一、实验内容

1、 实验内容一

(1) 将实验 2 中的相应文件（arp.c，arpcache.c，device_internal.c，icmp.c，ip_base.c，rtable.c，rtable_internal.c）复制过来，编译生成 tcp_stack

(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，连接 h1 并正确收发数据（./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.py 替换其中任意一端，对端都能正确收发数据

2、 实验内容二

(1) 修改 tcp_apps.c（以及 tcp_stack.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 将收到的数据存储到文件 server-output.dat

(6) 使用 md5sum 比较两个文件是否完全相同

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

二、实验流程

1、 实验内容一

(1) 修改 arpcache.h

```
void *arpcache_sweep(void *);
```

(2) 修改 tcp.c

```
void handle_tcp_packet(char *packet, struct iphdr *ip, struct tcphdr *tcp)
{
    /*if (tcp_checksum(ip, tcp) != tcp->checksum) {
        log(ERROR, "received tcp packet with invalid checksum, drop it.");
        return;
    }*/

    struct tcp_cb cb;
    tcp_cb_init(ip, tcp, &cb);

    struct tcp_sock *tsk = tcp_sock_lookup(&cb);

    if (tsk) {
        tcp_process(tsk, &cb, packet);
    }
    else {
        log(ERROR, "do not find socket, ip: "IP_FMT" port: %d\n",
HOST_IP_FMT_STR(cb.daddr), cb.dport);
    }
}
```

(3) 修改 tcp_in.c

```
struct tcp_sock *alloc_child_tcp_sock(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    struct tcp_sock *child = alloc_tcp_sock();
    memcpy((char *)child, (char *)tsk, sizeof(struct tcp_sock));
    child->parent = tsk;
    child->sk_sip = cb->daddr;
    child->sk_sport = cb->dport;
    child->sk_dip = cb->saddr;
    child->sk_dport = cb->sport;
    child->iss = tcp_new_iss();
    child->snd_nxt = child->iss;
    child->rcv_nxt = cb->seq + 1;
    tcp_sock_listen_enqueue(child);
    tcp_set_state(child, TCP_SYN_RECV);
    tcp_hash(child);
    return child;
}

void handle_recv_data(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    while (ring_buffer_free(tsk->rcv_buf) < cb->pl_len) {
        sleep_on(tsk->wait_recv);
    }

    pthread_mutex_lock(&tsk->rcv_buf->rw_lock);
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    tsk->rcv_wnd -= cb->pl_len;
    pthread_mutex_unlock(&tsk->rcv_buf->rw_lock);
    wake_up(tsk->wait_recv);

    tsk->rcv_nxt = cb->seq + cb->pl_len;
    tsk->snd_una = cb->ack;
    tcp_send_control_packet(tsk, TCP_ACK);
}

void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    struct tcphdr *tcp = packet_to_tcp_hdr(packet);
    if (tcp->flags & TCP_RST) {
        tcp_sock_close(tsk);
        return;
    }

    switch (tsk->state) {
        case TCP_LISTEN: {
            if (tcp->flags & TCP_SYN) {
                //tcp_set_state(tsk, TCP_SYN_RECV);
                struct tcp_sock *child = alloc_child_tcp_sock(tsk, cb);
                tcp_send_control_packet(child, TCP_SYN|TCP_ACK);
            }
            return;
        }
        case TCP_SYN_SENT: {
            if (tcp->flags & (TCP_ACK | TCP_SYN)) {
                tcp_set_state(tsk, TCP_ESTABLISHED);
                tsk->rcv_nxt = cb->seq + 1;
            tsk->snd_una = cb->ack;
                wake_up(tsk->wait_connect);
                tcp_send_control_packet(tsk, TCP_ACK);
            }
            return;
        }
        case TCP_SYN_RECV: {
            if (tcp->flags & TCP_ACK) {
                if (tcp_sock_accept_queue_full(tsk->parent)) {
                    return;
                }
                struct tcp_sock *child = tcp_sock_listen_dequeue(tsk->parent);
                if (child != tsk) {
                    log(ERROR, "child != tsk\n");
                }
                tcp_sock_accept_enqueue(tsk);
                tcp_set_state(tsk, TCP_ESTABLISHED);
                tsk->rcv_nxt = cb->seq;
              tsk->snd_una = cb->ack;
                wake_up(tsk->parent->wait_accept);
            }
            return;
        }
        default: {
            break;
        }
    }

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

    switch (tsk->state) {
        case TCP_ESTABLISHED: {
            if (tcp->flags & TCP_FIN) {
                tcp_set_state(tsk, TCP_CLOSE_WAIT);
                tsk->rcv_nxt = cb->seq + 1;
                tsk->snd_una = cb->ack;
                tcp_send_control_packet(tsk, TCP_ACK);
                wake_up(tsk->wait_recv);
            }
            else if (tcp->flags & TCP_ACK) {
                if (cb->pl_len == 0) {
```

```
            tsk->rcv_nxt = cb->seq;
            tsk->snd_una = cb->ack;
            tcp_update_window_safe(tsk, cb);
        }
        else {
            handle_recv_data(tsk, cb);
        }
    }
    break;
}
case TCP_LAST_ACK: {
    if (tcp->flags & TCP_ACK) {
        tcp_set_state(tsk, TCP_CLOSED);
        tsk->rcv_nxt = cb->seq;
        tsk->snd_una = cb->ack;
        tcp_unhash(tsk);
        //tcp_bind_unhash(tsk);
    }
    break;
}
case TCP_FIN_WAIT_1: {
    if (tcp->flags & TCP_ACK) {
        tcp_set_state(tsk, TCP_FIN_WAIT_2);
        tsk->rcv_nxt = cb->seq;
        tsk->snd_una = cb->ack;
    }
    break;
}
case TCP_FIN_WAIT_2: {
    if (tcp->flags & TCP_FIN) {
        tcp_set_state(tsk, TCP_TIME_WAIT);
        tsk->rcv_nxt = cb->seq + 1;
        tsk->snd_una = cb->ack;
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_timewait_timer(tsk);
    }
    break;
}
default: {
```

```
            break;
        }
    }
}
```

(4) 修改 tcp_sock.c

```
void init_tcp_stack()
{
    for (int i = 0; i < TCP_HASH_SIZE; i++)
        init_list_head(&tcp_established_sock_table[i]);

    for (int i = 0; i < TCP_HASH_SIZE; i++)
        init_list_head(&tcp_listen_sock_table[i]);

    for (int i = 0; i < TCP_HASH_SIZE; i++)
        init_list_head(&tcp_bind_sock_table[i]);

    init_list_head(&timer_list);

    pthread_t timer;
    pthread_create(&timer, NULL, tcp_timer_thread, NULL);
}
void free_tcp_sock(struct tcp_sock *tsk)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    tsk->ref_cnt--;
    if (tsk->ref_cnt == 0) {
        free(tsk);
    }
}

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

    return NULL;
}

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

    return NULL;
}

int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    u16 sport = tcp_get_port();
    if (sport == 0) {
        return -1;
    }
    rt_entry_t *entry = longest_prefix_match(ntohl(skaddr->ip));
    if (entry == NULL) {
        return -1;
    }
    tsk->sk_sip = entry->iface->ip;
    tsk->sk_sport = sport;
    tsk->sk_dip = ntohl(skaddr->ip);
    tsk->sk_dport = ntohs(skaddr->port);
    tcp_bind_hash(tsk);
    tcp_send_control_packet(tsk, TCP_SYN);
    tcp_set_state(tsk, TCP_SYN_SENT);
    tcp_hash(tsk);
    sleep_on(tsk->wait_connect);
    return sport;
    //return -1;
}

int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    return tcp_hash(tsk);
    //return -1;
}

// push the tcp sock into listen_queue
inline void tcp_sock_listen_enqueue(struct tcp_sock *tsk)
{
    if (!list_empty(&tsk->list))
        list_delete_entry(&tsk->list);
    list_add_tail(&tsk->list, &tsk->parent->listen_queue);
}

// pop the first tcp sock of the listen_queue
inline struct tcp_sock *tcp_sock_listen_dequeue(struct tcp_sock *tsk)
{
    struct tcp_sock *new_tsk = list_entry(tsk->listen_queue.next,
struct tcp_sock, list);
    list_delete_entry(&new_tsk->list);
    init_list_head(&new_tsk->list);

    return new_tsk;
}

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);
    }

    struct tcp_sock *child;
    if ((child = tcp_sock_accept_dequeue(tsk)) != NULL) {
        return child;
    }

    return NULL;
}

void tcp_sock_close(struct tcp_sock *tsk)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    switch (tsk->state) {
        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;
        }
    }
}

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

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

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

    while (remain_len) {
        send_len = min(remain_len, 1514 - ETHER_HDR_SIZE -
IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE);
        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 + already_len, send_len);
        tcp_send_packet(tsk, packet, packet_len);

        if (tsk->snd_wnd == 0) {
            sleep_on(tsk->wait_send);
        }
        remain_len -= send_len;
        already_len += send_len;
    }

    return len;
```

```
}
```

(5) 修改 tcp_timer.c

```
struct list_head timer_list;
//static struct list_head timer_list;

void tcp_scan_timer_list()
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    struct tcp_timer *time_entry = NULL, *time_q = NULL;
    list_for_each_entry_safe(time_entry, time_q, &timer_list, list) {
        if (time_entry->enable == 1 && time_entry->type == 0 && ((time(NULL)
- time_entry->timeout) > TCP_TIMEWAIT_TIMEOUT / 1000000)) {
            struct tcp_sock *tsk = timewait_to_tcp_sock(time_entry);
            list_delete_entry(&time_entry->list);
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
        }
    }
}

void tcp_set_timewait_timer(struct tcp_sock *tsk)
{
    //fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    tsk->timewait.type = 0;
    tsk->timewait.enable = 1;
    tsk->timewait.timeout = time(NULL);
    list_add_tail(&tsk->timewait.list, &timer_list);
}
```

(6) 修改 main.c

```
void handle_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_header *eh = (struct ether_header *)packet;

    // log(DEBUG, "got packet from %s, %d bytes, proto: 0x%04hx\n",
    //     iface->name, len, ntohs(eh->ether_type));
    switch (ntohs(eh->ether_type)) {
        case ETH_P_IP:
            handle_ip_packet(iface, packet, len);
            break;
        case ETH_P_ARP:
            handle_arp_packet(iface, packet, len);
            break;
        default:
            log(ERROR, "Unknown packet type 0x%04hx, ingore it.", \
                    ntohs(eh->ether_type));
            free(packet);
            break;
    }
}
```

(7) 修改 ring_buffer.h

```
#include <pthread.h>

struct ring_buffer {
    int size;
    int head;     // read from head
    int tail;     // write from tail
    pthread_mutex_t rw_lock;
    char buf[0];
};
```

(8) 修改 tcp_sock.h

```
void tcp_sock_listen_enqueue(struct tcp_sock *tsk);
struct tcp_sock *tcp_sock_listen_dequeue(struct tcp_sock *tsk);
```

(9) 修改 tcp_timer.h

```
extern struct list_head timer_list;
```

(10) 将实验 2 中的相应文件（arp.c, arpcache.c, device_internal.c, icmp.c, ip_base.c, rtable.c, rtable_internal.c）复制过来

(11) 修改 arp.c

```
//log(DEBUG, "handle arp send request packet\n");
//log(DEBUG, "handle arp packet\n");
```

(12) 修改 ip_base.c

```
// determine the next hop for the destination IP address
u32 get_next_hop(rt_entry_t *entry, u32 dst)
{
    if (entry->gw)
        return entry->gw;
    else
        return dst;
}

// void ip_send_packet(char *packet, int len)
// send IP packet for icmp
void icmp_ip_send_packet(char *packet, int len)
{
    struct iphdr *iphdr = packet_to_ip_hdr(packet);
    u32 dest_ip = ntohl(iphdr->daddr);
    rt_entry_t *rt_dest = longest_prefix_match(dest_ip);

    iface_send_packet(rt_dest->iface, packet, len);
}

void ip_send_packet(char *packet, int len)
{
    struct iphdr *ip = packet_to_ip_hdr(packet);
    u32 dst = ntohl(ip->daddr);
    rt_entry_t *entry = longest_prefix_match(dst);
    if (!entry) {
        log(ERROR, "Could not find forwarding rule for IP (dst:"IP_FMT")
packet.", HOST_IP_FMT_STR(dst));
        free(packet);
        return;
    }

    u32 next_hop = get_next_hop(entry, dst);
    iface_info_t *iface = entry->iface;

    iface_send_packet_by_arp(iface, next_hop, packet, len);
}
```

(13) 编译生成 tcp_stack

```
wasder@WASDER:~/exp3/5-tcp_stack-1$ make
```

(14) 修改给定网络拓扑（tcp_topo.py）第 19、30 行

```
print('%s should be set executable by using `chmod +x $script_name`' % (fname))
print('`%s` is required but missing, which could be installed via `apt` or
`aptitude`' % (program))
```

(15) 运行给定网络拓扑（tcp_topo.py）

```
wasder@WASDER:~/exp3/5-tcp_stack-1$ sudo python3 tcp_topo.py
```

(16) 在节点 h1 上执行 TCP 程序：执行脚本（disable_offloading.sh, disable_tcp_rst.sh）

```
mininet> xterm h1
h1# cd scripts/
h1# ./disable_offloading.sh
h1# ./disable_tcp_rst.sh
```

(17) 在节点 h1 上执行 TCP 程序：在 h1 上运行 TCP 协议栈的服务器模式（./tcp_stack server 10001）

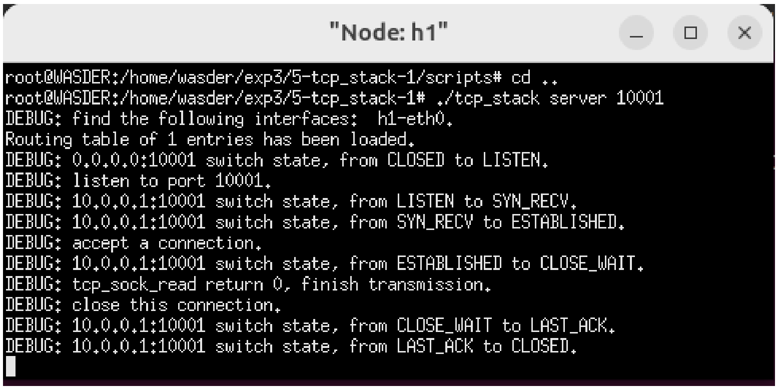
```
h1# cd..
h1# ./tcp_stack server 10001
```

(6) 在节点 h2 上执行 TCP 程序：执行脚本（disable_offloading.sh, disable_tcp_rst.sh）

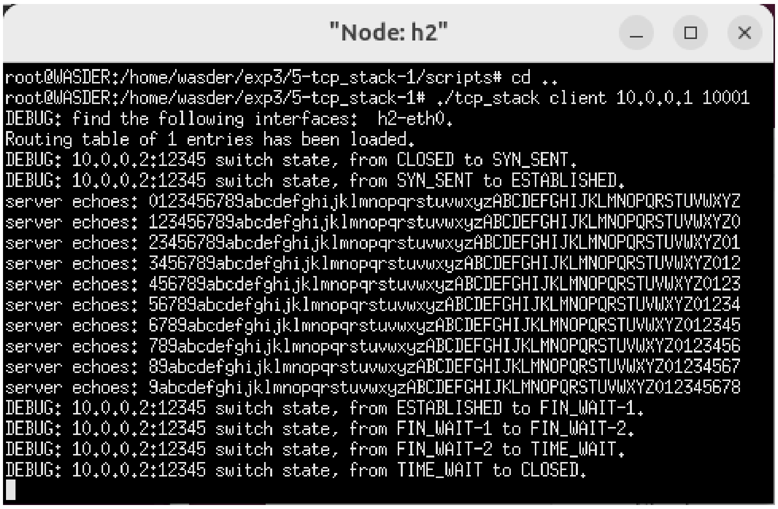
```
mininet> xterm h2
h2# cd scripts/
h2# ./disable_offloading.sh
h2# ./disable_tcp_rst.sh
```

(7) 在节点 h2 上执行 TCP 程序：在 h2 上运行 TCP 协议栈的客户端模式，连接 h1 并正确收发数据（./tcp_stack client 10.0.0.1 10001）：client 向 server 发送数据，server 将数据 echo 给 client（本实验 server 和 client）

```
h2# cd..
h2# ./tcp_stack client 10.0.0.1 10001
```

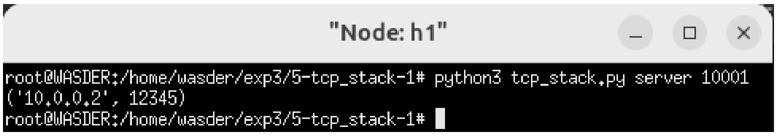


```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1/scripts# cd ..
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from LISTEN to SYN_RECV.
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: 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.
```

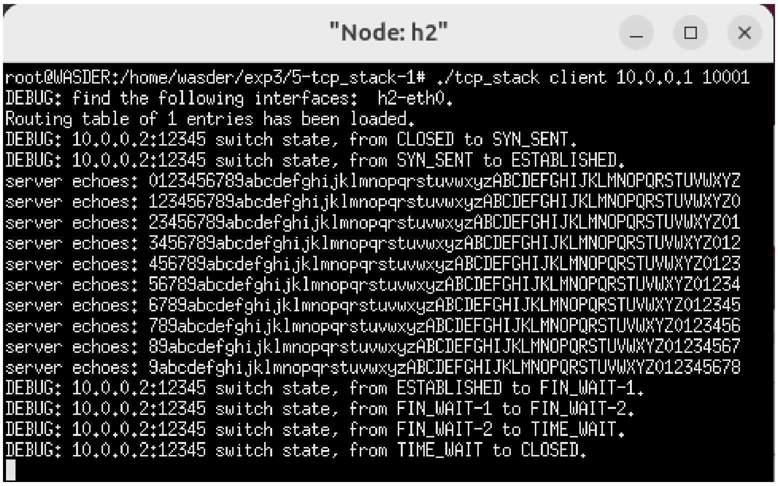


```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1/scripts# cd ..
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 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: 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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

(8) 使用 tcp_stack.py 替换其中任意一端，对端都能正确收发数据（标准 server 和本实验 client）

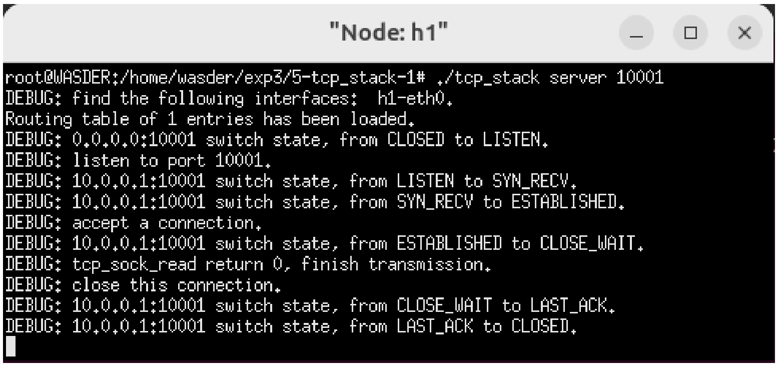


```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# python3 tcp_stack.py server 10001
('10.0.0.2', 12345)
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# 
```

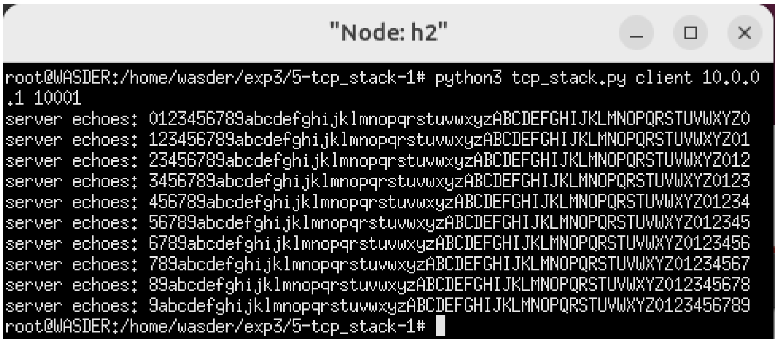


```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
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: 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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

(9) 使用 tcp_stack.py 替换其中任意一端，对端都能正确收发数据（本实验 server 和标准 client）



```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
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 LISTEN to SYN_RECV.
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: 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.
```



```
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# python3 tcp_stack.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
root@WASDER:/home/wasder/exp3/5-tcp_stack-1# 
```

2、 实验内容二

(1) 修改 tcp_apps.c，使之能够收发文件

```
#define SEND_ONCE_SIZE 10000

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 *rbuf = malloc(SEND_ONCE_SIZE + 1);
    char wbuf[1024];
    int rlen = 0;
    FILE *fp = fopen("server-output.dat","wb");
    int write_len = 0;
    int num = 0;

    while (1) {
    //rlen = tcp_sock_read(csk, rbuf, 1000);
    rlen = tcp_sock_read(csk, rbuf, SEND_ONCE_SIZE);
        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);

            write_len = fwrite(rbuf, 1, rlen, fp);
        if (write_len != rlen) {
            log(ERROR, "write: %d, rlen: %d", write_len, rlen);
            exit(1);
        }
        log(DEBUG, "write: %d", write_len);

        num += write_len;
        sprintf(wbuf, "server echoes: recv ok (%d)", num);
        if (tcp_sock_write(csk, wbuf, strlen(wbuf)) < 0) {
            log(DEBUG, "tcp_sock_write return negative value, something
goes wrong.");
            exit(1);
        }
    }
    else {
        log(DEBUG, "tcp_sock_read return negative value, something goes
wrong.");
        exit(1);
    }

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

    tcp_sock_close(csk);

    fclose(fp);

    return NULL;
}

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);

   FILE *fp = fopen("client-input.dat", "r");
   fseek(fp, 0, SEEK_END);
   int dlen = ftell(fp);
   char *wbuf = malloc(dlen+1);
   fseek(fp, 0, SEEK_SET);
   fread(wbuf, 1, dlen, fp);

   char rbuf[1001];
   int rlen = 0;
   int remain_len = dlen;
   int send_ptr = 0;
   int send_len;
   /*
   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);

       if (tcp_sock_write(tsk, wbuf, dlen) < 0)
           break;
   */
   while (remain_len > 0) {
       send_len = min(remain_len, SEND_ONCE_SIZE);
       if (tcp_sock_write(tsk, &wbuf[send_ptr], send_len) < 0) {
           log(ERROR, "socket write failed");
           break;
```

```
            }

            send_ptr += send_len;
            remain_len -= send_len;
            log(DEBUG, "send: %d, remain: %d, total: (%d/%d)", send_len,
        remain_len, send_ptr, dlen);

            usleep(50000);

            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 negative value, something goes
wrong.");
                exit(1);
            }
            sleep(1);
        }
        tcp_sock_close(tsk);

        free(wbuf);

        fclose(fp);

        return NULL;
    }
```

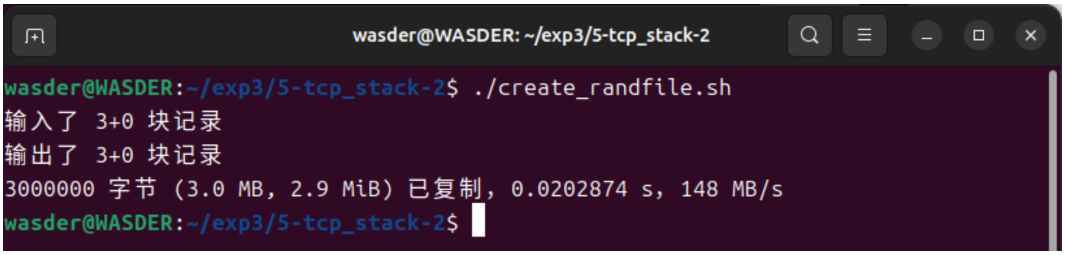
(2) 修改 tcp_stack.py，使之能够收发文件

```
    def server(port):
        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)

    f = open('server-output.dat', 'wb')

    while True:
        data = cs.recv(1000)
        print (len(data))
        f.write(data)
        if data:
            if PYVER == 3:
                #data = "server echoes: " + data.decode()
                data = "server echoes: recv ok"
                cs.send(data.encode())
            else:
                #data = 'server echoes: ' + data
                data = 'server echoes: recv ok'
                cs.send(data)
            else:
                break

    f.close()
    s.close()

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

    f = open('client-input.dat', 'r')
    file_str = f.read()
    length = len(file_str)
    i = 0
    '''
    for i in range(10):
        new_data = data[i:] + data[:i+1]
        if PYVER == 3:
            s.send(new_data.encode())
            print(s.recv(1000).decode())
        else:
            s.send(new_data)
            print(s.recv(1000))
        sleep(1)
    '''
    while length > 0:
        send_len = min(length, 10000)
        new_data = file_str[i: i+send_len]
        if PYVER == 3:
            s.send(new_data.encode())
            sleep(1)
            print(s.recv(1000).decode())
        else:
            s.send(new_data)
            sleep(1)
            print(s.recv(1000))
        length -= send_len
        i += send_len

    f.close()
    s.close()
```

(3) 编译生成 tcp_stack

```
wasder@WASDER:~/exp3/5-tcp_stack-2$ make
```

(4) 执行 create_randfile.sh，生成待传输数据文件 client-input.dat



```
wasder@WASDER:~/exp3/5-tcp_stack-2$ ./create_randfile.sh
输入了 3+0 块记录
输出了 3+0 块记录
3000000 字节 (3.0 MB, 2.9 MiB) 已复制, 0.0202874 s, 148 MB/s
wasder@WASDER:~/exp3/5-tcp_stack-2$ 
```

(5) 运行给定网络拓扑（tcp_topo.py）

```
wasder@WASDER:~/exp3/5-tcp_stack-2$ sudo python3 tcp_topo.py
```

(6) 在节点 h1 上执行 TCP 程序：执行脚本（disable_offloading.sh, disable_tcp_rst.sh）

```
mininet> xterm h1
h1# cd scripts/
h1# ./disable_offloading.sh
h1# ./disable_tcp_rst.sh
```

(7) 在节点 h1 上执行 TCP 程序：在 h1 上运行 TCP 协议栈的服务器模式（./tcp_stack server 10001）

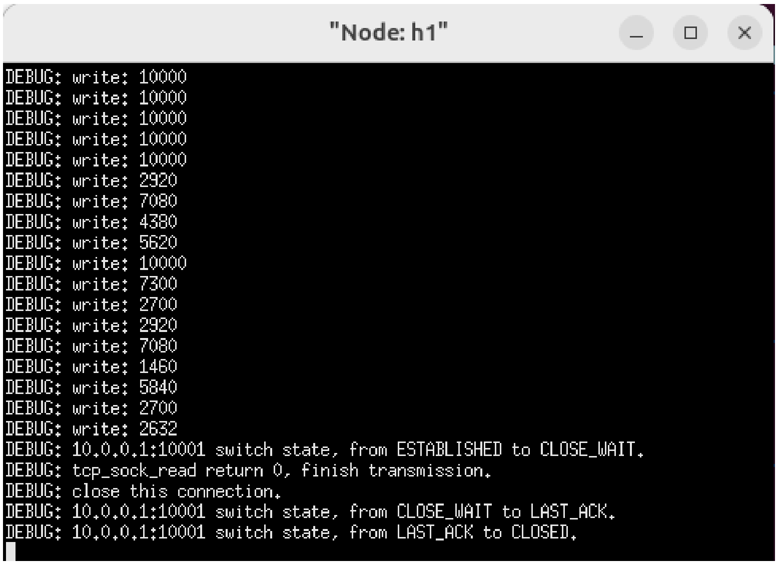
```
h1# cd..
h1# ./tcp_stack server 10001
```

(8) 在节点 h2 上执行 TCP 程序：执行脚本（disable_offloading.sh, disable_tcp_rst.sh）

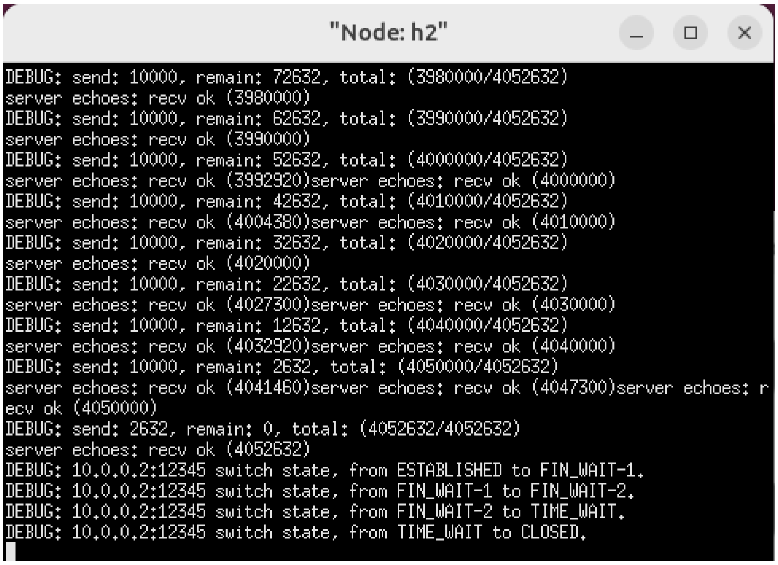
```
mininet> xterm h2
h2# cd scripts/
h2# ./disable_offloading.sh
h2# ./disable_tcp_rst.sh
```

(9) 在节点 h2 上执行 TCP 程序：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 将收到的数据存储到文件 server-output.dat（本实验 server 和 client）

```
h2# cd..
h2# ./tcp_stack client 10.0.0.1 10001
```

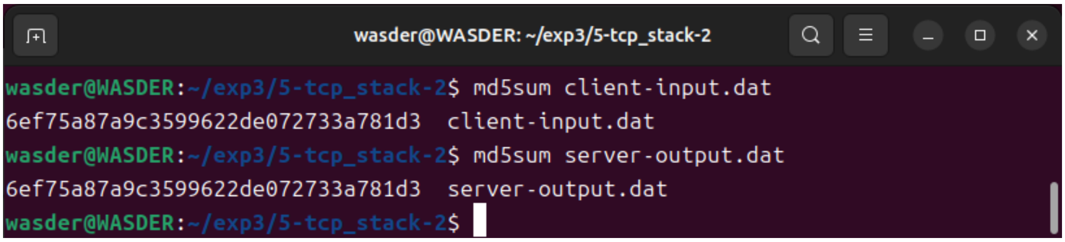


```
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 2920
DEBUG: write: 7080
DEBUG: write: 4380
DEBUG: write: 5620
DEBUG: write: 10000
DEBUG: write: 7300
DEBUG: write: 2700
DEBUG: write: 2920
DEBUG: write: 7080
DEBUG: write: 1460
DEBUG: write: 5840
DEBUG: write: 2700
DEBUG: write: 2632
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: 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: send: 10000, remain: 72632, total: (3980000/4052632)
server echoes: recv ok (3980000)
DEBUG: send: 10000, remain: 62632, total: (3990000/4052632)
server echoes: recv ok (3990000)
DEBUG: send: 10000, remain: 52632, total: (4000000/4052632)
server echoes: recv ok (3992920)server echoes: recv ok (4000000)
DEBUG: send: 10000, remain: 42632, total: (4010000/4052632)
server echoes: recv ok (4004380)server echoes: recv ok (4010000)
DEBUG: send: 10000, remain: 32632, total: (4020000/4052632)
server echoes: recv ok (4020000)
DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
server echoes: recv ok (4027300)server echoes: recv ok (4030000)
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
server echoes: recv ok (4032920)server echoes: recv ok (4040000)
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
server echoes: recv ok (4041460)server echoes: recv ok (4047300)server echoes: r
ecv ok (4050000)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
server echoes: recv ok (4052632)
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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

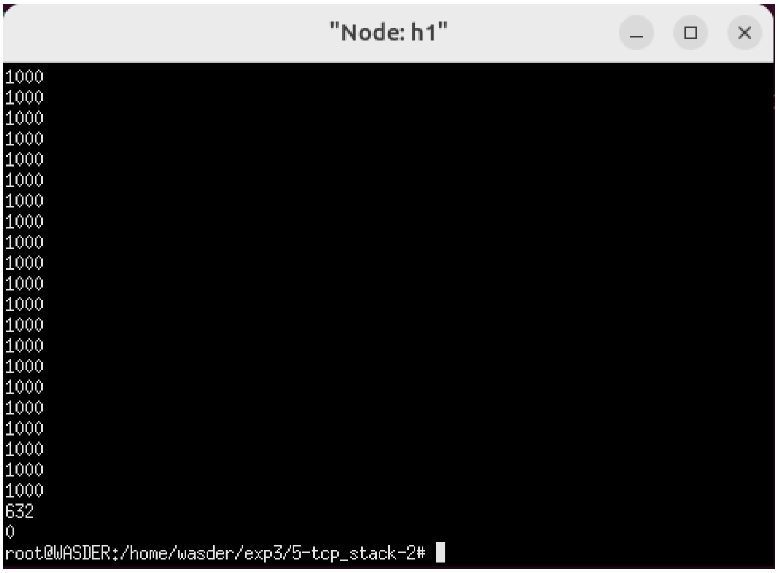
(10) 使用 md5sum 比较两个文件是否完全相同



wasder@WASDER: ~/exp3/5-tcp_stack-2

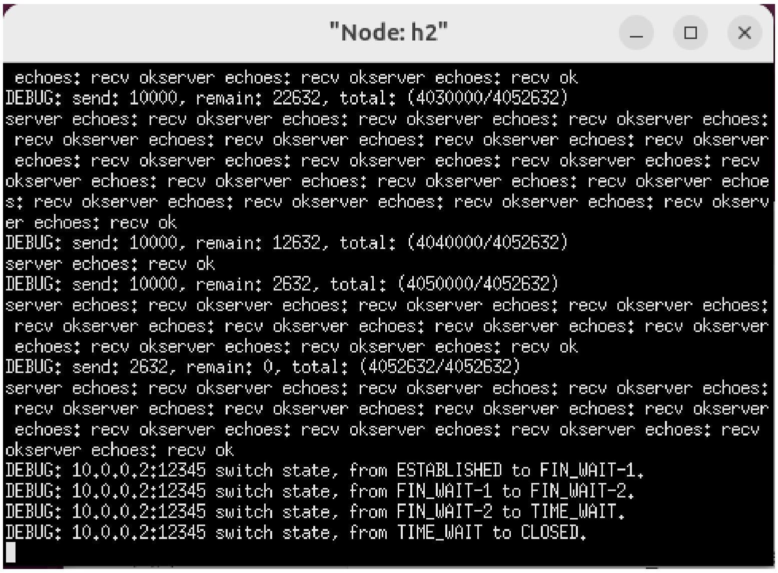
```
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum client-input.dat
6ef75a87a9c3599622de072733a781d3  client-input.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum server-output.dat
6ef75a87a9c3599622de072733a781d3  server-output.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ 
```

(11) 使用 tcp_stack.py 替换其中任意一端，对端都能正确收发数据（标准 server 和本实验 client）



"Node: h1"

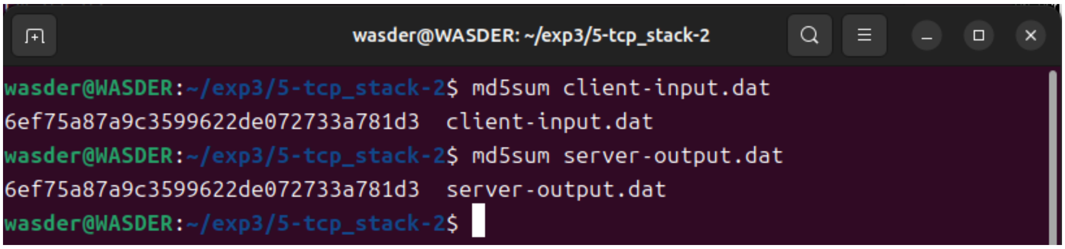
```
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
632
0
root@WASDER:/home/wasder/exp3/5-tcp_stack-2# 
```



"Node: h2"

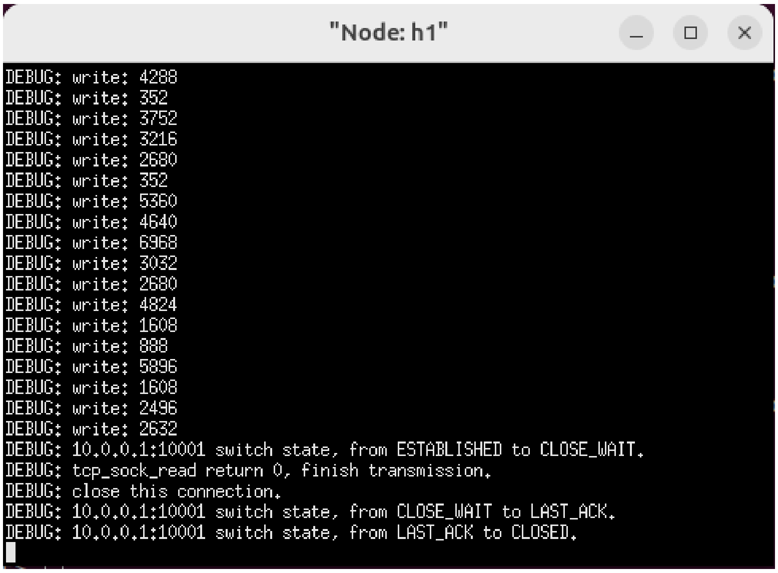
```
 echoes: recv okserver echoes: recv okserver echoes: recv ok
DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
server echoes: recv okserver echoes: recv okserver echoes: recv okserver echoes:
 recv okserver echoes: recv okserver echoes: recv okserver echoes: recv okserver
 echoes: recv okserver echoes: recv okserver echoes: recv okserver echoes: recv
okserver echoes: recv okserver echoes: recv okserver echoes: recv okserver echoe
s: recv okserver echoes: recv okserver echoes: recv okserver echoes: recv okserv
er echoes: recv ok
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
server echoes: recv ok
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
server echoes: recv okserver echoes: recv okserver echoes: recv okserver echoes:
 recv okserver echoes: recv okserver echoes: recv okserver echoes: recv okserver
 echoes: recv okserver echoes: recv okserver echoes: recv ok
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
server echoes: recv okserver echoes: recv okserver echoes: recv okserver echoes:
 recv okserver echoes: recv okserver echoes: recv okserver echoes: recv okserver
 echoes: recv okserver echoes: recv okserver echoes: recv okserver echoes: recv
okserver echoes: recv ok
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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

(12) 使用 md5sum 比较两个文件是否完全相同（标准 server 和本实验 client）

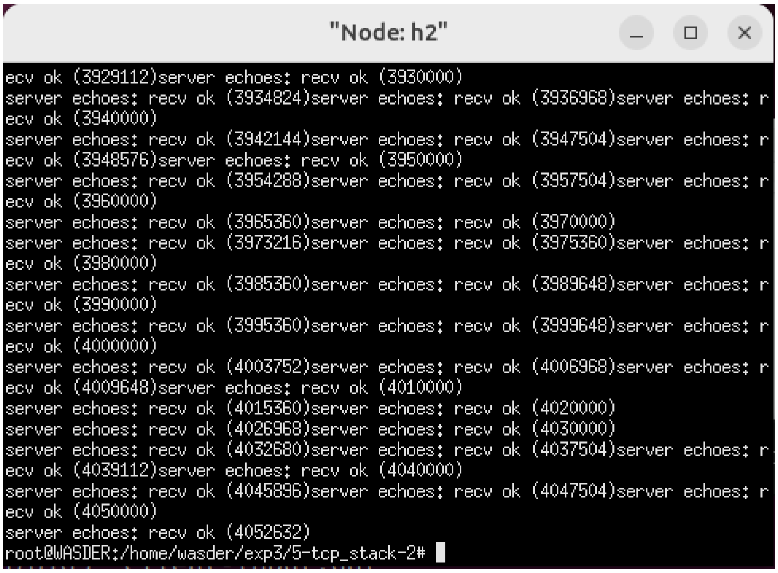


```
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum client-input.dat
6ef75a87a9c3599622de072733a781d3  client-input.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum server-output.dat
6ef75a87a9c3599622de072733a781d3  server-output.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ 
```

(13) 使用 tcp_stack.py 替换其中任意一端，对端都能正确收发数据（本实验 server 和标准 client）

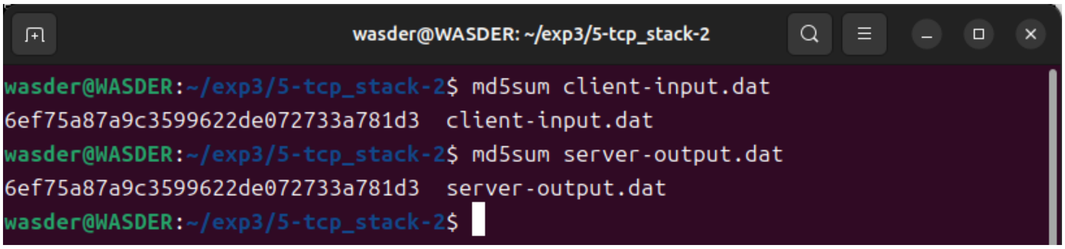


```
DEBUG: write: 4288
DEBUG: write: 352
DEBUG: write: 3752
DEBUG: write: 3216
DEBUG: write: 2680
DEBUG: write: 352
DEBUG: write: 5360
DEBUG: write: 4640
DEBUG: write: 6968
DEBUG: write: 3032
DEBUG: write: 2680
DEBUG: write: 4824
DEBUG: write: 1608
DEBUG: write: 888
DEBUG: write: 5896
DEBUG: write: 1608
DEBUG: write: 2496
DEBUG: write: 2632
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: 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.
```



```
ecv ok (3929112)server echoes: recv ok (3930000)
server echoes: recv ok (3934824)server echoes: recv ok (3936968)server echoes: r
ecv ok (3940000)
server echoes: recv ok (3942144)server echoes: recv ok (3947504)server echoes: r
ecv ok (3948576)server echoes: recv ok (3950000)
server echoes: recv ok (3954288)server echoes: recv ok (3957504)server echoes: r
ecv ok (3960000)
server echoes: recv ok (3965360)server echoes: recv ok (3970000)
server echoes: recv ok (3973216)server echoes: recv ok (3975360)server echoes: r
ecv ok (3980000)
server echoes: recv ok (3985360)server echoes: recv ok (3989648)server echoes: r
ecv ok (3990000)
server echoes: recv ok (3995360)server echoes: recv ok (3999648)server echoes: r
ecv ok (4000000)
server echoes: recv ok (4003752)server echoes: recv ok (4006968)server echoes: r
ecv ok (4009648)server echoes: recv ok (4010000)
server echoes: recv ok (4015360)server echoes: recv ok (4020000)
server echoes: recv ok (4026968)server echoes: recv ok (4030000)
server echoes: recv ok (4032680)server echoes: recv ok (4037504)server echoes: r
ecv ok (4039112)server echoes: recv ok (4040000)
server echoes: recv ok (4045896)server echoes: recv ok (4047504)server echoes: r
ecv ok (4050000)
server echoes: recv ok (4052632)
root@WASDER:/home/wasder/exp3/5-tcp_stack-2# 
```

(14) 使用 md5sum 比较两个文件是否完全相同（本实验 server 和标准 client）



```
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum client-input.dat
6ef75a87a9c3599622de072733a781d3  client-input.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ md5sum server-output.dat
6ef75a87a9c3599622de072733a781d3  server-output.dat
wasder@WASDER:~/exp3/5-tcp_stack-2$ 
```