

网络传输机制实验一

学号： 2021K8009929010

姓名： 贾城昊

一、 实验题目：网络传输机制实验一

二、 实验任务：

了解 Socket 数据结构、TCP 连接管理和状态转移以及数据包处理流程，实现 TCP 协议栈的相应服务函数，实现 TCP 建立连接与断开连接的数据包处理机制，实现短消息的收发与大文件传送。

三、 实验流程

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
 - 执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
 - 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
 - 执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
 - 在 h2 上运行 TCP 协议栈的客户端模式 (./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 抓包来验证建立和断开连接的正确性

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
 - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)
 - 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
 - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)
 - 在 h2 上运行 TCP 协议栈的客户端模式(./tcp_stack client 10.0.0.1 10001)
 - 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

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
 - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)
 - 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
 - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)
 - 在 h2 上运行 TCP 协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)

- Client 发送文件 client-input.dat 给 server, server 将收到的数据存储在文件 server-output.dat

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

四、实验过程

(一) socket 连接管理

1. 发起连接

由 tcp_sock_connect 函数负责处理客户端的一方主动发起连接。

由于是客户端主动发起连接, 此时可以完全确定四元组。首先分配源端口, 查找源 IP 地址, 确定四元组。然后发出 SYN 数据包, 请求连接, 并将 TCP 将状态转为 SYN_SENT。接着把 socket 进行 hash 后加入 established hash 表。最后 sleep on, 等待服务器回应 SYN 包。若 SYN 到达, 该线程被唤醒, 说明此时连接已经建立完成。

具体代码如下所示:

```

// 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 * rt = longest_prefix_match(tsk->sk_dip);
    if (!rt) {
        log(ERROR, "cannot find route to dest_ip.");
        return -1;
    }
    tsk->sk_sip = rt->iface->ip;

    err = tcp_sock_set_sport(tsk, 0);
    if (err) {
        log(ERROR, "setting sport when connecting failed.");
        return err;
    }

    tcp_set_state(tsk, TCP_SYN_SENT);
    err = tcp_hash(tsk);
    if (err) {
        log(ERROR, "hashing into hash_table when connecting failed.");
        return err;
    }

    tcp_send_control_packet(tsk, TCP_SYN);

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

    return 0;
}

```

2. 监听端口

tcp_sock_listen 函数负责服务器监听端口。具体来说，设置 backlog，将状态切换到 LISTEN，并将 socket 加入 listen hash 表。

具体实现的代码如下：

```

// set backlog (the maximum number of pending connection request), 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 port %hu.", tsk->sk_sport);

    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    int err = tcp_hash(tsk);
    if (err) {
        log(ERROR, "hashing into hash_table when listening failed.");
        return err;
    }

    return 0;
}

```

3. 等待并接受连接

tcp_sock_accept 函数负责实现服务器的一方等待并接受连接。具体来说，如果没有成功连接的 socket，即 accept_queue 为空，阻塞等待。否则，若 accept_queue 非空，则取出队列中第一个 tcp sock 并 accept。

具体的实现代码如下：

```

// if accept_queue is not empty, 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);
}

```

4. 关闭连接

tcp_sock_close 函数负责完成主动断开和被动断开连接。

因此需要分两种情况处理。如果当前是 ESTABLISHED 或者 SYN_RECV 状态，此时为主动断开连接，向对方发送 FIN 和 ACK 信号，并转到 FIN_WAIT_1 状态；如果当前是 CLOSE_WAIT 状态，此时为被动方断开连接，向对方发送 FIN 和 ACK 信号，并转到 LAST_ACK 状态。最后，如果为其他状态，则直接断开连接，把状态改为 CLOSED，并释放资源即可。(具体状态转移图会在后面给出)

具体的实现代码如下：

```
// 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 sock " IP_FMT " <-> " IP_FMT " 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);
            free_tcp_sock(tsk);
            break;
        }
        case(TCP_ESTABLISHED){
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            free_tcp_sock(tsk);
            break;
        }
        case(TCP_CLOSE_WAIT){
            tcp_set_state(tsk, TCP_LAST_ACK);
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            break;
        }
        default:{
            log(DEBUG, "TCP state default when closing tcp socket");
            tcp_set_state(tsk, TCP_CLOSED);

            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
            free_tcp_sock(tsk);
            break;
        }
    }
}
```

5. 读取缓存区数据

`tcp_sock_read` 函数用于用户进程读取缓存区数据。首先判断缓存区是否有数据可读，如果没有则挂起。这还需要判断 TCP 状态是否为 `CLOSE_WAIT`，这是用于断开连接后让 `read` 函数返回 0。未断开连接则不能在缓存区空时返回 0，因为此时不清楚之后还会有数据到达，返回 0 意味着不再会有数据达到。

读缓存之前先申请读写锁，原因是仅凭 `wait_recv` 是不足以完成同步和互斥的，比如第二次收到数据包时，可能缓存区既有足够区域写，又是非空的。这样协议栈写和用户进程读会同时进行。因此这里需要用读写锁进一步互斥。

注意读缓存读取得字节数不一定是请求的 `len`，如果缓存数据超过 `len`，则读取 `len` 个字节；否则读取当前所有字节。因此需要返回实际读取字节数 `rlen`。

具体的实现代码如下：

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len) {
    while (ring_buffer_empty(tsk->rcv_buf)) {
        if (tsk->state == TCP_CLOSE_WAIT) {
            return 0;
        }
        sleep_on(tsk->wait_recv);
    }

    pthread_mutex_lock(&tsk->rcv_buf_lock);
    int rlen = read_ring_buffer(tsk->rcv_buf, buf, len);
    tsk->rcv_wnd += rlen;
    pthread_mutex_unlock(&tsk->rcv_buf_lock);

    wake_up(tsk->wait_recv);
    return rlen;
}
```

6. 发送数据

`tcp_sock_write` 函数负责用户进程发送数据。由于以太网长度上限是 1514，所以发送时每个数据包最多只能携带给定数量的字节数，如果超过这个上限，则需要分成多个数据包发送。当发送窗口为 0 时需要停止发送，等待对面更新接收窗口时再次唤起。

具体的实现代码如下：

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len) {
    int send_len, packet_len;
    int remain_len = len;
    int handled_len = 0;

    while (remain_len) {
        send_len = min(remain_len, 1514 - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE);
        if (tsk->snd_wnd < send_len) {
            sleep_on(tsk->wait_send);
        }
        packet_len = send_len + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;
        char *packet = (char *)malloc(packet_len);
        memcpy(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE, buf + handled_len, send_len);
        tcp_send_packet(tsk, packet, packet_len);

        tsk->snd_wnd -= send_len;
        remain_len -= send_len;
        handled_len += send_len;
    }

    return handled_len;
}
```

（二）TCP 连接管理和数据包处理

1. 查找监听哈希表

`tcp_sock_lookup_listen` 函数负责查找 listen hash 表。注意只用 sport 作为 key，不使用 saddr，因为此时以 0.0.0.0 代表主机自身。

具体实现代码如下：


```

// 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 value = tcp_hash_function(0, 0, sport, 0);
    struct list_head *list = &tcp_listen_sock_table[value];

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

    return NULL;
}

```

2. 查找建立哈希表

tcp_sock_lookup_established 函数负责查找 established hash 表。与查找 listen hash 表基本相同，此时使用 4 元组作为 key。。

具体实现的代码如下：

```

// 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 value = tcp_hash_function(saddr, daddr, sport, dport);
    struct list_head *list = &tcp_established_sock_table[value];

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

    return NULL;
}

```

3. 处理接收到的数据包

`handle_rcv_data` 函数用于处理接收到的数据包，将数据写入用户缓存。首先判断写入缓存的长度是否合法，如果合法的话，接着判断缓存区是否足够存放这些数据，如果不足，则返回 0。在实际写缓存前，还需要获取读写锁。仅凭 `wait_rcv` 是不足以完成同步和互斥的，比如第二次收到数据包时，可能缓存区既有足够区域写，又是非空的。这样协议栈写和用户进程读会同时进行。因此这里需要用读写锁进一步互斥。

除了读写互斥，还需要实现流量控制。接收窗口等同于缓存区的剩余字节数，当协议栈写入缓存，接收窗口就要相应减少。写完后释放读写锁并唤醒 `wait_rcv`，返回 1。

具体代码如下：

```
// handle the rcv data from TCP packet
int handle_tcp_rcv_data(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(DEBUG, "receive packet is larger than rcv_buf_free, drop it.");

        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_rcv);

    pthread_mutex_unlock(&tsk->rcv_buf_lock);

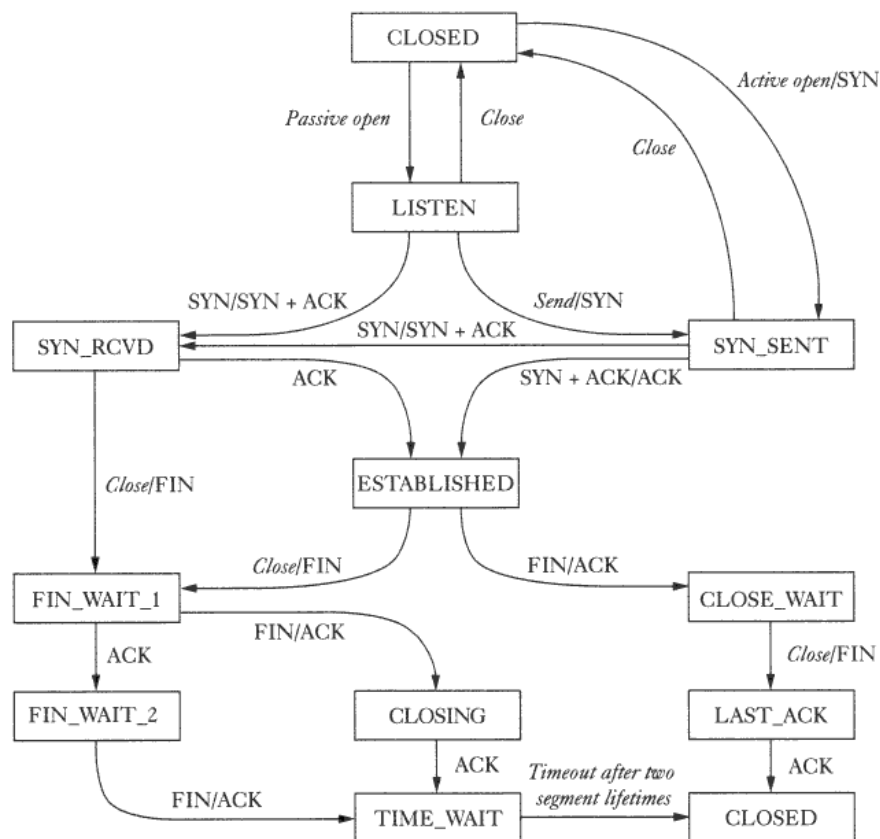
    return 1;
}
```

4. TCP 数据包和连接管理

tcp_process 函数，处理 TCP 数据包和连接管理。检查校验和已经在上一级函数中处理，这里略过。那么接下来首先是检查 socket 连接是否存在。然后处理 RST 包，如果为 RST，直接结束连接，回收资源。

```
if (!tsk) {  
    log(ERROR, "No process listening!!!\n");  
    tcp_send_reset(cb);  
    return;  
}  
if (cb->flags & TCP_RST) {  
    tcp_set_state(tsk, TCP_CLOSED);  
    // release TCP socket  
  
    tcp_unhash(tsk);  
    tcp_bind_unhash(tsk);  
  
    return;  
}
```

接着根据 TCP 状态机的状态进行处理。完整的转移图如下：



如果是 LISTEN 状态收到 SYN，回应建立连接，建立一个 child socket 来负责与该客户进行连接，child socket 的状态设置为 SYN_RECV，最后发送 SYN 和 ACK 包。

```
switch(tsk->state){
    case(TCP_LISTEN):{
        if (cb->flags == TCP_SYN) {
            // alloc child sock
            struct tcp_sock * child_tsk = alloc_tcp_sock();
            child_tsk->parent = tsk;
            tsk->ref_cnt += 1;

            child_tsk->local.ip = cb->daddr;
            child_tsk->local.port = cb->dport;
            child_tsk->peer.ip = cb->saddr;
            child_tsk->peer.port = cb->sport;

            child_tsk->iss = tcp_new_iss();
            child_tsk->snd_nxt = child_tsk->iss;
            child_tsk->rcv_nxt = cb->seq_end;

            tcp_set_state(child_tsk, TCP_SYN_RECV);

            tcp_hash(child_tsk);
            init_list_head(&child_tsk->bind_hash_list);

            log(DEBUG, "Pass " IP_FMT ":%hu <-> " IP_FMT ":%hu from process to listen_queue",
                HOST_IP_FMT_STR(child_tsk->sk_ip), child_tsk->sk_sport,
                HOST_IP_FMT_STR(child_tsk->sk_dip), child_tsk->sk_dport);
            list_add_tail(&child_tsk->list, &tsk->listen_queue);

            // send SYN + ACK
            tcp_send_control_packet(child_tsk, TCP_SYN | TCP_ACK);
        }
        else{
            log(DEBUG, "Current state is TCP_LISTEN but rcv not SYN");
        }
        break;
    }
}
```

如果是 SYN_SENT 状态，收到 SYN 与 ACK 包，说明服务器响应握手，状态转换为 ESTABLISHED，更新 rcv_nxt、snd_una 与发送窗口，发送 ACK 包回应，最后唤醒客户端进程；如果只收到 SYN 而没收到 ACK 包，这表示对端接受了连接请求，但尚未发送 ACK。此时，连接进入 SYN_RECV 状态，然后发送带有 SYN 和 ACK 标志的 TCP 控制包，完成三次握手的最后一步（）。

```

}
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);

        // send ACK;
        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);
        // send SYN + ACK;
        tcp_send_control_packet(tsk, TCP_SYN | TCP_ACK);
    }
    else{
        log(DEBUG, "Current state is TCP_SYN_SENT but rcv not SYN or SYN|ACK");
    }
    break;
}
}
case(TCP_SYN_RECV):{

```

如果是 SYN_RECV 状态下收到 ACK，首先需要检查接收到的序列号，若没问题，则代表 3 次握手完成，更新 rcv_nxt、snd_una 与发送窗口，此时如果 accept queue 未滿，加入 accept queue，并把 TCP 状态机状态改为 ESTABLISHED，最后唤醒服务器进程。如果此时 accept queue 已经满了，则只能关闭连接，将状态改为 CLOSED，并发送 RST 包，释放对应的资源。

```

}
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);
                // send RST
                tcp_send_control_packet(tsk, TCP_RST);

                tcp_unhash(tsk);
                tcp_bind_unhash(tsk);

                // remove from listen list
                list_delete_entry(&tsk->list);
                free_tcp_sock(tsk);
                log(DEBUG, "tcp_sock accept queue is full, so the tsk should be freed.");
            }
            else {
                tcp_set_state(tsk, TCP_ESTABLISHED);
                tcp_sock_accept_enqueue(tsk);

                // wake up user process for accept
                wake_up(tsk->parent->wait_accept);
            }
        }
        else {
            log(ERROR, "tsk->parent is NULL\n");
        }
    }
    else{
        log(DEBUG, "Current state is TCP_SYN_RECV but rcv not ACK");
    }
    break;
}
}

```

如果是 ESTABLISHED 状态，首先检查接收到的序列号，如果不合法，则直接返回。只要 TCP 控制包中包含 ACK 标志，就需要更新窗口信息与 snd_una。而如果此时 TCP 控制包中还包含 FIN 标志，则切换到 CLOSE_WAIT 状态，更新 rcv_nxt，发送 ACK，唤醒对端等待接收的程序。否则，处理接收到的数据（包括对 ring buffer 进行写操作，更新接受窗口，醒对端等待接收的程序等，具体介绍可参见第 3 小点），更新 rcv_nxt 并发送 ACK。

```

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);
        // send ACK;
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);

        wake_up(tsk->wait_rcv);
    } else {
        if (handle_tcp_rcv_data(tsk, cb)) {
            tsk->rcv_nxt = cb->seq_end;
            tcp_send_control_packet(tsk, TCP_ACK);
        }
    }
    break;
}

```

如果是 FIN_WAIT_1 状态，说明本地端已启动连接终止，并正在等待远程端的确认，此时已发送了一个 FIN 数据包。

那么，在该状态下，收到一个 TCP 包后，首先检查接收到的序列，不合法则返回。然后更新 rcv_nxt, snd_una 与窗口信息。接着，如果 TCP 控制包中同时包含 FIN 和 ACK 标志，并且发送序列号 snd_nxt 等于确认序列号 snd_una，则切换到 TCP_TIME_WAIT 状态，设置 TIME_WAIT 定时器，发送 ACK；如果只包含 ACK 标志，并且 snd_nxt 等于 snd_una，则切换到 TCP_FIN_WAIT_2 状态；如果 TCP 控制包中只包含 FIN 标志，则切换到 TCP_CLOSING 状态，发送 ACK。

(定时器作用是等到时间后由另一线程正式断开连接并释放资源。)

```

case(TCP_FIN_WAIT_1):{
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }

    // do something but not this stage
    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);

        // send ACK;
        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);

        // send ACK;
        tcp_send_control_packet(tsk, TCP_ACK);
    }
    break;
}

```

如果是 FIN_WAIT_2 状态，说明本地端已收到对其连接终止请求的确认，但正在等待远程端的终止请求。

如果是 CLOSING 状态，说明本地端已经收到远程端的 FIN 包，同时本地端也向远程端发送了 ACK 包，但仍然在等待对端的 ACK 包

对于这两个状态，不需要做什么特殊的处理。检查接收到的序列号，更新 rcv_nx, snd_una 与滑动窗口。然后如果接收到了对应的包（FIN_WAIT_2 状态接受到了 FIN 包，CLOSING 状态接收到了 ACK 包），则切换到 TCP_TIME_WAIT 状态，设置 TIME_WAIT 定时器。如果是 FIN_WAIT_2 状态，还需要发送 ACK 进行回应。


```

case(TCP_FIN_WAIT_2):{
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }

    // do something but not this stage
    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);

        // send ACK;
        tcp_send_control_packet(tsk, TCP_ACK);
    }
    break;
}
case(TCP_CLOSING):{
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }

    // do something but not this stage
    tsk->rcv_nxt = cb->seq_end;

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

    tcp_set_state(tsk, TCP_TIME_WAIT);
    tcp_set_timewait_timer(tsk);
    break;
}

```

而对于 TIME_WAIT 和 CLOSE_WAIT 在此处均不需要进行处理，前者等待计时器时间到后由另一线程正式断开连接并释放资源，后者由 tcp_sock_close 函数进行处理即可。

对于 LAST_ACK 状态，其代表本地端已启动连接终止，已收到远程端的确认，正在等待最终确认以完成连接关闭。所以检查完接收到的序列号，更新 rcv_nxt, snd_una, 窗口信息后，如果同时包含 ACK 标志，并且 snd_nxt 等于 snd_una，则切换到 CLOSED 状态，释放套接字资源。

对于 CLOSED 状态，表明此时 TCP 连接已关闭，不能发送或接收数据，所以释放资源即可。

```

}
case(TCP_TIME_WAIT):{
    log(DEBUG, "receive a packet of a TCP_TIME_WAIT sock.");
    // do something but not this stage
    break;
}
case(TCP_CLOSE_WAIT):{
    log(DEBUG, "receive a packet of a TCP_CLOSE_WAIT sock.");
    // nothing to do;
    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);

        // release the sock
        tcp_unhash(tsk);
        tcp_bind_unhash(tsk);

        free_tcp_sock(tsk);
    }
    break;
}
case(TCP_CLOSED):{
    log(DEBUG, "this socket is closed");

    // release the sock
    tcp_unhash(tsk);
    tcp_bind_unhash(tsk);
    break;
}
}

```

(三) TCP 定时器管理

1. 定时器的设置

`tcp_set_timewait_timer` 函数用于设置 TCP 连接的 Time-Wait 定时器。主要包括：启用定时器，并设置其类型为 `TIMER_TYPE_TIME_WAIT`。设定定时器的超时时间为 `TCP_TIMEWAIT_TIMEOUT`。将相应的 TCP 连接加入到 Time-Wait 定时器列表中，并增加该连接的引用计数。

具体代码如下所示

```

// 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;

    // refer to this sock in timewait list
    tsk->ref_cnt += 1;
    log(DEBUG, "insert " IP_FMT " <-> " 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);
}

```

2. 定时器队列扫描与释放

tcp_scan_timer_list 用于扫描定时器队列。这一队列中的 TCP 状态描述符在进入 TIME_WAIT 状态时加入队列，达到两倍 MSL 时间后释放资源。对于 TIMER_TYPE_TIME_WAIT 类型的定时器，将相应的 TCP 连接从 TCP_TIME_WAIT 状态迁移到 TCP_CLOSED 状态，释放相关资源，并从定时器列表中移除。而对于 TIMER_TYPE_RETRANS 类型的定时器，在当前版本中没有特定的操作。

具体代码如下所示：

```

// scan the timer_list, find the tcp sock which stays for at 2 ms, 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) {
                    // do TCP_TIME_WAIT to TCP_CLOSED
                    timer_p->enable = 0;

                    tsk = timewait_to_tcp_sock(timer_p);
                    //assert(tsk->state == TCP_TIME_WAIT);
                    tcp_set_state(tsk, TCP_CLOSED);

                    tcp_unhash(tsk);
                    tcp_bind_unhash(tsk);

                    // remove reference from timewait list
                    list_delete_entry(&timer_p->list);
                    free_tcp_sock(tsk);

                    // just leave the closed sock in accept_queue/user
                } else if (timer_p->type == TIMER_TYPE_RETRANS) {
                    tsk = retrans_timer_to_tcp_sock(timer_p);
                    // nothing to do in this version
                }
            }
        }
    }

    pthread_mutex_unlock(&timer_list_lock);
}

```

（四）TCP 数据传输应用

1. 短消息传输（服务器端）

tcp_server 函数作为 TCP 服务器应用程序，用于实现短消息接收。监听指定端口，接受连接请求，在连接建立后，它会循环接收客户端发送的文本数据。每次接收到数据后，服务器将数据进行简单处理，添加回显信息后返回给客户端，直到客户端关闭连接。完成上述操作后，服务器关闭连接。。

具体代码如下所示

```

// 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.");

    char rbuf[1001];
    char wbuf[1024];
    int rlen = 0;
    while (1) {
        rlen = tcp_sock_read(csk, rbuf, 1000);
        if (rlen == 0) {
            log(DEBUG, "tcp_sock_read return 0, finish transmission.");
            break;
        }
        else if (rlen > 0) {
            rbuf[rlen] = '\0';
            sprintf(wbuf, "server echoes: %s", rbuf);
            if (tcp_sock_write(csk, wbuf, strlen(wbuf)) < 0) {
                log(DEBUG, "tcp_sock_write return a negative value, something goes wrong.");
                exit(1);
            }
        }
        else {
            log(DEBUG, "tcp_sock_read return a negative value, something goes wrong.");
            exit(1);
        }
    }

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

    tcp_sock_close(csk);

    return NULL;
}

```

2. 短消息传输（客户端）

tcp_client 函数作为 TCP 客户端应用程序，连接到指定的服务器。在连接建立后，客户端会循环发送一批文本数据。每次发送完数据后，等待服务器回显后将其输出到标准输出。客户端会持续发送和接收数据，直至完成指定次数的数据传输或发生错误，然后关闭连接。

具体代码如下所示：

```

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);
    }

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

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

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

        rlen = tcp_sock_read(tsk, rbuf, 1000);
        if (rlen == 0) {
            log(DEBUG, "tcp_sock_read return 0, finish transmission.");
            break;
        }
        else if (rlen > 0) {
            rbuf[rlen] = '\0';
            fprintf(stdout, "%s\n", rbuf);
        }
        else {
            log(DEBUG, "tcp_sock_read return a negative value, something goes wrong.");
            exit(1);
        }
        sleep(1);
    }

    tcp_sock_close(tsk);

    free(wbuf);

    return NULL;
}

```

3. 文件传输（服务器端）

tcp_server_file 函数作为 TCP 文件传输服务器应用程序，监听指定端口并接受一个连接请求。在连接建立后，服务器会循环接收客户端发送的文件数据，并将数据写 server-output.dat 文件。整个过程中，服务器持续接收和写入数据，直至客户端关闭连接。连接完成后，服务器关闭连接。。

具体的实现代码如下：

```

void *tcp_server_file(void *arg)
{
    FILE * f = fopen("server-output.dat", "wb");
    if (!f) {
        log(ERROR, "open file server-output.dat failed");
    }
    log(DEBUG, "open file server-output.dat");

    //struct timeval tv_start, tv_end;
    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.");
    //gettimeofday(&tv_start, NULL);

    char data_buf[10030];
    int data_len = 0;
    while (1) {
        data_len = tcp_sock_read(csk, data_buf, 10024);
        if (data_len > 0) {
            fwrite(data_buf, 1, data_len, f);
        } else {
            log(DEBUG, "peer closed.");
            break;
        }
    }

    //gettimeofday(&tv_end, NULL);
    //long time_res = get_interval(tv_start, tv_end);
    //time_res /= 1000000;
    //fprintf(stderr, "used time: %ld s\n", time_res);

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

    tcp_sock_close(csk);

    return NULL;
}

```

4. 文件传输（客户端）

tcp_client_file 函数作为 TCP 文件传输客户端应用程序，连接到指定的服务器后，客户端会循环读取 client-input.dat 文件的数据，并将数据发送到服务器。整个过程中，客户端持续读取文件并发送数据，直至文件中的数据全部发送完毕或发生错误，然后关闭连接。

具体的实现代码如下：

```

// 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_file(void *arg)
{
    FILE * f = fopen("client-input.dat", "rb");
    if (!f) {
        log(ERROR, "open file client-input.dat failed");
    }
    log(DEBUG, "open file client-input.dat");

    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 success.");

    char data_buf[10030];
    int data_len = 0;

    int send_size = 0;
    while (1) {
        data_len = fread(data_buf, 1, 10024, f);
        if (data_len > 0) {
            send_size += data_len;
            log(DEBUG, "sent %d Bytes", send_size);
            tcp_sock_write(tsk, data_buf, data_len);
        } else {
            log(DEBUG, "the file has been sent completely.");
            break;
        }
        usleep(10000);
    }

    fclose(f);
    tcp_sock_close(tsk);

    return NULL;
}

```

五、实验结果与分析

(一) 连接管理

1. 本实验 Server 与本实验 Client

H1: 本实验 server

H2: 本实验 client

运行结果如下图所示:


```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:12345 from process to listen_queue
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: close 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: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:12345.
```

```
"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H1、H2 的状态转移行为符合预期结果。

2. 标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# python2 tcp_stack_conn.py
server 10001
root@Computer:~/workspace/Network/lab12/12-tcp_stack#
```

```
"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H2 的状态转移行为符合预期结果。

3. 本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client

运行结果如下图所示:

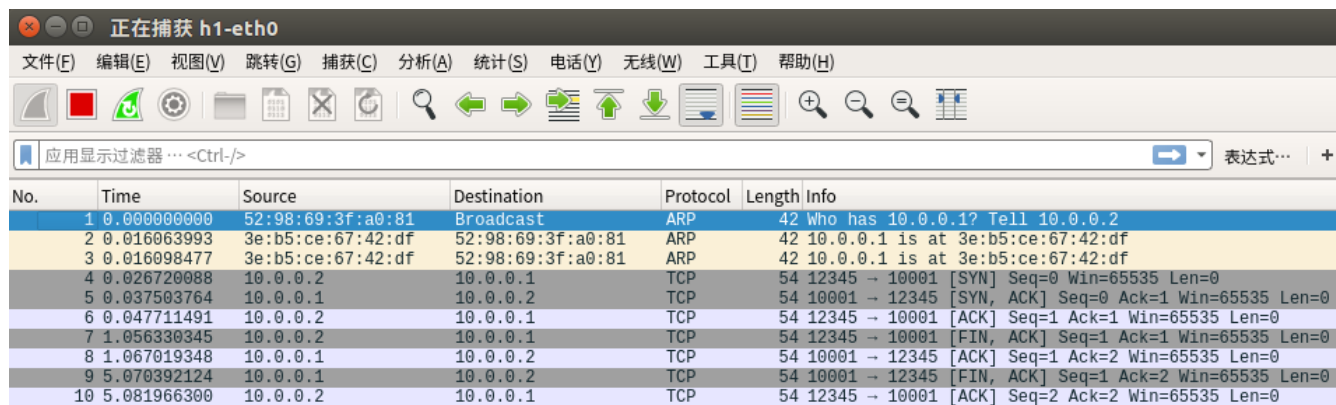
```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:60672 from process to listen_queue
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: close sock 10.0.0.1:10001 <-> 10.0.0.2:60672, state CLOSE_WAIT
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:60672.

"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# python2 tcp_stack_conn.py
client 10.0.0.1 10001
root@Computer:~/workspace/Network/lab12/12-tcp_stack#
```

H1 的状态转移行为符合预期结果。

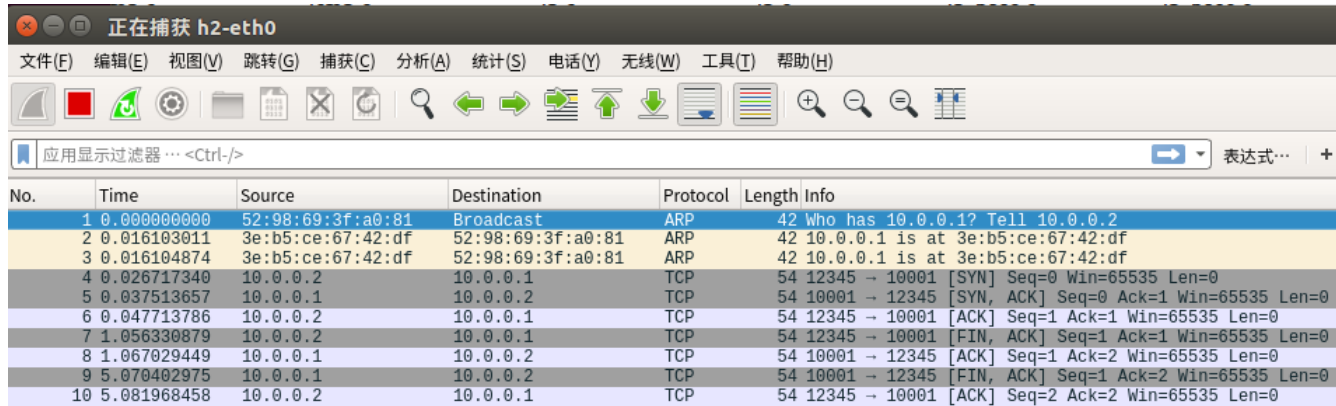
4. wireshark 抓包验证

本实验 server 与本实验 client 的抓包结果如下：



正在捕获 h1-eth0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	52:98:69:3f:a0:81	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.016063993	3e:b5:ce:67:42:df	52:98:69:3f:a0:81	ARP	42	10.0.0.1 is at 3e:b5:ce:67:42:df
3	0.016098477	3e:b5:ce:67:42:df	52:98:69:3f:a0:81	ARP	42	10.0.0.1 is at 3e:b5:ce:67:42:df
4	0.026720088	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.037503764	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.047711491	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.056330345	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.067019348	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.070392124	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.081966300	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0



正在捕获 h2-eth0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	52:98:69:3f:a0:81	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.016103011	3e:b5:ce:67:42:df	52:98:69:3f:a0:81	ARP	42	10.0.0.1 is at 3e:b5:ce:67:42:df
3	0.016104874	3e:b5:ce:67:42:df	52:98:69:3f:a0:81	ARP	42	10.0.0.1 is at 3e:b5:ce:67:42:df
4	0.026717340	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.037513657	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.047713786	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.056330879	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.067029449	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.070402975	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.081968458	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

可以看到，收发包与预期一致，本实验实现的功能正确。

(二) 短消息收发

1. 本实验 Server 与本实验 Client

H1：本实验 server

H2：本实验 client

运行结果如下图所示：

```

"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:12345 from process to listen_queue
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, finish transmission.
DEBUG: close this connection.
DEBUG: close 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: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:12345.

"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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.
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
DEBUG: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H1、H2 的状态转移行为符合预期结果，且 H1、H2 能正确收发数据。

2. 标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# python2 tcp_stack_trans.p
y server 10001
('10.0.0.2', 12345)
root@Computer:~/workspace/Network/lab12/12-tcp_stack#

"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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.
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
DEBUG: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H2 的状态转移行为符合预期结果，且 H1、H2 能正确收发数据。

3. 本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client

运行结果如下图所示：

```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:60700 from process to listen_queue
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, finish transmission.
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:60700, state CLOSE_WAIT
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:60700.
```

```
"Node: h2"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# python2 tcp_stack_trans.p
y client 10.0.0.1 10001
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
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
root@Computer:~/workspace/Network/lab12/12-tcp_stack#
```

H1 的状态转移行为符合预期结果，且 H1、H2 能正确收发数据。

综上，本实验 server 和 client 能正确收发数据，功能正确。

(三) 文件传输

1. 编写 Python 脚本

参考 tcp_stack_trans.py 编写 python 文件 tcp_stack_file.py, 使之能实现文件的传输，具体的代码如下：

```

def server(port, filename):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    s.bind(('0.0.0.0', int(port)))
    s.listen(3)

    cs, addr = s.accept()
    print addr

    with open(filename, 'wb') as f:
        while True:
            data = cs.recv(1024)
            if data:
                f.write(data)
            else:
                break

    s.close()

def client(ip, port, filename):
    s = socket.socket()
    s.connect((ip, int(port)))

    f = open('client-input.dat', 'r')
    file_str = f.read()
    length = len(file_str)
    i = 0

    while length > 0:
        send_len = min(length, 100000)
        s.send(file_str[i: i+send_len])
        sleep(0.1)
        print("send:{0}, remain:{1}, total: {2}/{3}".format(i, length, i, i+length))
        length -= send_len
        i += send_len

    f.close()
    s.close()

if __name__ == '__main__':
    if sys.argv[1] == 'server':
        server(sys.argv[2], "server-output.dat")
    elif sys.argv[1] == 'client':
        client(sys.argv[2], sys.argv[3], "client-input.dat")

```

2. 本实验 Server 与本实验 Client

H1: 本实验 server

H2: 本实验 client

运行结果如下图所示:


```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:12345 from process to listen_queue
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: peer closed.
used time: 4 s
DEBUG: close this connection.
DEBUG: close 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: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:12345.

"Node: h2"
DEBUG: sent 3919384 Bytes
DEBUG: sent 3929408 Bytes
DEBUG: sent 3939432 Bytes
DEBUG: sent 3949456 Bytes
DEBUG: sent 3959480 Bytes
DEBUG: sent 3969504 Bytes
DEBUG: sent 3979528 Bytes
DEBUG: sent 3989552 Bytes
DEBUG: sent 3999576 Bytes
DEBUG: sent 4009600 Bytes
DEBUG: sent 4019624 Bytes
DEBUG: sent 4029648 Bytes
DEBUG: sent 4039672 Bytes
DEBUG: sent 4049696 Bytes
DEBUG: sent 4052632 Bytes
DEBUG: the file has been sent completely.
DEBUG: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H1、H2 的状态转移行为符合预期结果。

MD5 验证结果如下：

```
sai@Computer: ~/workspace/Network/lab12/12-tcp_stack
sai@Computer:~$ cd workspace/Network/lab12/12-tcp_stack/
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum client-input.dat > c
checksum1.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum server-output.dat >
checksum2.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ diff checksum1.txt checksum
2.txt
1c1
< 05f7c5290651bce2fa0369f1921b3ccb  client-input.dat
---
> 05f7c5290651bce2fa0369f1921b3ccb  server-output.dat
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$
```

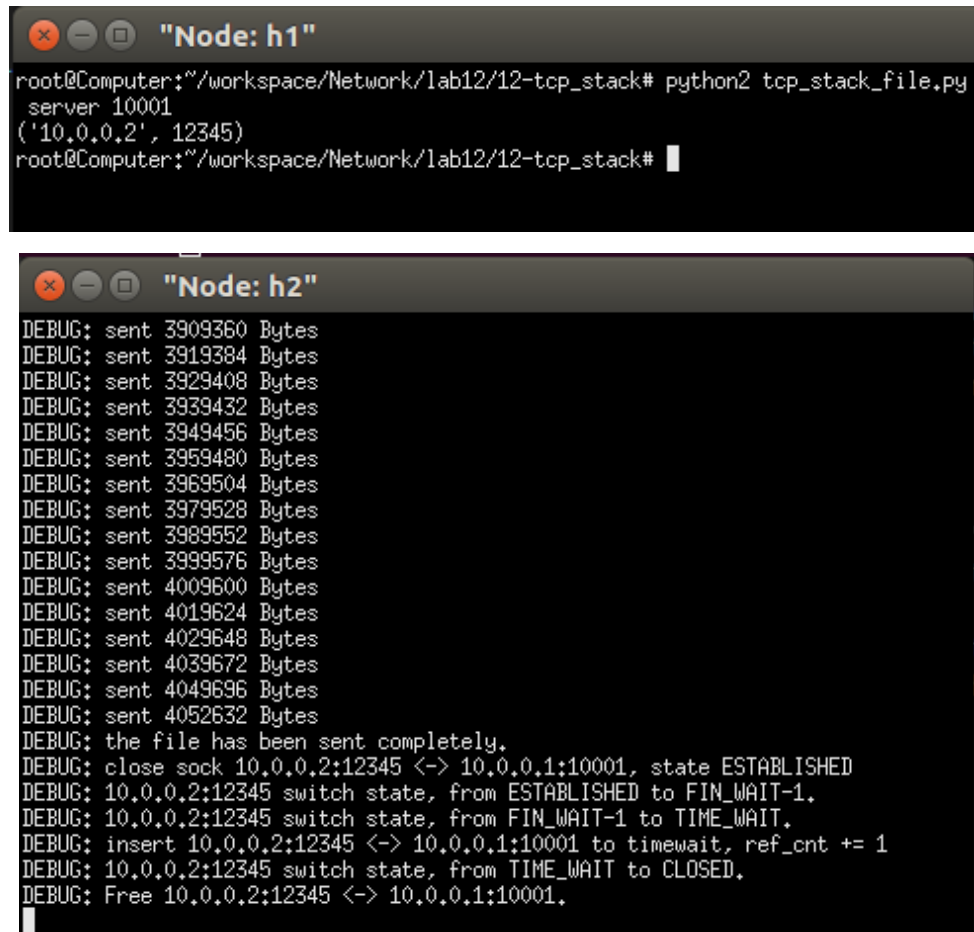
可以看出，H1、H2 能正确传输文件。

3. 标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client

运行结果如下图所示:

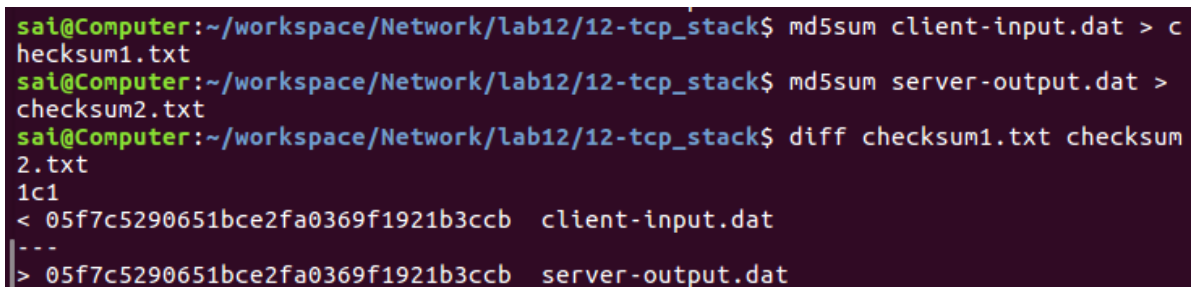


```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# python2 tcp_stack_file.py
server 10001
('10.0.0.2', 12345)
root@Computer:~/workspace/Network/lab12/12-tcp_stack#

"Node: h2"
DEBUG: sent 3909360 Bytes
DEBUG: sent 3919384 Bytes
DEBUG: sent 3929408 Bytes
DEBUG: sent 3939432 Bytes
DEBUG: sent 3949456 Bytes
DEBUG: sent 3959480 Bytes
DEBUG: sent 3969504 Bytes
DEBUG: sent 3979528 Bytes
DEBUG: sent 3989552 Bytes
DEBUG: sent 3999576 Bytes
DEBUG: sent 4009600 Bytes
DEBUG: sent 4019624 Bytes
DEBUG: sent 4029648 Bytes
DEBUG: sent 4039672 Bytes
DEBUG: sent 4049696 Bytes
DEBUG: sent 4052632 Bytes
DEBUG: the file has been sent completely.
DEBUG: close 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 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

H2 的状态转移行为符合预期结。

MD5 验证结果如下:



```
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum client-input.dat > checksum1.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum server-output.dat > checksum2.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ diff checksum1.txt checksum2.txt
1c1
< 05f7c5290651bce2fa0369f1921b3ccb  client-input.dat
---
> 05f7c5290651bce2fa0369f1921b3ccb  server-output.dat
```

可以看出, H1、H2 能正确传输文件

4. 本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab12/12-tcp_stack# ./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
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:60886 from process to listen_queue
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: peer closed.
used time: 4 s
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:60886, state CLOSE_WAIT
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:60886.
```

```
"Node: h2"
send:1800000, remain:2252632, total: 1800000/4052632
send:1900000, remain:2152632, total: 1900000/4052632
send:2000000, remain:2052632, total: 2000000/4052632
send:2100000, remain:1952632, total: 2100000/4052632
send:2200000, remain:1852632, total: 2200000/4052632
send:2300000, remain:1752632, total: 2300000/4052632
send:2400000, remain:1652632, total: 2400000/4052632
send:2500000, remain:1552632, total: 2500000/4052632
send:2600000, remain:1452632, total: 2600000/4052632
send:2700000, remain:1352632, total: 2700000/4052632
send:2800000, remain:1252632, total: 2800000/4052632
send:2900000, remain:1152632, total: 2900000/4052632
send:3000000, remain:1052632, total: 3000000/4052632
send:3100000, remain:952632, total: 3100000/4052632
send:3200000, remain:852632, total: 3200000/4052632
send:3300000, remain:752632, total: 3300000/4052632
send:3400000, remain:652632, total: 3400000/4052632
send:3500000, remain:552632, total: 3500000/4052632
send:3600000, remain:452632, total: 3600000/4052632
send:3700000, remain:352632, total: 3700000/4052632
send:3800000, remain:252632, total: 3800000/4052632
send:3900000, remain:152632, total: 3900000/4052632
send:4000000, remain:52632, total: 4000000/4052632
root@Computer:~/workspace/Network/lab12/12-tcp_stack#
```

H1 的状态转移行为符合预期结果。

MD5 验证结果如下:

```

sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum client-input.dat > checksum1.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum server-output.dat > checksum2.txt
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ diff checksum1.txt checksum2.txt
1c1
< 05f7c5290651bce2fa0369f1921b3ccb  client-input.dat
---
> 05f7c5290651bce2fa0369f1921b3ccb  server-output.dat
sai@Computer:~/workspace/Network/lab12/12-tcp_stack$ md5sum client-input.dat > checksum1.txt

```

可以看出，H1、H2 能正确传输文件

综上，本实验 server 和 client 能正确收发文件，功能正确。

六、实验总结

在本次实验中，通过实验内容一我了解了 TCP 协议栈的功能、socket 数据结构的设计，以及 TCP 建立和断开连接的处理流程。另外通过实验内容二和实验内容三，我了解了 TCP 协议栈的传输数据功能、socket 发送与读取数据的处理流程，同时也了解到简单的流量控制方式，这让我对 TCP 协议有了更多的认识。综上，本次实验让我对 TCP 协议有了更深的了解，为之后的实验奠定了基础。