Lab of Computer Network: TCP Stack **Fall 2024**

Report 12 — December 3

Lecturer: Wu Qinghua *Completed by: 2022K8009929010 Zhang Jiawei*

12.1 实验内容

1. 实验内容一：连接管理
   (1) 运行给定网络拓扑 (tcp_topo.py)
   (2) 在节点 h1 上执行 TCP 程序
      i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
      ii. 在 h1 上运行 TCP 协议栈的服务器模式 (./tcp_stack server 10001)
   (3) 在节点 h2 上执行 TCP 程序
      i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
      ii. 在 h2 上运行 TCP 协议栈的客户端模式，连接至 h1，显示建立连接成功后自动断开连接 (./tcp_stack client 10.0.0.1 10001)
   (4) 可以在一端用 tcp_stack_conn.py 替换 tcp_stack 执行，测试另一端
   (5) 通过 wireshark 抓包来来验证建立和断开连接的正确性
2. 实验内容二：短消息收发
   (1) 参照 tcp_stack_trans.py，修改 tcp_apps.c，使之能够收发短消息
   (2) 运行给定网络拓扑 (tcp_topo.py)
   (3) 在节点 h1 上执行 TCP 程序
      i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
      ii. 在 h1 上运行 TCP 协议栈的服务器模式 (./tcp_stack server 10001)
   (4) 在节点 h2 上执行 TCP 程序
      i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
      ii. 在 h2 上运行 TCP 协议栈的客户端模式，连接至 h1，发送短消息 (./tcp_stack client 10.0.0.1 10001)。即 client 向 server 发送数据，server 将数据 echo 给 client
   (5) 使用 tcp_stack_trans.py 替换其中任意一端，对端都能正确收发数据
3. 实验内容三：大文件传送
   (1) 修改 tcp_apps.c(以及 tcp_stack_trans.py)，使之能够收发文件
   (2) 执行 create_randfile.sh，生成待传输数据文件 client-input.dat

(3) 运行给定网络拓扑 (tcp_topo.py)

(4) 在节点 h1 上执行 TCP 程序

  i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh),禁止协议栈的相应功能

  ii. 在 h1 上运行 TCP 协议栈的服务器模式 (./tcp_stack server 10001)

(5) 在节点 h2 上执行 TCP 程序

  i. 执行脚本 (disable_tcp_rst.sh, disable_offloading.sh),禁止协议栈的相应功能

  ii. 在 h2 上运行 TCP 协议栈的客户端模式,连接至 h1,发送文件 (./tcp_stack client 10.0.0.1 10001)

(6) 使用 md5sum 比较两个文件是否完全相同

(7) 使用 tcp_stack_trans.py 替换其中任意一端,对端都能正确收发数据

12.2 实验过程

12.2.1 socket 连接

对于客户端主动发起连接的情况，先由 socket 信息确定四元组 (`sip, sport, dip, dport`)；接着发出 SYN 数据包，设置状态为 `TCP_SYN_SENT`；再对 socket 进行哈希，插入到 `bind_table` 中；然后进入 `sleep_on`,等待服务器返回的 SYN 包;如果收到了 SYN 包,唤醒线程,说明连接建立成功:

```c
// connect to the remote tcp sock specified by skaddr
//
// XXX: skaddr here contains network-order variables
// 1. initialize the four key tuple (sip, sport, dip, dport);
// 2. hash the tcp sock into bind_table;
// 3. send SYN packet, switch to TCP_SYN_SENT state, wait for the incoming
//    SYN packet by sleep on wait_connect;
// 4. if the SYN packet of the peer arrives, this function is notified, which
//    means the connection is established.
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    int err = 0;

    tsk->sk_dip = ntohl(skaddr->ip);
    tsk->sk_dport = ntohs(skaddr->port);
    rt_entry_t *entry = longest_prefix_match(tsk->sk_dip);
    if (!entry) {
        log(ERROR, "no route to "IP_FMT".", HOST_IP_FMT_STR(tsk->sk_dip));
        return -1;
    }
    tsk->sk_sip = entry->iface->ip;
```

```
    err = tcp_sock_set_sport(tsk, 0);
    if (err) {
        log(ERROR, "tcp sock set sport failed.");
        return err;
    }

    tcp_set_state(tsk, TCP_SYN_SENT);
    err = tcp_hash(tsk);
    if (err) {
        log(ERROR, "tcp sock hash failed.");
        return err;
    }

    tcp_send_control_packet(tsk, TCP_SYN);

    err = sleep_on(tsk->wait_connect);
    if (err) {
        log(ERROR, "sleep on wait_connect failed.");
        return err;
    }

    return 0;
}
```

对于服务器来说，需要主动监听端口，等待客户端的连接请求。具体操作是设置 backlog，然后设置状态为 `TCP_LISTEN`，将 socket 哈希之后插入到 `listen_table` 中，等待客户端的连接请求。如果 `accept_queue` 空，说明没有成功连接的 socket，则取出队列中第一个 TCP socket 接受连接：

```
// set backlog (the maximum number of pending connection requst), switch the
// TCP_STATE, and hash the tcp sock into listen_table
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    log(DEBUG, "listening on port %hu.", tsk->sk_sport);
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    return tcp_hash(tsk);
}

// if accept_queue is not emtpy, pop the first tcp sock and accept it,
// otherwise, sleep on the wait_accept for the incoming connection requests
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
```

```
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    while (list_empty(&tsk->accept_queue))
        sleep_on(tsk->wait_accept);

    return tcp_sock_accept_dequeue(tsk);
}
```

当需要关闭连接时，分为两种情况，即主动关闭和被动关闭。若当前状态为 TCP_ESTABLISHED 或者 TCP_SYN_RECV 状态，则为主动关闭连接，向对方发送 FIN 和 ACK 信号，将状态变为 TCP_FIN_WAIT_1；若当前状态为 TCP_CLOSE_WAIT 状态，则为被动断开连接，向对方发送 FIN 和 ACK 信号，将状态变为 TCP_LAST_ACK。需要另外说明的是，如果遇到其他状态，则直接断开连接：

```
// close the tcp sock, by releasing the resources, sending FIN/RST packet
// to the peer, switching TCP_STATE to closed
void tcp_sock_close(struct tcp_sock *tsk)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

    log(DEBUG, "close tcp sock: "IP_FMT":%hu -> "IP_FMT":%hu, state: %s.",
            HOST_IP_FMT_STR(tsk->sk_sip), tsk->sk_sport,
            HOST_IP_FMT_STR(tsk->sk_dip), tsk->sk_dport,
            tcp_state_str[tsk->state]);
    switch(tsk->state) {
        case TCP_SYN_RECV:
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            break;

        case TCP_ESTABLISHED:
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            break;

        case TCP_CLOSE_WAIT:
            tcp_set_state(tsk, TCP_LAST_ACK);
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            break;

        default:
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
            break;
```

```
    }

    free_tcp_sock(tsk);
}
```

此外，用户进程还需要从缓存区读取数据，以及还向用户进程发送数据。这里我使用两个函数来实现上述功能。

对于读取数据，首先判断缓存区是否为空，还需判断 TCP 状态是否为 TCP_CLOSE_WAIT，若不是则进入 sleep_on 等待，否则直接返回 0，表明之后不会再有数据到达。当缓冲区不为空时，置起互斥锁，将数据从缓冲区中读出来，然后释放互斥锁，返回读取的数据长度：

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)
{
    pthread_mutex_lock(&tsk->rcv_buf_lock);
    while (ring_buffer_empty(tsk->rcv_buf)) {
        if (tsk->state == TCP_CLOSED || tsk->state == TCP_LAST_ACK || tsk->state ==
            TCP_CLOSE_WAIT) {
            pthread_mutex_unlock(&tsk->rcv_buf_lock);
            return 0;
        }
        else {
            pthread_mutex_unlock(&tsk->rcv_buf_lock);
            sleep_on(tsk->wait_recv);
            pthread_mutex_lock(&tsk->rcv_buf_lock);
        }
    }

    int rlen = min(len, ring_buffer_used(tsk->rcv_buf));
    read_ring_buffer(tsk->rcv_buf, buf, rlen);
    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);
    pthread_mutex_unlock(&tsk->rcv_buf_lock);

    return rlen;
}
```

对于发送数据，先确定需要发送的数据包长度。因为以太网帧的最大长度为 1514 字节，所以数据包长度太长时，需要分片发送。接着将数据包封装成以太网帧，发送数据包：

```
int tcp_send_data(struct tcp_sock *tsk, char *buf, int len)
{
    len = min(len, ETH_FRAME_LEN - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE);
    int pkt_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE + len;
    char *packet = malloc(pkt_len);
```

```c
    char *payload = packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;
    memcpy(payload, buf, len);

    while (tsk->snd_wnd < len) {
        log(DEBUG, "wait for sending window.");
        tsk->snd_wnd = 0;
        sleep_on(tsk->wait_send);
    }

    tcp_send_packet(tsk, packet, pkt_len);
    return len;
}

int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)
{
    while (len > 0) {
        int wlen = tcp_send_data(tsk, buf, len);
        buf += wlen;
        len -= wlen;
    }
    return len;
}
```

12.2.2 TCP 连接与数据包处理

我们需要根据传入的 TCP 报文找到其对应的 socket,需要查找的是 `listen_table` 和 `established_table`。查找 `listen_table` 时,以 `sport` 为关键字,查找 `established_table` 时,以 `dip, dport, sip, sport` 为关键字。若找到了对应的 socket 则返回:

```c
// lookup tcp sock in established_table with key (saddr, daddr, sport, dport)
struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    int hash = tcp_hash_function(saddr, daddr, sport, dport);
    struct list_head *list = &tcp_established_sock_table[hash];

    struct tcp_sock *tsk;
    list_for_each_entry(tsk, list, hash_list) {
        if (saddr == tsk->sk_sip && daddr == tsk->sk_dip &&
            sport == tsk->sk_sport && dport == tsk->sk_dport)
            return tsk;
    }
```

```c
    return NULL;
}

// lookup tcp sock in listen_table with key (sport)
//
// In accordance with BSD socket, saddr is in the argument list, but never used.
struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    int hash = tcp_hash_function(0, 0, sport, 0);
    struct list_head *list = &tcp_listen_sock_table[hash];

    struct tcp_sock *tsk;
    list_for_each_entry(tsk, list, hash_list) {
        if (sport == tsk->sk_sport)
            return tsk;
    }

    return NULL;
}

// lookup tcp sock in both established_table and listen_table
struct tcp_sock *tcp_sock_lookup(struct tcp_cb *cb)
{
    u32 saddr = cb->daddr,
        daddr = cb->saddr;
    u16 sport = cb->dport,
        dport = cb->sport;

    struct tcp_sock *tsk = tcp_sock_lookup_established(saddr, daddr, sport, dport);
    if (!tsk)
        tsk = tcp_sock_lookup_listen(saddr, sport);

    return tsk;
}
```

对于接收到的数据包，先判断数据包长度是否合法，若合法则检查缓冲区是否有足够的空间存放数据包，若有则将数据包存入缓冲区，若没有则返回 0。在写入缓存时依然需要置起互斥锁，同时还需要实现流量控制，即将接收窗口赋值为缓存区的剩余空间：

```c
// handle the recv of the incoming TCP packet
int handle_tcp_recv(struct tcp_sock *tsk, struct tcp_cb *cb)
{
```

```
    if (cb->pl_len <= 0)
        return 0;

    pthread_mutex_lock(&tsk->rcv_buf_lock);
    if (cb->pl_len > ring_buffer_free(tsk->rcv_buf)) {
        log(ERROR, "no enough space in rcv_buf, drop the packet.");
        pthread_mutex_unlock(&tsk->rcv_buf_lock);
        return 0;
    }
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);
    wake_up(tsk->wait_recv);
    pthread_mutex_unlock(&tsk->rcv_buf_lock);
    return 1;
}
```

接下来是根据 TCP 状态机处理接收到的数据包，这是本次实验中最重要的部分。先检查是否存在 socket 连接，再检查是否为 RST 包，若是则直接关闭连接。然后进入状态机处理，根据不同的状态处理不同的数据包。

对于 `TCP_LISTEN` 状态，若是 SYN 包，则应该建立连接，创建新的子 socket 连接，状态设置为 `TCP_SYN_RECV`，并发送 SYN 和 ACK 包。

对于 `TCP_SYN_SENT` 状态，若是 SYN 和 ACK 包，说明服务器已经接受了连接请求，然后更新 `rcv_nxt` 和 `snd_una`，状态设置为 `TCP_ESTABLISHED`，并发送 ACK 包，唤醒等待连接的进程；如果只收到了 SYN 包，表明对方想要主动建立连接，进入 `TCP_SYN_RECV` 状态，发送 SYN 和 ACK 包。

对于 `TCP_SYN_RECV` 状态，若是 ACK 包，先检查序列号是否有效，若有效则表明三次握手成功，更新 `rcv_nxt` 和 `snd_una`。再检查 `accept_queue` 是否已满，若满则丢弃数据包，设置状态为 `TCP_CLOSED`，关闭连接；若不满则将 socket 插入 `accept_queue`，状态设置为 `TCP_ESTABLISHED`，唤醒等待连接的进程。

对于 `TCP_ESTABLISHED` 状态，先检查序列号是否有效，若无效则丢弃数据包。再检查如果有 ACK 标志，则更新 `snd_una`，若还有 FIN 标志，则进入 `TCP_CLOSE_WAIT` 状态，更新 `rcv_nxt`，发送 ACK 包，唤醒对端等待接收的进程，否则应该处理数据包，调用 `handle_tcp_recv` 函数，更新 `rcv_nxt`，发送 ACK 包。

对于 `TCP_FIN_WAIT_1` 状态，则本地已经发送了 FIN 包，等待对端的 ACK 包。对于接收到的数据包，先检查序列号是否有效，若无效则丢弃数据包。若是 FIN 与 ACK 包，且发送序列号等于确认序列号，则进入 `TCP_TIME_WAIT` 状态，设置定时器，发送 ACK 包。若只是 ACK 包，且发送序列号等于确认序列号，则进入 `TCP_FIN_WAIT_2` 状态。若只是 FIN 包，则进入 `TCP_CLOSING` 状态，发送 ACK 包。

对于 `TCP_FIN_WAIT_2` 状态，则本地已收到对端确认本地的 FIN 包，等待对端的 FIN 包。对于 `TCP_CLOSING` 状态，则本地已经收到对端的 FIN 包且已发送 ACK 包，等待对端的 ACK 包。这两个状态也只需要先检查序列号是否有效，若无效则丢弃数据包，然后更新 `rcv_nxt` 和 `snd_una`，如果接收到了该状态对应的数据包，则进入 `TCP_TIME_WAIT` 状态，设置定时器。对于 `TCP_FIN_WAIT_2` 状态还需要发送 ACK 包。

对于 `TCP_TIME_WAIT` 状态和 `TCP_CLOSE_WAIT` 状态，什么都不用做。

对于 `TCP_LAST_ACK` 状态，本地已经发送了 FIN 包，且已收到对端的确认，等待最终的 ACK 包以关闭连接。对于接收到的数据包，先检查序列号是否有效，若无效则丢弃数据包，然后更新 `rcv_nxt` 和 `snd_una`，若接收到了 ACK 包，且发送序列号等于确认序列号，则进入 `TCP_CLOSED` 状态，关闭连接，释放资源。

对于 TCP_CLOSED 状态,只需要释放资源即可。

```
// Process the incoming packet according to TCP state machine.
void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
   // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
   if (!tsk) {
      log(ERROR, "no tcp sock to process packet.\n");
      tcp_send_reset(cb);
      return;
   }
   if (cb->flags & TCP_RST) {
      tcp_set_state(tsk, TCP_CLOSED);
      tcp_unhash(tsk);
      tcp_bind_unhash(tsk);
      return;
   }

   switch (tsk->state) {
      case TCP_LISTEN:
         if (cb->flags == TCP_SYN) {
            struct tcp_sock *ctsk = alloc_tcp_sock();
            ctsk->parent = tsk;
            tsk->ref_cnt += 1;
            ctsk->local.ip = cb->daddr;
            ctsk->local.port = cb->dport;
            ctsk->peer.ip = cb->saddr;
            ctsk->peer.port = cb->sport;
            ctsk->iss = tcp_new_iss();
            ctsk->snd_nxt = ctsk->iss;
            ctsk->rcv_nxt = cb->seq_end;

            tcp_set_state(ctsk, TCP_SYN_RECV);
            tcp_hash(ctsk);
            init_list_head(&ctsk->bind_hash_list);
            log(DEBUG, "child "IP_FMT":%hu join the listen queue of parent
                "IP_FMT":%hu", HOST_IP_FMT_STR(ctsk->sk_sip), ntohs(ctsk->sk_sport),
                HOST_IP_FMT_STR(tsk->sk_sip), ntohs(tsk->sk_sport));
            list_add_tail(&ctsk->list, &tsk->listen_queue);

            tcp_send_control_packet(ctsk, TCP_SYN | TCP_ACK);
         }
         else
```

```
            log(ERROR, "received packet is not SYN but current state is LISTEN, drop
                it.");
        break;

    case TCP_SYN_SENT:
        if (cb->flags == (TCP_SYN | TCP_ACK)) {
            tsk->rcv_nxt = cb->seq_end;
            tcp_update_window_safe(tsk, cb);
            tsk->snd_una = cb->ack;
            tcp_set_state(tsk, TCP_ESTABLISHED);
            tcp_send_control_packet(tsk, TCP_ACK);
            wake_up(tsk->wait_connect);
        }
        else if (cb->flags == TCP_SYN) {
            tsk->rcv_nxt = cb->seq_end;
            tcp_set_state(tsk, TCP_SYN_RECV);
            tcp_send_control_packet(tsk, TCP_SYN | TCP_ACK);
        }
        else
            log(ERROR, "received packet is not SYN or SYN_ACK but current state is
                SYN_SENT, drop it.");
        break;

    case TCP_SYN_RECV:
        if (cb->flags == TCP_ACK) {
            if (!is_tcp_seq_valid(tsk, cb))
                return;
            tsk->rcv_nxt = cb->seq_end;
            tcp_update_window_safe(tsk, cb);
            tsk->snd_una = cb->ack;

            if (tsk->parent) {
                if (tcp_sock_accept_queue_full(tsk->parent)) {
                    tcp_set_state(tsk, TCP_CLOSED);
                    tcp_send_control_packet(tsk, TCP_RST);
                    tcp_unhash(tsk);
                    tcp_bind_unhash(tsk);

                    list_delete_entry(&tsk->list);
                    free_tcp_sock(tsk);
                    log(ERROR, "accept queue is full, drop this connection.");
                }
                else {
```

```
                tcp_set_state(tsk, TCP_ESTABLISHED);
                tcp_sock_accept_enqueue(tsk);
                wake_up(tsk->parent->wait_accept);
            }
        }
        else
            log(ERROR, "no parent tcp sock to accept child connection.");
    }
    else
        log(ERROR, "received packet is not ACK but current state is SYN_RECV, drop
            it.");
    break;

case TCP_ESTABLISHED:
    if (!is_tcp_seq_valid(tsk, cb))
        return;

    if (cb->flags & TCP_ACK) {
        tcp_update_window_safe(tsk, cb);
        tsk->snd_una = cb->ack;
    }
    if (cb->flags & TCP_FIN) {
        tcp_set_state(tsk, TCP_CLOSE_WAIT);
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);
        wake_up(tsk->wait_recv);
    }
    else {
        if (handle_tcp_recv(tsk, cb)) {
            tsk->rcv_nxt = cb->seq_end;
            tcp_send_control_packet(tsk, TCP_ACK);
        }
    }
    break;

case TCP_FIN_WAIT_1:
    if (!is_tcp_seq_valid(tsk, cb))
        return;
    tsk->rcv_nxt = cb->seq_end;

    if (cb->flags & TCP_ACK) {
        tcp_update_window_safe(tsk, cb);
        tsk->snd_una = cb->ack;
```

```
        }
        if ((cb->flags & TCP_FIN) && (cb->flags & TCP_ACK) && tsk->snd_nxt ==
            tsk->snd_una) {
            tcp_set_state(tsk, TCP_TIME_WAIT);
            tcp_set_timewait_timer(tsk);
            tcp_send_control_packet(tsk, TCP_ACK);
        }
        else if ((cb->flags & TCP_ACK) && tsk->snd_nxt == tsk->snd_una)
            tcp_set_state(tsk, TCP_FIN_WAIT_2);
        else if (cb->flags & TCP_FIN) {
            tcp_set_state(tsk, TCP_CLOSING);
            tcp_send_control_packet(tsk, TCP_ACK);
        }
        break;

    case TCP_FIN_WAIT_2:
        if (!is_tcp_seq_valid(tsk, cb))
            return;
        tsk->rcv_nxt = cb->seq_end;

        if (cb->flags & TCP_ACK) {
            tcp_update_window_safe(tsk, cb);
            tsk->snd_una = cb->ack;
        }
        if (cb->flags & TCP_FIN) {
            tcp_set_state(tsk, TCP_TIME_WAIT);
            tcp_set_timewait_timer(tsk);
            tcp_send_control_packet(tsk, TCP_ACK);
        }
        break;

    case TCP_TIME_WAIT:
        log(DEBUG, "received packet in TCP_TIME_WAIT state");
        break;

    case TCP_CLOSE_WAIT:
        log(DEBUG, "received packet in TCP_CLOSE_WAIT state");
        break;

    case TCP_LAST_ACK:
        if (!is_tcp_seq_valid(tsk, cb))
            return;
        tsk->rcv_nxt = cb->seq_end;
```

```
            if (cb->flags & TCP_ACK) {
                tcp_update_window_safe(tsk, cb);
                tsk->snd_una = cb->ack;
            }
            if ((cb->flags & TCP_ACK) && tsk->snd_nxt == tsk->snd_una) {
                tcp_set_state(tsk, TCP_CLOSED);
                tcp_unhash(tsk);
                tcp_bind_unhash(tsk);
            }
            break;

        case(TCP_CLOSED):
            log(DEBUG, "the tcp sock is already closed.");
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
            break;

        default:
            log(ERROR, "unknown tcp state %d", tsk->state);
            break;
    }
}
```

12.2.3 定时器

定时器的实现是通过设定类型为 TIMER_TYPE_TIME_WAIT、超时时间为 TCP_TIME_WAIT_TIMEOUT 的定时器,将相应 TCP 连接加入定时器列表中,增加引用计数:

```
// set the timewait timer of a tcp sock, by adding the ti0.0mer into timer_list
void tcp_set_timewait_timer(struct tcp_sock *tsk)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    tsk->timewait.enable = 1;
    tsk->timewait.type = TIMER_TYPE_TIME_WAIT;
    tsk->timewait.timeout = TCP_TIMEWAIT_TIMEOUT;

    tsk->ref_cnt += 1;
    log(DEBUG, "insert " IP_FMT ":%hu <-> " IP_FMT ":%hu to timewait, ref_cnt += 1",
            HOST_IP_FMT_STR(tsk->sk_sip), tsk->sk_sport,
            HOST_IP_FMT_STR(tsk->sk_dip), tsk->sk_dport);
    pthread_mutex_lock(&timer_list_lock);
    list_add_tail(&tsk->timewait.list, &timer_list);
```

```
    pthread_mutex_unlock(&timer_list_lock);
}
```

通过扫描定时器队列，将 `TIMER_TYPE_TIME_WAIT` 类型的定时器对应的 socket 连接从 `TCP_TIME_WAIT` 状态转换为 `TCP_CLOSED` 状态，释放资源，并从定时器队列中删除；对于 `TIMER_TYPE_RETRANS` 类型的定时器，目前无需处理：

```
// scan the timer_list, find the tcp sock which stays for at 2*MSL, release it
void tcp_scan_timer_list()
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    pthread_mutex_lock(&timer_list_lock);

    struct tcp_timer *timer_p = NULL, *timer_q = NULL;
    list_for_each_entry_safe(timer_p, timer_q, &timer_list, list) {
        if (timer_p->enable) {
            timer_p->timeout -= TCP_TIMER_SCAN_INTERVAL;
            if (timer_p->timeout <= 0) {
                struct tcp_sock *tsk = NULL;
                if (timer_p->type == TIMER_TYPE_TIME_WAIT) {
                    timer_p->enable = 0;
                    tsk = timewait_to_tcp_sock(timer_p);
                    tcp_set_state(tsk, TCP_CLOSED);
                    tcp_unhash(tsk);
                    tcp_bind_unhash(tsk);
                    list_delete_entry(&timer_p->list);
                    free_tcp_sock(tsk);
                }
                else if (timer_p->type == TIMER_TYPE_RETRANS)
                    tsk = retranstimer_to_tcp_sock(timer_p);
            }
        }
    }
    pthread_mutex_unlock(&timer_list_lock);
}
```

12.2.4 TCP 传输应用

对于短消息收发，在服务器端应当监听端口，等待客户端连接，连接之后循环接受数据包，然后将数据包 echo 回客户端，直到客户端断开连接，服务器端也断开连接；对于客户端，连接服务器端，循环发送数据包，，直到完成指定次数的数据包发送或发送错误，断开连接：

```
// tcp server application, listens to port (specified by arg) and serves only one
```

```c
// connection request
void *tcp_server(void *arg)
{
    u16 port = *(u16 *)arg;
    struct tcp_sock *tsk = alloc_tcp_sock();

    struct sock_addr addr;
    addr.ip = htonl(0);
    addr.port = port;
    if (tcp_sock_bind(tsk, &addr) < 0) {
        log(ERROR, "tcp_sock bind to port %hu failed", ntohs(port));
        exit(1);
    }

    if (tcp_sock_listen(tsk, 3) < 0) {
        log(ERROR, "tcp_sock listen failed");
        exit(1);
    }

    log(DEBUG, "listen to port %hu.", ntohs(port));

    struct tcp_sock *csk = tcp_sock_accept(tsk);

    log(DEBUG, "accept a connection.");

    // sleep(5);
    char rbuf[1001], wbuf[1024];
    int rlen = 0;
    while (1) {
        rlen = tcp_sock_read(csk, rbuf, 1000);
        if (rlen == 0) {
            log(DEBUG, "tcp_sock_read return 0, the peer has closed.");
            break;
        }
        else if (rlen > 0) {
            rbuf[rlen] = '\0';
            sprintf(wbuf, "server echoes: %s", rbuf);
            if (tcp_sock_write(csk, wbuf, strlen(wbuf)) < 0) {
                log(ERROR, "tcp_sock_write failed.");
                exit(1);
            }
        }
        else {
```

```
            log(ERROR, "tcp_sock_read return %d, this is an error.", rlen);
            exit(1);
        }
    }

    log(DEBUG, "close this connection.");

    tcp_sock_close(csk);

    return NULL;
}

// tcp client application, connects to server (ip:port specified by arg), each
// time sends one bulk of data and receives one bulk of data
void *tcp_client(void *arg)
{
    struct sock_addr *skaddr = arg;

    struct tcp_sock *tsk = alloc_tcp_sock();

    if (tcp_sock_connect(tsk, skaddr) < 0) {
        log(ERROR, "tcp_sock connect to server ("IP_FMT":%hu)failed.", \
                NET_IP_FMT_STR(skaddr->ip), ntohs(skaddr->port));
        exit(1);
    }

    // sleep(1);
    char *data = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    int len = strlen(data);
    char *wbuf = malloc(len + 1);
    char rbuf[1001];
    int rlen = 0;

    int n = 10;
    for (int i = 0; i < n; i++) {
        memcpy(wbuf, data+i, len-i);
        if (i > 0)
            memcpy(wbuf+len-i, data, i);

        int slen;
        if ((slen = tcp_sock_write(tsk, wbuf, len)) < 0)
            break;
```

```
        rlen = tcp_sock_read(tsk, rbuf, 1000);
        if (rlen == 0) {
            log(DEBUG, "tcp_sock_read return 0, the peer has closed.");
            break;
        }
        else if (rlen > 0) {
            rbuf[rlen] = '\0';
            fprintf(stdout, "%s\n", rbuf);
        }
        else {
            log(ERROR, "tcp_sock_read return %d, this is an error.", rlen);
            exit(1);
        }

        sleep(1);
    }

    tcp_sock_close(tsk);

    free(wbuf);

    return NULL;
}
```

对于大文件传输,服务器端应当监听端口,等待客户端连接,连接之后循环接受文件数据包,然后将数据包写入文件 server-output.dat,直到客户端断开连接,服务器端也断开连接;对于客户端,连接服务器端,循环发送 client-input.dat 文件数据包,直到完成文件数据包发送或发送错误,断开连接:

```
void *tcp_server_file(void *arg)
{
    FILE *fp = fopen("server-output.dat", "wb");
    if (fp == NULL) {
        log(ERROR, "open file server-output.dat failed.");
        exit(1);
    }
    log(DEBUG, "open file server-output.dat success.");

    u16 port = *(u16 *)arg;
    struct sock_addr skaddr;
    struct tcp_sock *tsk = alloc_tcp_sock();
    skaddr.ip = htonl(0);
    skaddr.port = port;
```

```
    if (tcp_sock_bind(tsk, &skaddr) < 0) {
        log(ERROR, "tcp_sock bind to port %hu failed.", ntohs(port));
        exit(1);
    }
    if (tcp_sock_listen(tsk, 3) < 0) {
        log(ERROR, "tcp_sock listen failed.");
        exit(1);
    }
    log(DEBUG, "listening to port %hu.", ntohs(port));

    struct tcp_sock *csk = tcp_sock_accept(tsk);
    log(DEBUG, "accept a connection.");

    char dbuf[10030];
    int dlen = 0;
    while (1) {
        dlen = tcp_sock_read(csk, dbuf, 10024);
        if (dlen > 0)
            fwrite(dbuf, 1, dlen, fp);
        else {
            log(DEBUG, "tcp_sock_read return %d, the peer has closed.", dlen);
            break;
        }
    }

    log(DEBUG, "close this connection.");
    fclose(fp);
    tcp_sock_close(csk);

    return NULL;
}

void *tcp_client_file(void *arg)
{
    FILE *fp = fopen("client-input.dat", "rb");
    if (fp == NULL) {
        log(ERROR, "open file client-input.dat failed.");
        exit(1);
    }

    struct sock_addr *skaddr = arg;
    struct tcp_sock *tsk = alloc_tcp_sock();
    if (tcp_sock_connect(tsk, skaddr) < 0) {
```

```
        log(ERROR, "tcp_sock connect to server ("IP_FMT":%hu) failed.",
            NET_IP_FMT_STR(skaddr->ip), ntohs(skaddr->port));
        exit(1);
    }
    log(DEBUG, "connect to server ("IP_FMT":%hu) success.", NET_IP_FMT_STR(skaddr->ip),
        ntohs(skaddr->port));

    char dbuf[10030];
    int dlen = 0;
    int slen = 0;

    while (1) {
        dlen = fread(dbuf, 1, 10024, fp);
        if (dlen > 0) {
            slen += dlen;
            log(DEBUG, "send %d byte.", slen);
            tcp_sock_write(tsk, dbuf, dlen);
        }
        else {
            log(DEBUG, "file has sent done.");
            break;
        }
        usleep(1000);
    }

    fclose(fp);
    tcp_sock_close(tsk);

    return NULL;
}
```

12.3 实验结果

12.3.1 短消息收发

使用自己的可执行文件和实验提供的标准分别进行测试。

自己的可执行文件分别作为服务器端和客户端:

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening on port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: child 10.0.0.1:4391 join the listen queue of parent 0.0.0.0:4391
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: close this connection.
DEBUG: close tcp sock: 10.0.0.1:10001 -> 10.0.0.2:12345, state: CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: free tcp sock: 10.0.0.1:10001 -> 10.0.0.2:12345.
ERROR: received packet is not SYN but current state is LISTEN, drop it.
```

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
DEBUG: server echoes: server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHI
JKLMNOPQRSTUVWXYZ
DEBUG: server echoes: server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJ
KLMNOPQRSTUVWXYZ0
DEBUG: server echoes: server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJK
LMNOPQRSTUVWXYZ01
DEBUG: server echoes: server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZ012
DEBUG: server echoes: server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM
NOPQRSTUVWXYZ0123
DEBUG: server echoes: server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
OPQRSTUVWXYZ01234
DEBUG: server echoes: server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO
PQRSTUVWXYZ012345
DEBUG: server echoes: server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ0123456
DEBUG: server echoes: server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
RSTUVWXYZ01234567
DEBUG: server echoes: server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZ012345678
DEBUG: close tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001, state: ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: free tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001.
```

自己的可执行文件作为服务器端，实验提供的标准作为客户端：

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening on port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: child 10.0.0.1:4391 join the listen queue of parent 0.0.0.0:4391
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: close this connection.
DEBUG: close tcp sock: 10.0.0.1:10001 -> 10.0.0.2:57976, state: CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: free tcp sock: 10.0.0.1:10001 -> 10.0.0.2:57976.
ERROR: received packet is not SYN but current state is LISTEN, drop it.
```

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# python3 tcp_stack_trans.py client 10.0.0.1 10001
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

实验提供的标准作为服务器端,自己的可执行文件作为客户端：

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# python3 tcp_stack_trans.py server 10001
('10.0.0.2', 12345)
```

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_echo# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
DEBUG: server echoes: server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHI
JKLMNOPQRSTUVWXYZ
DEBUG: server echoes: server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJ
KLMNOPQRSTUVWXYZ0
DEBUG: server echoes: server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJK
LMNOPQRSTUVWXYZ01
DEBUG: server echoes: server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZ012
DEBUG: server echoes: server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM
NOPQRSTUVWXYZ0123
DEBUG: server echoes: server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
OPQRSTUVWXYZ01234
DEBUG: server echoes: server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO
PQRSTUVWXYZ012345
DEBUG: server echoes: server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ0123456
DEBUG: server echoes: server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
RSTUVWXYZ01234567
DEBUG: server echoes: server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZ012345678
DEBUG: close tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001, state: ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: free tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001.
```

可以看出,客户端均正确接收到了服务器端 echo 的数据,且状态转移正确。

12.3.2 大文件传输

使用自己的可执行文件和实验提供的标准分别进行测试。

自己的可执行文件分别作为服务器端和客户端:

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab12
/12-tcp_stack_file# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat success.
DEBUG: listening on port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listening to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: child 10.0.0.1:4391 join the listen queue of parent 0.0.0.0:4391
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
ERROR: no enough space in rcv_buf, drop the packet.
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, the peer has closed.
DEBUG: close this connection.
DEBUG: close tcp sock: 10.0.0.1:10001 -> 10.0.0.2:12345, state: CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: free tcp sock: 10.0.0.1:10001 -> 10.0.0.2:12345.
ERROR: received packet is not SYN but current state is LISTEN, drop it.
```

```
DEBUG: send 3919384 byte.
DEBUG: send 3929408 byte.
DEBUG: send 3939432 byte.
DEBUG: send 3949456 byte.
DEBUG: send 3959480 byte.
DEBUG: send 3969504 byte.
DEBUG: send 3979528 byte.
DEBUG: send 3989552 byte.
DEBUG: send 3999576 byte.
DEBUG: send 4009600 byte.
DEBUG: send 4019624 byte.
DEBUG: send 4029648 byte.
DEBUG: send 4039672 byte.
DEBUG: send 4049696 byte.
DEBUG: send 4052632 byte.
DEBUG: file has sent done.
DEBUG: close tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001, state: ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: free tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001.
```

用 md5sum 检验结果:

```
❯ md5sum client-input.dat server-output.dat
e67c694d237c0daab5dde2aa6e8f42e4  client-input.dat
e67c694d237c0daab5dde2aa6e8f42e4  server-output.dat
```

自己的可执行文件作为服务器端,实验提供的标准作为客户端:

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab
12/12-tcp_stack_file# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat success.
DEBUG: listening on port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listening to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: child 10.0.0.1:4391 join the listen queue of parent 0.0.0.0:4391
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, the peer has closed.
DEBUG: close this connection.
DEBUG: close tcp sock: 10.0.0.1:10001 -> 10.0.0.2:38398, state: CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: free tcp sock: 10.0.0.1:10001 -> 10.0.0.2:38398.
ERROR: received packet is not SYN but current state is LISTEN, drop it.
```

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab
12/12-tcp_stack_file# python3 tcp_stack_trans.py client 10.0.0.1 10001
```

用 md5sum 检验结果:

```
❯ md5sum client-input.dat server-output.dat
f36b069e073a40a41d66f42f6593efbf  client-input.dat
f36b069e073a40a41d66f42f6593efbf  server-output.dat
```

实验提供的标准作为服务器端,自己的可执行文件作为客户端:

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/桌面/2024_zjw_ComputerNetwork/Lab
12/12-tcp_stack_file# python3 tcp_stack_trans.py server 10001
('10.0.0.2', 12345)
```

```
DEBUG: send 3909360 byte.
DEBUG: send 3919384 byte.
DEBUG: send 3929408 byte.
DEBUG: send 3939432 byte.
DEBUG: send 3949456 byte.
DEBUG: send 3959480 byte.
DEBUG: send 3969504 byte.
DEBUG: send 3979528 byte.
DEBUG: send 3989552 byte.
DEBUG: send 3999576 byte.
DEBUG: send 4009600 byte.
DEBUG: send 4019624 byte.
DEBUG: send 4029648 byte.
DEBUG: send 4039672 byte.
DEBUG: send 4049696 byte.
DEBUG: send 4052632 byte.
DEBUG: file has sent done.
DEBUG: close tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001, state: ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: free tcp sock: 10.0.0.2:12345 -> 10.0.0.1:10001.
```

用 `md5sum` 检验结果:

```
❯ md5sum client-input.dat server-output.dat
765a73b0331dce10b33f85ad7fad7e1d  client-input.dat
765a73b0331dce10b33f85ad7fad7e1d  server-output.dat
```

可以看出,因为 `md5sum` 检验结果一致,所以大文件传输也是正确的,且状态转移正确。

12.4 实验总结

本次实验主要是实现 TCP 协议栈的应用层,包括 TCP 连接的建立、数据传输、连接的关闭等。通过本次实验,我对 TCP 协议栈的实现有了更深入的了解,对 TCP 协议的状态机有了更深刻的认识。同时,通过本次实验,我也学会了如何使用多线程来实现 TCP 连接的并发处理,以及如何使用文件 I/O 来实现大文件的传输。