tcp 协议实验报告

黄道吉

2020 年 5 月 20 日

传输层协议是互联网的核心协议。这次实验要求实现 TCP 协议中收发数据的部分，并要求实现客户端的 socket 接口。这次实验要求的并不是一个完整的而是简化了的 TCP 协议，即只要求实现客户端的部分，并且发送和接受窗口都设置为 1。

1 数据结构

这次实验的中心是设计 TCB 的结构。在我的实现中，TCB 被设置为一个包含源/目的地址和端口，以及 TCP 协议必须的 seq 和 ack 的 struct。为了实现 TCP 状态转换的功能，还需要一个状态域，它的类型是 TCP_State，是一个枚举的结构，列出了它所有的可能取值。TCB 在初始化时需要为它设置独特的套接字编号，seqnum 和端口，并将状态初始化为关闭连接。

为保存连接状态，维护一个从套接字编号到的映射，这样 socket 函数可以从这个映射中找到对应的 tcb，读取需要的信息。为方便实现的 tcp 各函数能获得 socket 函数使用的 TCB，设置一个全局变量 tcb_cache，让 tcp 函数也能通过这个全局变量获取的信息（如果允许的话也可以在 tcp 函数中加一个参数，传入 TCB 的指针）。

2 实现细节

本节介绍我的实现的细节和只参考手册和查阅资料没有解决的问题。

2.1 函数逻辑

stud_tcp_input 这个函数首先计算 checksum,接收正确的包。之后检查序列号,不对的调用 tcp_DiscardPkt 丢包。在计算了新的序列号和 ack 之后，更新状态机的状态。

stud_tcp_output 这个函数首先构造 tcp 头，计算 checksum，之后发包、更新状态。

stud_tcp_socket 新建一个空的 tcb，加入表中，返回套接字号。

stud_tcp_connect 按照参数确定 tcb 各个域的信息，发送连接报文，接受返回的报文，更新状态。

stud_tcp_send 找到套接字号对应的 tcb，发送报文，等待确认。

stud_tcp_recv 找到 tcb，等待接受报文，发送确认报文。

stud_tcp_close 发送分手报文，等待对端收到和确认分手报文，再回传确认报文。

2.2 残存问题

实现的过程中仍然没有解决的问题有

- 手册中有提到 tcp_sendReport 函数，但是没有对应的 api 介绍，因此实现中没有写进这个函数。
- 手册没有提到 checksum 出错的报文需不需要调用 tcp_DiscardPkt 函数报错。
- stud_tcp_input 函数中发送报文的端口号暂且使用全局的端口号，如果有上机测试的结果，可能需要改成 tcb 当中的端口。
- sockaddr_in 结构似乎没有详细介绍它的各个域是什么
- 手册中要求 stud_tcp_recv 用 sendIpPkt 发送 ack 包，不清楚网络层函数如何发送传输层的包，我的实现使用了 stud_tcp_output 。

3 代码实现

```
/* global variables */
int gSrcPort = 2005;
int gDstPort = 2006;
int gSeqNum = 0;
int gAckNum = 0;
int global_socket_number = 0;

/* states for TCP automata */
typedef enum{TCP_closed, TCP_synsent, TCP_established, TCP_finwait1, TCP_finwait2, TCP_timewait} TCP_State;

/* APIs */
void tcp_DiscardPkt(char * pBuffer, int type);
void tcp_sendIpPkt(unsigned char* pData, uint16 len, unsigned int srcAddr,
unsigned int dstAddr, uint8 ttl);
int waitIpPacket(char *pBuffer, int timeout);
UINT32 getIpv4Address( );
UINT32 getServerIpv4Address( );

struct TCB{
    unsigned int source_address;
    unsigned int destination_address;
    unsigned int source_port;
    unsigned int destination_port;
    unsigned int seq;
    unsigned int ack;
    unsigned int socket_number;
    TCP_State state;

    TCB(){
        socket_number = global_socket_number++;
        seq_number = gSeqNum++;
        state = TCP_closed;
        srcPort = gSrcPort++;
```

```
    }
}

TCB* tcb_cache; // tmp tcb to pass some info from socket functions to tcp functions
map<int, TCB> socket2tcb;

int stud_tcp_input(char *pBuff, unsigned short len, unsigned int srcAddr,
unsigned int dstAddr){
    // where is document for tcp_sendReport function?
    /* check checksum */
    unsigned int header_checksum = 0;
    for(int i = 0; i < len + 20; i += 2){
        header_checksum += (pBuff[i] << 8) + pBuff[i + 1];
    }
    while(header_checksum >> 16){
        header_checksum == (header_checksum >> 16) + (header_checksum & 0xFFFF);
    }
    header_checksum = ~header_checksum;
    if(header_checksum != 0){
        // or call tcp_DiscardPkt()?
        return -1;
    }
    /* convert byte order: ...of what? */

    /* check seq number */
    unsigned int ack_number = *(int* )(pBuff + 8);
    if(ack_number != tcb->seq){
        tcp_DiscardPkt(pBuff, STUD_TCP_TEST_SEQNO_ERROR);
        return -1;
    }

    /* automata, update state? */
    unsigned int seq_number = *(int* )(pBuff + 4);
    tcb_cache->ack = seq_number + 1;
    tcb_cache->seq = ack_number;

    if(tcb_cache->state == TCP_synsent){
        /* what are src and dst port? */
        tcb_cache->state = TCP_established;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, gSrcPort, gDstPort, getIpv4Address(), getServerIpv4Address()
            );
    }else if(tcb_cache->state == TCP_finwait1){
        tcb_cache->state = TCP_finwait2;
    }else if(tcb_cache->state == TCP_finwait2){
        tcb_cache->state = TCP_timewait;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, gSrcPort, gDstPort, getIpv4Address(), getServerIpv4Address()
            );
    }
    return 0;
}

void stud_tcp_output(char *pData, unsigned short len, unsigned char flag,
unsigned short srcPort, unsigned short dstPort, unsigned int srcAddr, unsigned int
dstAddr){
    /* construct TCP head */
    char* head = malloc(20 + len); // no options
    (short* )head[0] = srcPort;
    (short* )head[1] = dstPort;
```

```
    (int* )head[1] = tcb_cache->seq;
    (int* )head[2] = tcb_cache->ack;
    head[13] = 0x50; // first 4-bits being header length (in words)
    head[14] = flag;
    (short* )head[8] = 1; // windowsize: 1
    memcpy(head + 20, pData, len);

    unsigned int header_checksum = 0;
    for(int i = 0; i < len + 20; i += 2){
        header_checksum += (head[i] << 8) + head[i + 1];
    }
    while(header_checksum >> 16){
        header_checksum == (header_checksum >> 16) + (header_checksum & 0xFFFF);
    }
    header_checksum = ~header_checksum;
    (short* )head[9] = header_checksum;

    tcp_sendIpPkt(head, 20 + len, srcAddr, dstAddr, 64); // ttl not specified

    if(flag == PACKET_TYPE_SYN && tcb_cache->state == TCP_closed){
        tcb_cache->state = TCP_synsent;
    }else if(flag == PACKET_TYPE_FIN_ACK && tcb_cache->state == TCP_established){
        tcb_cache->state = TCP_finwait1;
    }
    return ;
}

int stud_tcp_socket(int domain, int type, int protocol){
    TCB* tmp_tcb = new TCB();
    socket2tcb.insert(pair<int, TCB>(tmp_tcb->socket_number, *tcb));
    return tmp_tcb->socket_number;
}

int stud_tcp_connect(int sockfd, struct sockaddr_in* addr, int addrlen){
    map<int, TCB>::iterator it = socket2tcb.find(sockfd);
    tcb_cache = it->second;
    tcb_cache->srcAddr = getIpv4Address();
    tcb_cache->dstAddr = addr->?; // what is a socket structure pointer?
    tcb_cache->dstPort = addr->?; // same as above

    stud_tcp_output(NULL, 0, PACKET_TYPE_SYN, tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache->srcAddr,
        tcb_cache->dstAddr);

    char* head = malloc(200);
    int tmp = -1;
    while(tmp == -1){
        tmp = waitIpPacket(head, 1);
    }
    if(tmp[14] == PACKET_TYPE_SYN_ACK){
        tcb_cache->seq = *(int* )head[1];
        tcb_cache->ack = *(int* )head[2] + 1;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache->srcAddr,
            tcb_cache->dstAddr);
        tcb_cache->state = TCP_established;
        return 0;
    }else{
        return -1;
    }
```

```
    return -1;
}

int stud_tcp_send(int sockfd, const unsigned char* pData, unsigned short datalen,
int flags){
    /* what does "this function will send 'this is a tcp test' to server" mean?
     * should we ignore pData?
     */
    map<int, TCB>::iterator it = socket2tcb.find(sockfd);
    tcb_cache = it->second;
    if(tcb_cache->state == TCP_established){
        stud_tcp_output(pData, datalen, PACKET_TYPE_DATA, tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache->
            srcAddr, tcb_cache->dstAddr);

        char* head = malloc(200);
        int tmp = -1;
        while(tmp == -1){
            tmp = waitIpPacket(head, 1);
        }
        if(tmp[14] == PACKET_TYPE_ACK){
            if((int* )head[2] != tcb_cache->seq + datalen){
                tcp_DiscardPkt(head, STUD_TCP_TEST_SEQNO_ERROR);
                return -1;
            }
            tcb_cache->seq = (int* )head[2];
            tcb_cache->ack = (int* )head[1] + datalen;
            return 0;
        }
        return -1;
    }
    return -1;
}

int stud_tcp_recv(int sockfd, const unsigned char* pData, unsigned short datalen,
int flags){
    map<int, TCB>::iterator it = socket2tcb.find(sockfd);
    tcb_cache = it->second;
    if(tcb_cache->state == TCP_established){
        char* head = malloc(200);
        int tmp = -1;
        while(tmp == -1){
            tmp = waitIpPacket(head, 1);
        }
        memcpy(pData, head + 20, sizeof(head) - 20);
        // how to send ack(level4 packet) through sendIpPkt(level3 datagram)?
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache->srcAddr,
            tcb_cache->dstAddr);
        return 0;
    }
    return -1;
}

int stud_tcp_close( int sockfd ){
    map<int, TCB>::iterator it = socket2tcb.find(sockfd);
    tcb_cache = it->second;

    if(tcb_cache->state == TCP_established){
        tcb_cache->state = TCP_finwait1;
```

```
        stud_tcp_output(NULL, 0, PACKET_TYPE_FIN_ACK, tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache->
            srcAddr, tcb_cache->dstAddr);
        char* head = malloc(200);
        int tmp = -1;
        while(tmp == -1){
            tmp = waitIpPacket(head, 1);
        }
        if(head[14] == PACKET_TYPE_ACK){
            tcb_cache->state = TCP_finwait2;
            int tmp = -1;
            while(tmp == -1){
                tmp = waitIpPacket(head, 1);
            }
            if(head[14] == PACKET_TYPE_FIN_ACK){
                tcb_cache->seq = (int* )head[2];
                tcb_cache->ack = (int* )head[1]++;
                stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, , tcb_cache->srcPort, tcb_cache->dstPort, tcb_cache
                    ->srcAddr, tcb_cache->dstAddr);
                tcb_cache->state = TCP_timewait;
                return 0;
            }
            return -1;
        }
        return -1;
    }
    return -1;
}
```