网络传输机制实验三

中国科学院大学
袁欣怡 2018K8009929021
2021.6.29

实验内容

在之前的实验中，我们进行的是无丢包网络环境下的字符串和文件的传输。本次实验中，网络中可能存在丢包的情况，因此需要实现重传机制，进一步实现数据的可靠传输。具体需要实现的内容包括：

1. 维护发送队列。保存所有未收到 ACK 的包，以备后面需要重传。
2. 维护两个接收队列。有序接收队列用来保存数据供 app 读取，无序接收队列用来保存不连续的数据，等他们连起来后放入有序接收队列。
3. 维护超时重传计时器。在 tcp_sock 中添加定时器，并在合适的时候开启和关闭。

实验流程

1. 搭建实验环境

本实验中涉及到的文件主要有：

- main.c：编译后生成 tcp_stack 可执行文件，在两个 host 结点上运行。
- Makefile：处理终端 make all 和 make clean 命令。
- tcp_apps.c：完成服务器端运行的函数 tcp_server 和用户端运行的的函数 tcp_client，配合实现文件的 echo 。
- tcp_in.c：接收报文并处理需要的函数。本次实验修改其中 tcp_process 函数。

- tcp_sock.c： socket 相关操作需要的函数。本次实验修改其中 tcp_sock_read 和 tcp_sock_write 函数，并添加了处理发送队列和接收队列的函数。
- tcp_timer.c： 完成定时器相关的函数。本次实验中添加了关于超时重传计时器的处理。
- tcp_topo.py： 网络拓扑结构。本次实验在之前的基础上添加了2%的丢包率。

2. 实验代码设计

发送队列

所有没有收到 ACK 的数据、 FIN 、 SYN 包，均需要在发送后放入发送队列中，以备后面需要重传。我们需要在 tcp_out.c 中的发送函数中添加函数 send_buffer_ADD_TAIL 来实现这个功能。此函数的具体代码实现（tcp_sock.c）：

```
// add new packet to the tail of send_buffer
void send_buffer_ADD_TAIL(char* packet,int len)
{
    char* packet_copy = (char*)malloc(len);
    memcpy(packet_copy, packet, len);
    struct tcp_send_buffer_block* block = new_tcp_send_buffer_block();
    block->len = len;
    block->packet = packet_copy;
    int tcp_data_len = len- ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE;

    pthread_mutex_lock(&send_buffer.lock);
    send_buffer.size += tcp_data_len;
    list_add_tail(&block->list, &send_buffer.list);
    pthread_mutex_unlock(&send_buffer.lock);

    return ;
}
```

其中 tcp_send_buffer_block 是本次实验中新设计的数据结构，一个 block 用来记录通过 tcp_send_packet 函数发送出去的一个数据包。代码实现在 tcp_sock.h 中：

```
struct tcp_send_buffer{
    struct list_head list;
    int size;
    pthread_mutex_t lock;
    pthread_t thread_retrans_timer;
} send_buffer;

struct tcp_send_buffer_block{
    struct list_head list;
    int len; //length of packet
    char* packet;
};
```

当超时重传计时器判断需要重传时，调用函数 send_buffer_RETRAN_HEAD 重传发送队列中的第一个数据包。此函数的具体代码实现（tcp_sock.c）：

```
// Retrans the first packet in send_buffer
void send_buffer_RETRAN_HEAD(struct tcp_sock *tsk)
{
    if(list_empty(&send_buffer.list))
        return ;

    struct tcp_send_buffer_block *first_block = list_entry(send_buffer.list.next,struct
tcp_send_buffer_block, list);
    char* packet = (char*)malloc(first_block->len);
    memcpy(packet, first_block->packet, first_block->len);
    struct iphdr *ip = packet_to_ip_hdr(packet);
    struct tcphdr *tcp = (struct tcphdr *)((char *)ip + IP_BASE_HDR_SIZE);

    tcp->ack = htonl(tsk->rcv_nxt);
    tcp->checksum = tcp_checksum(ip, tcp);
    ip->checksum = ip_checksum(ip);

    ip_send_packet(packet, first_block->len);
    return ;
}
```

当收到ACK后，对应的包从发送队列中出队。我们在 tcp_in.c 中调用函数 send_buffer_ACK 来实现这个功能。此函数的具体代码实现（tcp_sock.c）：

```
// delete packets(seq<=ack) from send_buffer
void send_buffer_ACK(struct tcp_sock *tsk, u32 ack)
{
    struct tcp_send_buffer_block *block,*block_q;

    list_for_each_entry_safe(block, block_q, &send_buffer.list, list){
        struct iphdr *ip = packet_to_ip_hdr(block->packet);
        struct tcphdr *tcp = (struct tcphdr *)((char *)ip + IP_BASE_HDR_SIZE);
        int ip_tot_len = block->len - ETHER_HDR_SIZE;
        int tcp_data_len = ip_tot_len - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE;

        u32 seq = ntohl(tcp->seq);

        if( (less_than_32b(seq, ack)) ){
            pthread_mutex_lock(&send_buffer.lock);
            send_buffer.size -= tcp_data_len;
            list_delete_entry(&block->list);
            pthread_mutex_unlock(&send_buffer.lock);

            free(block->packet);
            free(block);
        }
```

```
    }
    return ;
}
```

如果三次重传都没有收到 ACK，则清空发送队列，释放连接（tcp_sock.c）：

```
// Retrans 3 times receiving no ACK
// Release this connection
void send_buffer_free()
{
    struct tcp_send_buffer_block *block,*block_q;

    list_for_each_entry_safe(block, block_q, &send_buffer.list, list){
        pthread_mutex_lock(&send_buffer.lock);
        list_delete_entry(&block->list);
        pthread_mutex_unlock(&send_buffer.lock);
        free(block->packet);
        free(block);
    }
    send_buffer.size = 0;
    return ;
}
```

接收队列

（1）有序接收队列

有序接收队列沿用上个实验中实现的环形缓存 ring_buffer 。

（2）乱序接收队列

设计乱序接收缓存块 tcp_ofo_block，每个 block 表示乱序收到的一个包。代码实现在 tcp_sock.h 中：

```
struct tcp_ofo_block {
    struct list_head list;
    u32 seq; // seq of the packet
    u32 len; // length of data
    char* data;
};
```

收到乱序数据包时，将数据包添加到 ofo 队列中，具体实现在 tcp_sock.c 中：

```
// put out_of_order packets into ofo_block
void ofo_packet_enqueue(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
    struct tcp_ofo_block* latest_ofo_block = (struct tcp_ofo_block*)malloc(sizeof(struct tcp_ofo_block));
    latest_ofo_block->seq = cb->seq;
    latest_ofo_block->len = cb->pl_len;
    latest_ofo_block->data = (char*)malloc(cb->pl_len);

```

```
    char* data_segment = packet +ETHER_HDR_SIZE +IP_BASE_HDR_SIZE
+TCP_BASE_HDR_SIZE;
    memcpy(latest_ofo_block->data, data_segment, cb->pl_len);

    int Isinserted = 0;
    struct tcp_ofo_block *block, *block_q;

    list_for_each_entry_safe(block, block_q, &tsk->rcv_ofo_buf, list){
        if(less_than_32b(latest_ofo_block->seq , block->seq)){
            list_add_tail(&latest_ofo_block->list, &block->list);
            Isinserted = 1;
            break;
        }
    }

    if(!Isinserted)
        list_add_tail(&latest_ofo_block->list, &tsk->rcv_ofo_buf);

    return ;
}
```

如果在 ofo 队列中的乱序数据包可以组成有序数据包，则放入有序接收队列中。具体代码实现（tcp_sock.c）：

```
// put in_order packets into recv_buf
int ofo_packet_dequeue(struct tcp_sock *tsk)
{
    u32 seq = tsk->rcv_nxt;
    struct tcp_ofo_block *block, *block_q;

    list_for_each_entry_safe(block, block_q, &tsk->rcv_ofo_buf, list){
        if((seq == block->seq)){ // in order
            while(block->len > ring_buffer_free(tsk->rcv_buf) ){
                fprintf(stdout, "sleep on buff_full \n");
                if(sleep_on(tsk->wait_recv)<0)
                    return 0;
                fprintf(stdout, "wake up \n");
            }

            write_ring_buffer(tsk->rcv_buf, block->data, block->len);
            wake_up(tsk->wait_recv);
            seq += block->len;
            tsk->rcv_nxt = seq;
            list_delete_entry(&block->list);
            free(block->data);
            free(block);
            continue;
        }
        else if(less_than_32b(seq, block->seq)) // not in order
```

```
            break;
        else // error
            return -1;
    }

    return 0;
}
```

超时重传计时器

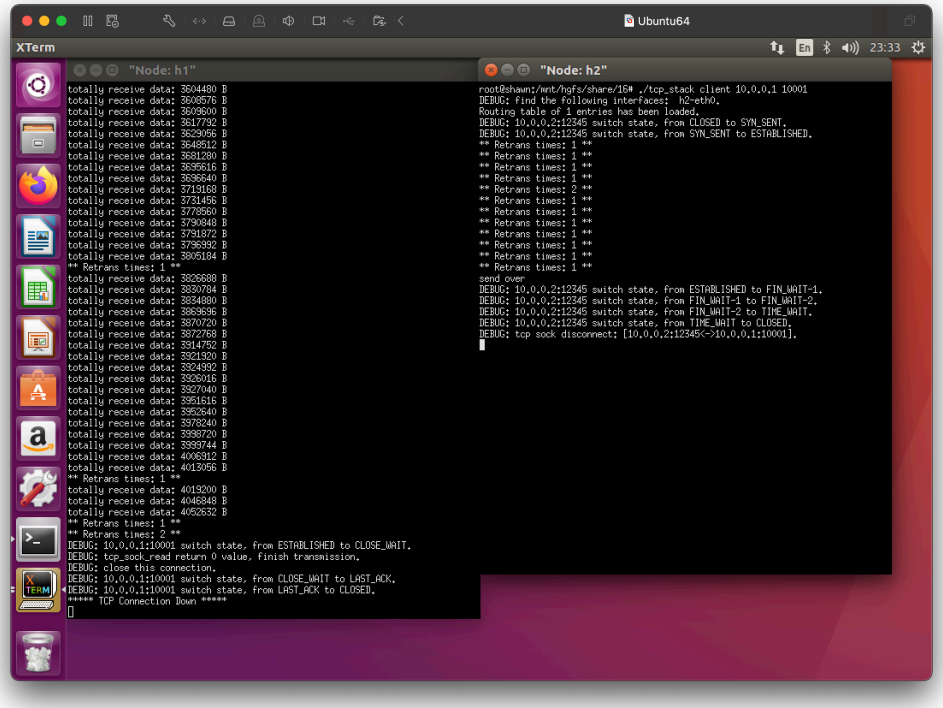
仿照之前设计的timer_list，设计超时重传计时器retrans_timer_list，并添加相应的函数（由于代码较长，不一一列出）：

```
// lab16 added:
void tcp_set_retrans_timer(struct tcp_sock *tsk); // 对timer进行初始化操作，并添加到链表retrans_timer_list中，启动重传计时器
void tcp_update_retrans_timer(struct tcp_sock *tsk); // 还原timer为初始化之后的状态
void tcp_unset_retrans_timer(struct tcp_sock *tsk); // 从retrans_timer_list中删除计时器并释放空间，关闭计时器
void tcp_scan_retrans_timer_list(); // 周期性扫描retrans_timer_list，检查是否有重传次数超过3次的
void *tcp_retrans_timer_thread(void *arg); // 重传计时器线程，每10ms调用一次tcp_scan_retrans_timer_list函数
```

3. 启动脚本进行测试

传输过程：

```
make clean

make all

./create_randfile.sh

sudo python tcp_topo_loss.py

mininet> xterm h1 h2

h1# ./tcp_stack server 10001

h2# ./tcp_stack client 10.0.0.1 10001
```



比较 server-output.dat 和 client-output.dat 内容：

```
shawn@shawn:/mnt/hgfs/share/16$ md5sum server-output.dat
41149212000017c35bfac7d442954edb  server-output.dat
shawn@shawn:/mnt/hgfs/share/16$ md5sum client-input.dat
41149212000017c35bfac7d442954edb  client-input.dat
shawn@shawn:/mnt/hgfs/share/16$ diff server-output.dat client-input.dat
shawn@shawn:/mnt/hgfs/share/16$
```

本次实验调试中，尝试略微调高丢包率以测试网络传输的鲁棒性，表现良好。综上可见，本次实验成功。

实验总结与思考题

本次代码很多，调试很考验耐心。

参考资料

通过学校朋辈辅导时询问了学长实验细节，并参考了2017级李昊宸和蔡润泽同学上传至Github的实现思路。非常感谢学长们的帮助。