实验内容

1) 了解 TCP 协议的主要内容，并针对客户端角色的、“停一等”模式的 TCP 协议，完成对接收和发送流程的设计。
2) 实现 TCP 报文的接收流程，重点是报文接收的有限状态机。
3) 实现 TCP 报文的发送流程，完成 TCP 报文的封装处理。
4) 实现客户端 Socket 函数接口。
5) 设计保存 TCP 连接相关信息的数据结构（一般称为 TCB，Transmission Control Block）。
6) TCP 协议的接收处理。

学生需要实现 stud_tcp_input( )函数，完成检查校验和、字节序转换功能（对头部中的选项不做处理），重点实现客户端角色的 TCP 报文接收的有限状态机。不采用捎带确认机制，收到数据后马上回复确认，以满足“停一等”模式的需求。

7) TCP 协议的封装发送。

学生需要实现 stud_tcp_output( )函数，完成简单的 TCP 协议的封装发送功能。为保证可靠传输，要在收到对上一个报文的确认后才能够继续发送。

实现代码思路

要点：设计 TCB 和 TCP_Header，同时设计上应该从用户接口的实现开始考虑，才能明白 stud_tcp_input 和 stud_tcp_output 函数应该怎么设计。

在实现中，维护一张 TCB 表格，和当前的 TCB 指针。可以看出 stud_tcp_input 和 stud_tcp_output 函数的参数是没有 socketfd 的，也就要求在调用这两个函数之前，就必须找到这两个函数操作的 TCB，并且在这个函数中需要修改 ack 和 seq 的值。

TCP 状态机设计：

1、 stud_tcp_socket 函数建立新的 TCB，并且加入 TCB 表中，返回 socketfd。
2、 stud_tcp_connect 函数通过传入的 socketfd，找到需要操作的 TCB 之后，调用 stud_tcp_output 以发送 PACKET_TYPE_SYN 请求同步。之后如果收到正确的回复 PACKET_TYPE_SYN_ACK 就发送 ACK 给对方，完成三次握手，状态进入到 ESTABLISHED 状态
3、 stud_tcp_send 函数通过传入的 socketfd，找到需要操作的 TCB 之后，判断当前 TCB 的状态是否处于 ESTABLISHED 状态，并进行数据发送，等待返回结果。
4、 stud_tcp_recv 函数通过传入的 socketfd，找到需要操作的 TCB 之后，判断当前 TCB 的状态是否处于 ESTABLISHED 状态，并进行数据接收，返回对应 ACK
5、 stud_tcp_close 函数通过传入的 socketfd，找到需要操作的 TCB 之后，判断当前 TCB 的状态是否处于 ESTABLISHED 状态，并进行 FIN1->FIN2->TIMEOUT 的状态转换

问题

手册实现方面写得不够清晰，还是参考了一下其他人的实现思路，比如应该从接口去考虑，而不是一上来就实现 stud_tcp_input（这是因为 stud_tcp_input 函数没有 socketfd 参数，找不到当前的 TCB，从而一开始觉得无从下手。但如果从 stud_tcp_send 函数开始想就容易了，在这个函数中先找到 TCB 然后采用 stud_tcp_input 函数进行操作）。

解决方法

参考了一下其他人的实现思路，比如应该从接口去考虑，而不是一上来就实现 stud_tcp_input（这是因为 stud_tcp_input 函数没有 socketfd 参数，找不到当前的 TCB，从而一开始觉得无从下手。但如果从 stud_tcp_send 函数开始想就容易了，在这个函数中先找到 TCB 然后采用 stud_tcp_input 函数进行操作）。

代码

```
/*
* THIS FILE IS FOR TCP TEST
*/

#include "sysInclude.h"
#include <map>
using namespace std;

// States
#define CLOSED          1
#define SYN_SENT        2
#define ESTABLISHED 3
#define FIN_WAIT1       4
#define FIN_WAIT2       5
#define TIME_WAIT       6

extern void tcp_DiscardPkt(char *pBuffer, int type);
extern void tcp_sendReport(int type);
extern void tcp_sendIpPkt(unsigned char *pData, UINT16 len, unsigned int    srcAddr, unsigned
int dstAddr, UINT8    ttl);
extern int waitIpPacket(char *pBuffer, int timeout);
extern unsigned int getIpv4Address();
extern unsigned int getServerIpv4Address();

int gSrcPort = 2005;
int gDstPort = 2006;
int gSeqNum = 1;
int gAckNum = 1;
int socknum = 1;

// Transmission Control Block
struct TCB
{
        unsigned int srcAddr;
        unsigned int dstAddr;
        unsigned short srcPort;
```

```
        unsigned short dstPort;
        unsigned int seq;
        unsigned int ack;
        int sockfd;
        BYTE state;
        unsigned char* data;

        // Initialization & Update numbers
        void init()
        {
                sockfd = socknum++;
                srcPort = gSrcPort++;
                seq = gSeqNum++;
                ack = gAckNum;
                state = CLOSED;
        }
};

// TCP header
struct TCPHead
{
        UINT16 srcPort;
        UINT16 destPort;
        UINT32 seqNo;
        UINT32 ackNo;
        UINT8 headLen;
        UINT8 flag;
        UINT16 windowsize;
        UINT16 checksum;
        UINT16 urgentPointer;
        char data[100];

        // Little endian to big endian
        void ntoh()
        {
                checksum = ntohs(checksum);
                srcPort = ntohs(srcPort);
                destPort = ntohs(destPort);
                seqNo = ntohl(seqNo);
                ackNo = ntohl(ackNo);
                windowsize = ntohs(windowsize);
                urgentPointer = ntohs(urgentPointer);
        }
```

```
	// Checksum update
	unsigned int CheckSum(unsigned int srcAddr,
				unsigned int dstAddr,
				int type, int len)
	{
		unsigned int sum = 0;
		sum += srcPort + destPort;
		sum += (seqNo >> 16) + (seqNo & 0xFFFF);
		sum += (ackNo >> 16) + (ackNo & 0xFFFF);
		sum += (headLen << 8) + flag;
		sum += windowsize + urgentPointer;
		sum += (srcAddr >> 16) + (srcAddr & 0xffff);
		sum += (dstAddr >> 16) + (dstAddr & 0xffff);
		sum += IPPROTO_TCP;
		sum += 0x14;

		if (type == 1)
		{
			sum += len;
			for (int i = 0; i < len; i += 2)
				sum += (data[i] << 8) + (data[i + 1] & 0xFF);
		}
		sum += (sum >> 16);
		return (~sum) & 0xFFFF;
	}
};


map<int, TCB*> TCBTable;
TCB *tcb;

int stud_tcp_input(char *pBuffer,
			unsigned short len,
			unsigned int srcAddr,
			unsigned int dstAddr)
{
	srcAddr = ntohl(srcAddr);
	dstAddr = ntohl(dstAddr);

	TCPHead* head = (TCPHead *)pBuffer;
	head->ntoh();

	// Check checksum
	if (head->CheckSum(srcAddr, dstAddr, 0, 0) != head->checksum)
```

```
        return -1;
    // Check sequence number
    if (head->ackNo != tcb->seq + (tcb->state != FIN_WAIT2))
    {
        tcp_DiscardPkt(pBuffer, STUD_TCP_TEST_SEQNO_ERROR);
        return -1;
    }
    tcb->ack = head->seqNo + 1;
    tcb->seq = head->ackNo;

    // SYN_SEND to ESTABLISHED by sending ACK
    if (tcb->state == SYN_SENT)
    {
        tcb->state = ESTABLISHED;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK,
                        DEFAULT_TCP_SRC_PORT,
                        DEFAULT_TCP_DST_PORT,
                        getIpv4Address(),
                        getServerIpv4Address());
    }
    // FIN_WAIT1 to FIN_WAIT2 when receiving ACK
    else if (tcb->state == FIN_WAIT1)
        tcb->state = FIN_WAIT2;
    // FIN_WAIT2 to TIME_WAIT by sending ACK
    else if (tcb->state == FIN_WAIT2)
    {
        tcb->state = TIME_WAIT;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK,
                        DEFAULT_TCP_SRC_PORT,
                        DEFAULT_TCP_DST_PORT,
                        getIpv4Address(),
                        getServerIpv4Address());
    }
    else return -1;
    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)
```

```
{
	if (tcb == NULL)
	{
		tcb = new TCB();
		tcb->init();
	}
	// construct and send TCP packet
	TCPHead* head = new TCPHead();
	memcpy(head->data, pData, len);
	head->srcPort = srcPort;
	head->destPort = dstPort;
	head->seqNo = tcb->seq;
	head->ackNo = tcb->ack;
	head->headLen = 0x50;
	head->flag = flag;
	head->windowsize = 1;
	head->checksum = head->CheckSum(srcAddr, dstAddr,
		(flag == PACKET_TYPE_DATA), len);
	head->ntoh();
	tcp_sendIpPkt((unsigned char*)head, 20 + len,
			srcAddr, dstAddr, 60);

	// These state transfers cannot be achieved in stud_tcp_input()
	// CLOSED to SYN_SENT when sending SYN (caused by stud_tcp_connect())
	if (flag == PACKET_TYPE_SYN && tcb->state == CLOSED)
		tcb->state = SYN_SENT;
	// ESTABLISHED to FIN_WAIT1 when sending FIN (caused by stup_tcp_close())
	if (flag == PACKET_TYPE_FIN_ACK && tcb->state == ESTABLISHED)
		tcb->state = FIN_WAIT1;
}

int stud_tcp_socket(int domain,
				int type,
				int protocol)
{
	// Construct TCB and build socket connection
	tcb = new TCB();
	tcb->init();
	TCBTable.insert(std::pair<int, TCB *>(tcb->sockfd, tcb));
	return (socknum - 1);
}

int stud_tcp_connect(int sockfd,
				struct sockaddr_in *addr,
```

```
                                int addrlen)
{
    int res = 0;
    map<int, TCB*>::iterator iter = TCBTable.find(sockfd);
    tcb = iter->second;
    // Set IPv4 addresses
    tcb->dstPort = ntohs(addr->sin_port);
    tcb->state = SYN_SENT;
    tcb->srcAddr = getIpv4Address();
    tcb->dstAddr = htonl(addr->sin_addr.s_addr);

    // Send SYN and start connecting procedure
    stud_tcp_output(NULL, 0, PACKET_TYPE_SYN,
                    tcb->srcPort, tcb->dstPort,
                    tcb->srcAddr, tcb->dstAddr);
    // Wait for response
    TCPHead* r = new TCPHead();
    res = waitIpPacket((char*)r, 5000);
    while (res == -1)
        res = waitIpPacket((char*)r, 5000);

    // Respond by send ACK
    if (r->flag == PACKET_TYPE_SYN_ACK)
    {
        tcb->ack = ntohl(r->seqNo) + 1;
        tcb->seq = ntohl(r->ackNo);
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK,
                        tcb->srcPort, tcb->dstPort,
                        tcb->srcAddr, tcb->dstAddr);
        tcb->state = ESTABLISHED;
        return 0;
    }
    return -1;
}

int stud_tcp_send(int sockfd,
                  const unsigned char *pData,
                  unsigned short datalen,
                  int flags)
{
    int res = 0;
    map<int, TCB*>::iterator iter = TCBTable.find(sockfd);
    tcb = iter->second;
```

```
    // Check if the connection is established
    if (tcb->state == ESTABLISHED)
    {
        // Send the packet
        tcb->data = (unsigned char*)pData;
        stud_tcp_output((char *)tcb->data, datalen, PACKET_TYPE_DATA,
                        tcb->srcPort, tcb->dstPort,
                        getIpv4Address(), tcb->dstAddr);

        // Wait for response
        TCPHead* r = new TCPHead();
        res = waitIpPacket((char*)r, 5000);
        while (res == -1)
            res = waitIpPacket((char*)r, 5000);

        // Check response
        if (r->flag == PACKET_TYPE_ACK)
        {
            // Sequence number error
            if (ntohl(r->ackNo) != (tcb->seq + datalen))
            {
                tcp_DiscardPkt((char*)r, STUD_TCP_TEST_SEQNO_ERROR);
                return -1;
            }
            tcb->ack = ntohl(r->seqNo) + datalen;
            tcb->seq = ntohl(r->ackNo);
            return 0;
        }
    }
    return -1;
}

int stud_tcp_recv(int sockfd,
                  unsigned char *pData,
                  unsigned short datalen,
                  int flags)
{
    int res = 0;
    map<int, TCB*>::iterator iter = TCBTable.find(sockfd);
    tcb = iter->second;

    // Check if the connection is established
    if (tcb->state == ESTABLISHED)
    {
```

```
            // Wait for packet
            TCPHead * r = new TCPHead();
            res = waitIpPacket((char*)r, 5000);
            while (res == -1)
                  res = waitIpPacket((char*)r, 5000);
            memcpy(pData, r->data, sizeof(r->data));
            // Respond by sending ACK
            stud_tcp_output(NULL, 0, PACKET_TYPE_ACK,
                              tcb->srcPort, tcb->dstPort,
                              getIpv4Address(), tcb->dstAddr);
            return 0;
      }
      return -1;
}

int stud_tcp_close(int sockfd)
{
      int res = 0;
      map<int, TCB*>::iterator iter = TCBTable.find(sockfd);
      tcb = iter->second;

      // Check if the socket connection is established
      if (tcb->state == ESTABLISHED)
      {
            // Send FIN
            stud_tcp_output(NULL, 0, PACKET_TYPE_FIN_ACK,
                              tcb->srcPort, tcb->dstPort,
                              getIpv4Address(), tcb->dstAddr);
            tcb->state = FIN_WAIT1;
            TCPHead *r = new TCPHead();

            // Wait for response
            res = waitIpPacket((char*)r, 5000);
            while (res == -1)
                  res = waitIpPacket((char*)r, 5000);

            // Responde by sending ACK
            if (r->flag == PACKET_TYPE_ACK)
            {
                  tcb->state = FIN_WAIT2;
                  res = waitIpPacket((char*)r, 5000);
                  while (res == -1)
                        res = waitIpPacket((char*)r, 5000);
                  if (r->flag == PACKET_TYPE_FIN_ACK)
```

```
                {
                    tcb->ack = ntohl(r->seqNo);
                    tcb->seq = ntohl(r->ackNo);
                    tcb->ack++;
                    stud_tcp_output(NULL, 0, PACKET_TYPE_ACK,
                                    tcb->srcPort, tcb->dstPort,
                                    getIpv4Address(), tcb->dstAddr);
                    tcb->state = TIME_WAIT;
                    return 0;
                }
            }
            return -1;
        }
        delete tcb;
        return -1;
}
```