计算机网络实验报告 tcp stack

2019K8009907018 王畅路

实验内容

- 补全实验代码，实现对 TCP 状态机的状态维护和更新。
- 运行给定网络拓扑，用两个节点分别作为服务器节点和客户端节点，验证握手机制的正确性。
- 补全实验代码，添加`tcp_sock`的读写函数，并调整相关代码，实现 TCP 接收的数据缓存机制。
- 运行给定网络拓扑，用两个节点分别作为服务器节点和客户端节点，数据缓存的正确性。
- 修改`tcp_apps.c`中的相关代码，实现文件收发，并检验其正确性

实验流程

TCP状态转换

TCP 状态机切换的函数是本次实验中被动响应和切换 TCP 传输状态的核心。如下图所示，代码中考虑了状态信号为 SYN 和 SYN|ACK 和 ACK 和 ACK|FIN 和 FIN 共计 5 种可能的情况；每种情况又根据本地TCP状态进行分类，使得状态可以被正确切换。

```
    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)) {
                    return;
                }
                struct tcp_sock *csk = tcp_sock_listen_dequeue(tsk);
                tcp_sock_accept_enqueue(csk);
                //tcp_set_state(csk, TCP_ESTABLISHED);
                csk->rcv_nxt = cb->seq;
                csk->snd_una = cb->ack;
                wake_up(tsk->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);
            }
            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;
        }
    }
```

以最为复杂的接收到 ACK 消息的情况为例进行分析，共有三种状态可能会接收到 ACK 消息，其他状态接收到时输出错误信息。对于 SYN_RECV 状态，需要唤醒待响应的队列，并切换状态至 ESTABLISHED；对于 FIN_WAIT_1 状态，只需切换状态至 FIN_WAIT_2；对于 LAST_ACK 状态，接收到 ACK 消息意味着传输结束，所以置状态为 CLOSED，并释放资源。接收到其他消息的情况不赘述。

超时中断函数

本部分的核心函数是`tcp_scan_timer_list`函数，用于定时唤醒扫描超时队列。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。、

```
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);
        }
    }
}
```

socket管理函数

listen

设置 backlog，将状态切换到 LISTEN，并将 socket 加入 listen hash 表

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    return tcp_hash(tsk);
}
```

accept

作为服务器的一方等待连接，如果没有成功连接的 socket，即 accept_queue 为空，阻塞等待。当被唤醒后，从 accept_queue 获取成功建立连接的 child socket，将其状态转换为 ESTABLISHED，并加入到 established hash表。最后返回 child socket。

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
    while (list_empty(&tsk->accept_queue)) {
        sleep_on(tsk->wait_accept);
    }
    struct tcp_sock *child;
    if ((child = tcp_sock_accept_dequeue(tsk)) != NULL) {
        tcp_set_state(child, TCP_ESTABLISHED);
        if (tcp_hash(child) == 0)
            return child;
        else
            return NULL;
    }
```

```
    return NULL;
}
```

close

如下图所示，根据触发该函数时的状态区分被动建立方和主动建立方，并分类讨论。

```
void tcp_sock_close(struct tcp_sock *tsk)
{
    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;
        }
    }
}
```

TCP数据发送

TCP 数据发送的第一步是确定发送数据的长度。考虑到缓冲区的数据可能无法一次性发送完毕，所以添加了一个 while 循环来实现对输入 buf 的检查。在`tcp_sock_write`函数中仅需要将恰当长度的数据拷贝进入数据包空间的对应位置即可，各头部信息会在后续的函数调用中被逐层添加。发送结束后记录已发送长度，并将自身挂起进入已发送队列，直至接收到来自对端的 ACK 信息才会被唤醒，从而防止发送速度过快引起错误。

```
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;
}
```

TCP数据接受

由于涉及环形 buffer 的读取，所以`tcp_sock_read`函数需要在互斥锁内完成。首先循环检测接收缓存是否为空，为空时释放锁资源并挂起，等待数据到达后重新申请锁资源。数据读取直接调用框架内函数即可。

```
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;
}
```

TCP 协议栈功能

协议栈需要实现收到数据传输相关数据包的处理，在 tcp_process 函数中增加 ESTABLISHED 状态下的数据包处理。如下图所示，TCP_ACK 部分为增加内容，对于收到有效数据（pl_len > 0）和只收到 ACK 分别处理。对于ACK，seq 和 ack 并不增加，不过要根据 ACK 数据包更自己的发送窗口。对于有数据的数据包，则交由
handle_recv_data 函数进一步处理。

```
    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;
        }
```

在实际写缓存前，还需要获取读写锁。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。
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_recv。最后更新 seq 和 ack，并返回 ACK 数据包，表示已接收到数据。

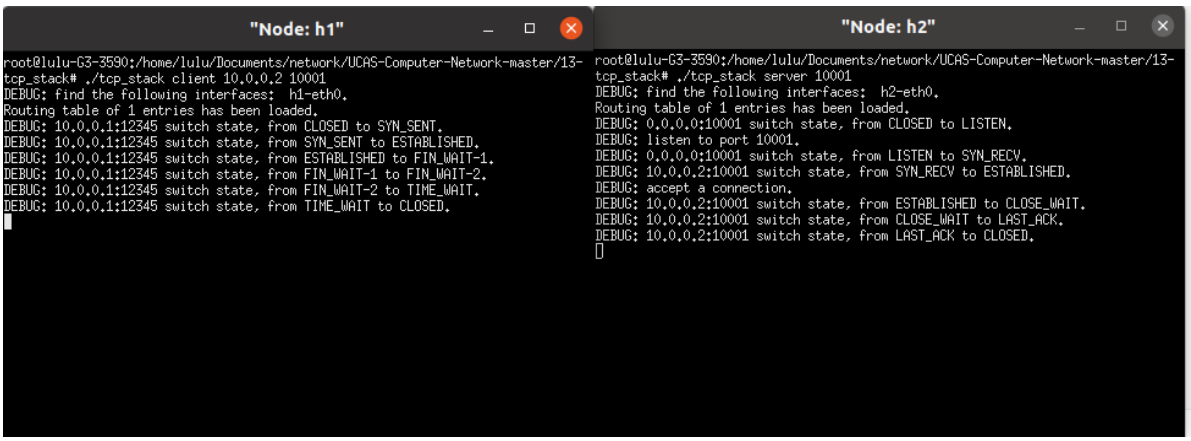
```
    pthread_mutex_lock(&tsk->rcv_buf->rw_lock);
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    tsk->rcv_wnd -= cb->pl_len;
    pthread_mutex_unlock(&tsk->rcv_buf->rw_lock);
    wake_up(tsk->wait_recv);

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

实验结果分析

连接建立

如下图所示，执行`tcp_topo.py`脚本后，将 h2 节点作为 server 端，将 h1 节点作为 client 端，建立 tcp 连
接。两节点间的交互结果如下图所示，可见连接正常。



消息收发

本实验 server 和 client
H2：本实验 server
H1：本实验 client

```
tcp_stack# ./tcp_stack client 10.0.0.2 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.1:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.1:12345 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: 10.0.0.1:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.1:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.1:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.1:12345 switch state, from TIME_WAIT to CLOSED.
```

```
tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces:  h2-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: 10.0.0.2:10001 switch state, from LISTEN to SYN_RECV.
DEBUG: 10.0.0.2:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.2:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: 10.0.0.2:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.2:10001 switch state, from LAST_ACK to CLOSED.
```

文件收发

本实验 server 和 client
H2：本实验 server
H1：本实验 client

```
DEBUG: write: 10000
DEBUG: write: 2920
DEBUG: write: 7080
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
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: send: 10000, remain: 52632, total: (4000000/4052632)
server echoes: recv ok (4000000)
DEBUG: send: 10000, remain: 42632, total: (4010000/4052632)
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 (4030000)
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
server echoes: recv ok (4040000)
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
server echoes: recv ok (4050000)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
server echoes: recv ok (4052632)
```

总结

实验实现了基本的接收缓存相关维护，涉及的细节同样相当丰富。让自己编写的服务器和客户端成功运行并不是件难事，但在加入对既有 API 的协同测试后，很多问题都会暴露出来。