717 ack:0 flag:;]
11 [1664877130340] [RTTS] [SamplerRTT:0.046789 EstablishedRTT:2.585482 DeviationRTt:-3.133166 TimeoutInterval:0.050000]
10 [1664877130340] [RWND] [size:38500]
9 [1664877130340] [RECV] [seq:0 ack:6881 flag:ACK]
8 [1664877130340] [RTTS] [SamplerRTT:0.047279 EstablishedRTT:2.268206 DeviationRTt:-2.757862 TimeoutInterval:0.050000]
7 [1664877130340] [RWND] [size:37125]
6 [1664877130340] [RECV] [seq:0 ack:8257 flag:ACK]
5 [1664877130340] [SEND] [seq:428593 ack:0 flag:;]
4 [1664877130340] [RTTS] [SamplerRTT:0.047465 EstablishedRTT:1.990614 DeviationRTt:-2.426219 TimeoutInterval:0.050000]
3 [1664877130340] [RWND] [size:35750]
2 [1664877130340] [RECV] [seq:0 ack:9633 flag:ACK]
1 [1664877130341] [SEND] [seq:428593 ack:0 flag:;]
526 [1664877130341] [RTTS] [SamplerRTT:0.048127 EstablishedRTT:1.747803 DeviationRTt:-2.135610 TimeoutInterval:0.050000]
1 [1664877130341] [RWND] [size:34375]
2 [1664877130341] [RECV] [seq:0 ack:10009 flag:ACK]
3 [1664877130341] [RTTS] [SamplerRTT:0.048323 EstablishedRTT:1.535368 DeviationRTt:-1.880997 TimeoutInterval:0.050000]
4 [1664877130341] [RWND] [size:34000]

```

图 5-4 接收 Seq 跟随变化图

5.3 流量控制的功能测试与结果分析

为最大限度地选出优秀的结果，我们将家庭 NAS 服务器系统装上了 Arch Linux 并安装 vagrant 等进行大量脚本测试，跑出了 1728 个结果（如图5-5），精选后，我们选出一个较为优秀的结果精选展示。

```

christopher@homeserver:~/christopher/Coding/netProj/tju_tcp/figure/rate:1000Mbpsdelay:1000msdelay:distro:0loss:30
build
figure
figure_too_many
inc
src
test
test_
test_
archive.zip
Client
handler.zip
Makefile
rxt_recv_file.txt
rxt_send_file.txt
server
tags
test_Makefile
test_tree
43 rate:1000Mbpsdelay:1000msdelay:distro:0loss:30
42 rate:1000Mbpsdelay:1000msdelay:distro:0loss:40
41 rate:1000Mbpsdelay:1000msdelay:distro:0loss:0
40 rate:1000Mbpsdelay:1000msdelay:distro:100loss:10
39 rate:1000Mbpsdelay:1000msdelay:distro:100loss:20
38 rate:1000Mbpsdelay:1000msdelay:distro:100loss:30
37 rate:1000Mbpsdelay:1000msdelay:distro:100loss:40
36 rate:1000Mbpsdelay:1000msdelay:distro:200loss:0
35 rate:1000Mbpsdelay:1000msdelay:distro:200loss:10
34 rate:1000Mbpsdelay:1000msdelay:distro:200loss:20
33 rate:1000Mbpsdelay:1000msdelay:distro:200loss:30
32 rate:1000Mbpsdelay:1000msdelay:distro:200loss:40
31 rate:1000Mbpsdelay:1000msdelay:distro:300loss:0
30 rate:1000Mbpsdelay:1000msdelay:distro:300loss:10
29 rate:1000Mbpsdelay:1000msdelay:distro:300loss:20
28 rate:1000Mbpsdelay:1000msdelay:distro:300loss:30
27 rate:1000Mbpsdelay:1000msdelay:distro:400loss:0
26 rate:1000Mbpsdelay:1000msdelay:distro:400loss:10
25 rate:1000Mbpsdelay:1000msdelay:distro:400loss:20
24 rate:1000Mbpsdelay:1000msdelay:distro:400loss:30
23 rate:1000Mbpsdelay:1000msdelay:distro:500loss:0
22 rate:1000Mbpsdelay:1000msdelay:distro:500loss:10
21 rate:1000Mbpsdelay:1000msdelay:distro:500loss:20
20 rate:1000Mbpsdelay:1000msdelay:distro:500loss:30
19 rate:1000Mbpsdelay:1000msdelay:distro:600loss:0
18 rate:1000Mbpsdelay:1000msdelay:distro:600loss:10
17 rate:1000Mbpsdelay:1000msdelay:distro:600loss:20
16 rate:1000Mbpsdelay:1000msdelay:distro:600loss:30
15 rate:1000Mbpsdelay:1000msdelay:distro:700loss:0
14 rate:1000Mbpsdelay:1000msdelay:distro:700loss:10
13 rate:1000Mbpsdelay:1000msdelay:distro:700loss:20
12 rate:1000Mbpsdelay:1000msdelay:distro:700loss:30
11 rate:1000Mbpsdelay:1000msdelay:distro:800loss:0
10 rate:1000Mbpsdelay:1000msdelay:distro:800loss:10
9 rate:1000Mbpsdelay:1000msdelay:distro:800loss:20
8 rate:1000Mbpsdelay:1000msdelay:distro:800loss:30
7 rate:1000Mbpsdelay:1000msdelay:distro:900loss:0
6 rate:1000Mbpsdelay:1000msdelay:distro:900loss:10
5 rate:1000Mbpsdelay:1000msdelay:distro:900loss:20
4 rate:1000Mbpsdelay:1000msdelay:distro:900loss:30
3 rate:1000Mbpsdelay:1000msdelay:distro:1000loss:0
2 rate:1000Mbpsdelay:1000msdelay:distro:1000loss:10
1 rate:1000Mbpsdelay:1000msdelay:distro:1000loss:20
0 sum, 1796 free 1729/1729 Bot
dixxx-xx-x 2 christopher christopher 10 2022-10-17 03:40

```

图 5-5 家庭 NAS 系统结果

我们选出在 100Mbps 20ms 延迟环境下的测试结果5-6。显示我们的流量控制效果是合理的，能够将窗口的变化实时体现在 swnd 中。

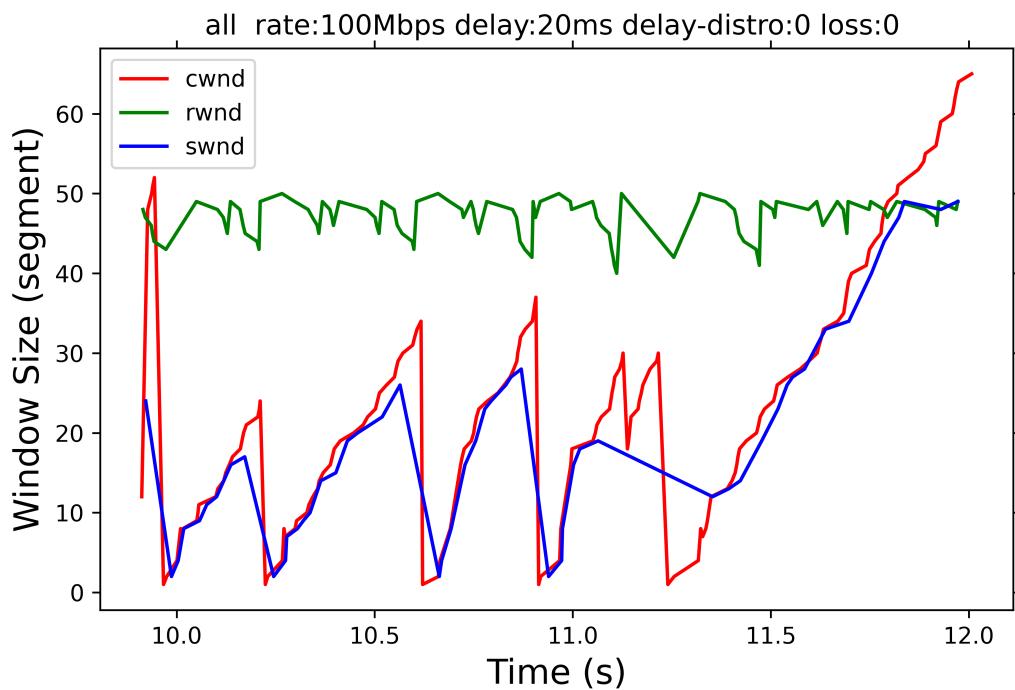


图 5-6 流量控制

5.4 连接关闭的功能测试与结果分析

我们顺利在 Auto Lab 上提交了正确答案，确实没太多特性需要测试，在此附上一个 AutoLab 上的截图证明（如图5-7）。

[\[双方同时关闭测试\]](#) 进行评分
[\[双方同时关闭测试\]](#) 双方同时关闭测试-服务器端部分的得分为 20 分 (满分20分)
[\[断开连接的测试\]](#) 两种情况的总得分为 100 分 (满分100分)

图 5-7 连接关闭证明

5.5 拥塞控制的功能测试与结果分析

经过脚本测试，我们选出了在环境 100Mbps 20ms 延迟的数据进行展示，可以看到，我们的 cwnd 实现能够在正确地情况出现时给予正确的处理：即 1. 刚开始时采取慢启动。2. 达到 sshresh 时采取拥塞避免。3. 在发生超时的同时将窗口大小降低并进行慢启动。4. 出现三次 ack 时降低一般的窗口。整个图形呈现锯齿状

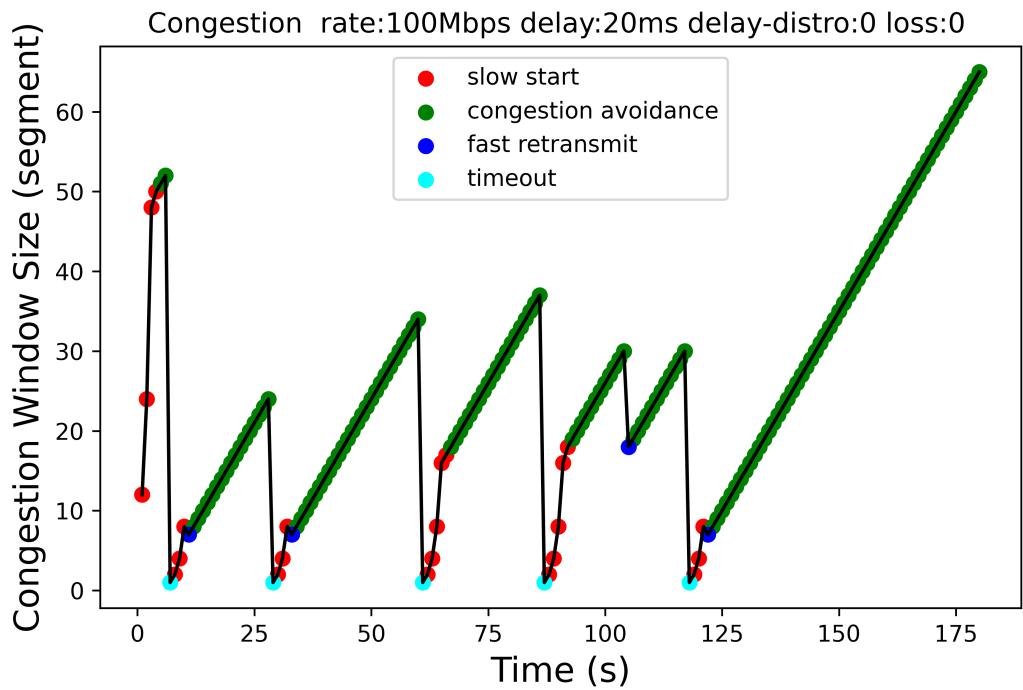


图 5-8 Congestion 窗口变化图

5.6 TCP 协议性能测试与结果分析

我们分别对吞吐率-窗口大小和吞吐率-丢包率两种情况做了测试（如图5-10和5-9所示）。

窗口大小 可见，随着窗口大小的增加，吞吐率有明显地增大。但在较大的两个数据：56 和 64 之间的吞吐率差距较小。这可能是因为我们仍然设置了对发送缓冲区的大小做了限制：即发送端在发送缓冲区满的时候需要停下来等待。且每次都需要对环境进行加锁、解锁，这些流程也会使得整体速率是有上限的。也就是说我们的表现是正确的。

丢包率 可见，吞吐率随着丢包率的提高而下降，我们选取的窗口大小为 32，符合图5-9 的数据。同时，在产生 1% 丢包率时，其吞吐率相较没有丢包率时有明显的下降。这表明在 TCP 现在的标准下，产生丢包等误判会严重降低吞吐率，我们或许有更好的设计思路能够提升这一情况下的吞吐率。

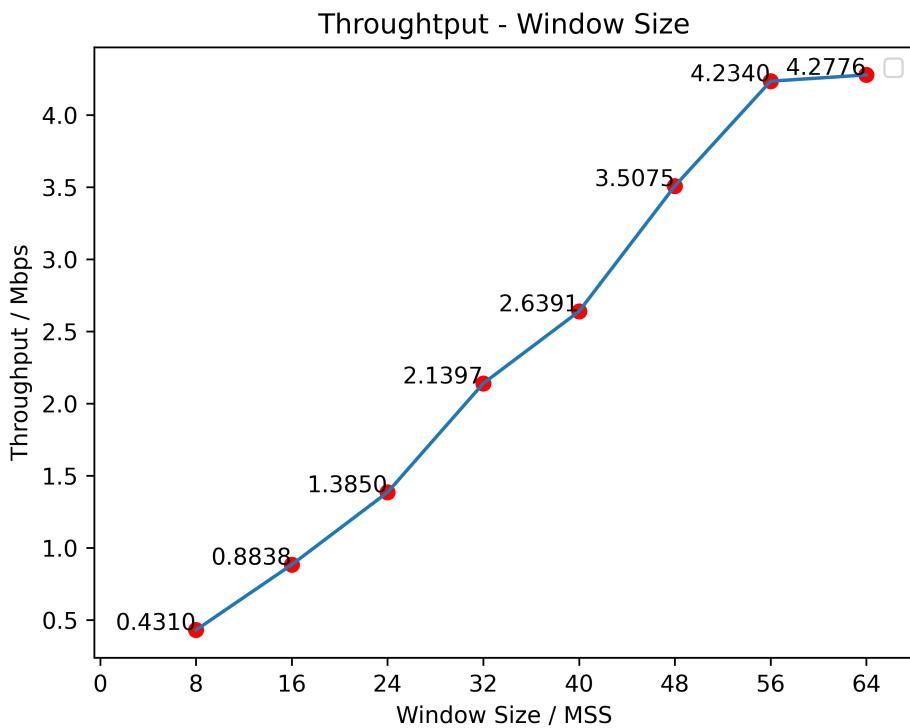


图 5-9 吞吐率-窗口大小

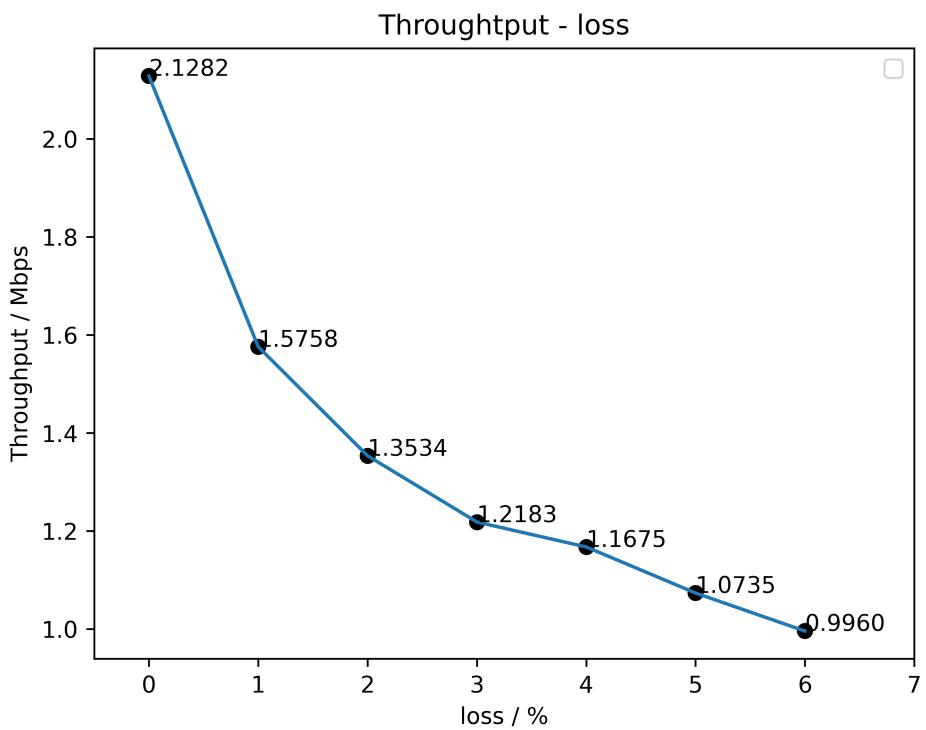


图 5-10 吞吐率-丢包率

六 总结

在本次实践中，我们顺利完成了每次任务，并拿到了每次 AutoLab 的满分。但在其中我们也遇到了不少困难。

环境配置 由于本人使用习惯需要在 Arch Linux 环境下进行实验，于是在了解 vagrant virtualbox 等环境作用原理之后，最终成功在 arch linux 环境下完成了实验。

tree 在设计接收乱序缓冲区时，我们首先考虑了直接使用 `recv_buf` 和一些指针进行操作。但这样的操作错误率太高，且耦合性过大，需要在每次放入、取出时频繁操作指针，不利于后续的维护更新。于是我们不仅将放入、取出的操作进行了封装。同时，由于考虑到这个实际应用的操作逻辑，结合自身在 ACM 中学到的数据结构，我果断考虑到了小根堆，但由于是 C 的环境，于是不得不手写一个小根堆，并做一层封装。

反复重复测试 在拥塞控制的测试中，我们始终难以得到心仪的结果，于是我们利用家里的一台闲置的主机，临时配置了 Arch Linux 和一些基础设施，通过 `ssh -R` 连接到服务器上，于是我们能够在学校也远程使用。最终我们编写脚本，在机器上跑了 3 天完成了 1700 多个测试结果，再使用 `vim` 及其脚本编写了一个 `html` 用于，在浏览器上找到了比较出色的几个结果用于报告。

总的来说，这次实践加深了我对 TCP 协议的理解和一些思考，也让我们能更加理性地对日常遇到的网络问题想出正确的策略。但仍有很多网络现象是我们尚不能解释的：例如不能在校园网环境下通过 `ssh` 连接另一台电脑等等。对于这些现象，我们十分感兴趣，在未来的日子里我们也会继续提升对计算机网络的认识和理解，争取回答这些问题。