北 京 大 学

信息科学技术学院

本科生实验报告

实验课程：计算机网络概论

实验名称：TCP 协议实习报告

学生姓名：季凯航

学　　号：1400012727

提交时间：2016 年 12 月 22 日

目录

一、实验目的

传输层是互联网协议栈的核心层次之一,它的任务是在源节点和目的节点间供端到端的、高效的数据传输功能。TCP 协议是主要的传输层协议，它为两个任意处理速率的、使用不可靠 IP 连接的节点之间，供了可靠的、具有流量控制和拥塞控制的、端到端的数据传输服务。TCP 协议不同于 IP 协议，它是有状态的，这也使其成为互联网协议栈中最复杂的协议之一。网络上多数的应用程序都是基于 TCP 协议的，如 HTTP、FTP 等。本实验的主要目的是学习和了解 TCP 协议的原理和设计实现的机制。

TCP 协议中的状态控制机制和拥塞控制算法是协议的核心部分。TCP 协议的复杂性主要源于它是一个有状态的协议，需要进行状态的维护和变迁。有限状态机可以很好的从逻辑上表示 TCP 协议的处理过程，理解和实现 TCP 协议状态机是本实验的重点内容。另外，由于在网络层不能保证分组顺序到达，因而在传输层要处理分组的乱序问题。只有在某个序号之前的所有分组都收到了，才能够将它们一起 交给应用层协议做进一步的处理。

拥塞控制算法对于 TCP 协议及整个网络的性能有着重要的影响。目前对于 TCP 协议研究的一个重要方向就是对于拥塞控制算法的改进。希望通过学习、实现和改进 TCP 协议的拥塞控制算法，增强大家对计算机网络进行深入研究的兴趣。

另外,TCP 协议还要向应用层 供编程接口,即网络编程中所普遍使用的 Socket 函数。通过实现这些接口函数，可以深入了解网络编程的原理， 高网络程序的设计和调试能力。

TCP 协议是非常复杂的，不可能在一个实验中完成所有的内容。出于工作量和实现复杂度的考虑，本实验对 TCP 协议进行适当的简化，只实现客户端角色的、“停－等”模式的 TCP 协议，能够正确的建立和拆除连接，接收和发送 TCP 报文，并向应用层 供客户端需要的 Socket 函数。

二、实验要求

实验要求主要包括:

1) 了解 TCP 协议的主要内容，并针对客户端角色的、“停－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。

三、实验内容

实验内容主要包括:

1) 设计保存TCP 连接相关信息的数据结构(一般称为TCB,Transmission Control Block)。

2) 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－等”模式的需求。

3) 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。

4) TCP 协议 供的Socket 函数接口

实现与客户端角色的 TCP 协议相关的 5 个 Socket 接口函数，stud_tcp_socket( )、stud_tcp_connect ()、stud_tcp_recv ()、stud_tcp_send ()和stud_tcp_close ()，将接口函数实现的内容与发送和接收流程有机地结合起来。

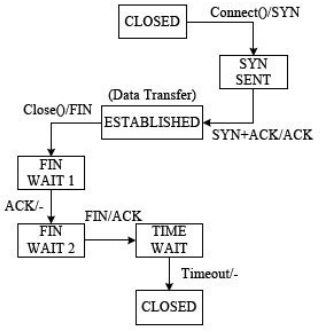
实验内容（1)、(2) 和（3）完成后，其正确性用测试例1——针对分组交互的测试来验证；实验内容（4）的正确性用测试例2——socket API 测试来验证。

四、实验过程

以下概述所需实现的各函数的主要流程。其中，stud_tcp_input() 和 stud_tcp_output()分别处理TCP 段的接收和封装发送；而各Socket 函数是为应用层提供不同功能的接口函数。

1. 客户端状态转换图

客户端的状态转换图示意如下:



2. stud_tcp_input()

该函数的主要流程如下:

- 将所有 TCP 头字段从网络顺序转化为主机顺序。
- 检查收到 TCP 段的校验和，如果出错则返回-1 并结束。注意，校验和的计算要包含伪头部分。
- 检查当前序列号。根据停等式协议，如果 seqNo , ackNo + 1 则调用 tcp_DiscardPkt 抛弃该段。注意状态为 FIN_WAIT2 时，该条件修改为 seqNo ,ackNo。
- 根据当前 TCB 的状态，按照有限状态转换机进行状态转换，同时调用 stud_tcp_output() 发送 ACK 段。

3. stud_tcp_output()

该函数的主要处理流程如下:

- 判断是否可以发送。
- 根据发送的数据段类型填充 TCP 头部各字段，计算校验和（包含伪头和数据），转化字节顺序，并调用 tcp_sendIpPkt 发送 TCP 段。
- 根据当前 TCB 的状态和发送 TCP 段的状态进行状态转换。

4. stud_tcp_socket()

该函数为创建 Socket 的接口函数，主要流程如下:

- 创建新的 TCB 结构，完成初始化过程。
- 将其加入 TCBTable 表中，返回其唯一描述符 socketfd。

5. stud_tcp_connect()

该函数为建立 TCP 连接的 Socket 接口函数，主要流程如下:

- 从 TCBTable 中查找相应的 socketfd, 填充其源和目的端口及地址。
- 三次握手：发送 SYN 报文并等待 SYN + ACK 应答，而后发送 ACK 应答。
- 完成相应的状态转换。

6. stud_tcp_send()

该函数为通过 Socket 发送 TCP 段的接口函数，主要流程如下:

- 判断是否处于 ESTABLISHED 状态。
- 将应用层数据复制到 TCB 缓冲区，调用 stud_tcp_output 函数发送 TCP 数据报。
- 等待 ACK 应答并更改 TCB 的 seqNo、ackNo 等信息。

7. stud_tcp_recv()

该函数为通过 Socket 接收 TCP 段的接口函数，主要流程如下：

- 判断是否处于 ESTABLISHED 状态。
- 等待函数被调用。
- 将数据从 TCB 的缓冲区中读出，交给应用层，并发送 ACK 应答。

8. stud_tcp_close()

该函数为关闭 TCP 连接的 Socket 接口函数，主要流程如下：

- 判断状态为 ESTABLISHED 状态时，进行相应的状态转换。否则直接删除 TCB 结构并退出。
- 调用 stud_tcp_output 函数发送 FIN 报文。
- 等待对方 ACK 应答，收到应答后发送 ACK 并变更状态为 TIME_WAIT。

五、附录——代码

```
#include "sysInclude.h"
#include <map>

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

using std::map;
using std::pair;

extern void tcp_DiscardPkt(char *pBuffer, 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, gAckNum = 1, socknum = 1;
```

```
class TCB {
public:
    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;

    void IniTCB() {
        sockfd = socknum++;
        srcPort = gSrcPort++;
        seq = gSeqNum++;
        ack = gAckNum;
        state = CLOSED;
    }
};

class TCPHead {
public:
    UINT16 srcPort;
    UINT16 destPort;
    UINT32 seqNo;
    UINT32 ackNo;
    UINT8 headLen;
    UINT8 flag;
    UINT16 windowsize;
    UINT16 checksum;
    UINT16 urgentPointer;
    char data[100];

    void NtoH() {
        checksum = ntohs(checksum);
        srcPort = ntohs(srcPort);
        destPort = ntohs(destPort);
        seqNo = ntohl(seqNo);
        ackNo = ntohl(ackNo);
        windowsize = ntohs(windowsize);
        urgentPointer = ntohs(urgentPointer);
    }
```

```
    unsigned int CalChecksum(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();

    if (head->CalChecksum(srcAddr, dstAddr, 0, 0) != head->checksum)
        return -1;

    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;

    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());
    }
    else if (tcb->state == FIN_WAIT1) {
        tcb->state = FIN_WAIT2;
    }
    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->IniTCB();
    }
    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->CalChecksum(srcAddr, dstAddr, (flag ==
PACKET_TYPE_DATA), len);
    head->NtoH();

    tcp_sendIpPkt((unsigned char*)head, 20 + len, srcAddr, dstAddr, 60);

    if (flag == PACKET_TYPE_SYN && tcb->state == CLOSED)
```

```
        tcb->state = SYN_SENT;
    if (flag == PACKET_TYPE_FIN_ACK && tcb->state == ESTABLISHED)
        tcb->state = FIN_WAIT1;
}

int stud_tcp_socket(int domain, int type, int protocol)
{
    tcb = new TCB();
    tcb->IniTCB();
    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;

    tcb->dstPort = ntohs(addr->sin_port);
    tcb->state = SYN_SENT;
    tcb->srcAddr = getIpv4Address();
    tcb->dstAddr = htonl(addr->sin_addr.s_addr);

    stud_tcp_output(NULL, 0, PACKET_TYPE_SYN, tcb->srcPort, tcb->dstPort,
tcb->srcAddr, tcb->dstAddr);
    TCPHead* r = new TCPHead();
    do {
        res = waitIpPacket((char*)r, 5000);
    } while (res == -1);

    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;

    if (tcb->state == ESTABLISHED) {
        tcb->data = (unsigned char*)pData;
        stud_tcp_output((char *)tcb->data, datalen, PACKET_TYPE_DATA,
tcb->srcPort, tcb->dstPort, getIpv4Address(), tcb->dstAddr);

        TCPHead* r = new TCPHead();
        do {
            res = waitIpPacket((char*)r, 5000);
        } while (res == -1);

        if (r->flag == PACKET_TYPE_ACK) {
            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;

    if (tcb->state == ESTABLISHED) {
        TCPHead * r = new TCPHead();
        do {
            res = waitIpPacket((char*)r, 5000);
        } while (res == -1);
        memcpy(pData, r->data, sizeof(r->data));
        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;

    if (tcb->state == ESTABLISHED) {
        stud_tcp_output(NULL, 0, PACKET_TYPE_FIN_ACK, tcb->srcPort,
tcb->dstPort, getIpv4Address(), tcb->dstAddr);
        tcb->state = FIN_WAIT1;
        TCPHead *r = new TCPHead();
        do {
            res = waitIpPacket((char*)r, 5000);
        } while (res == -1);

        if (r->flag == PACKET_TYPE_ACK) {
            tcb->state = FIN_WAIT2;
            do {
                res = waitIpPacket((char*)r, 5000);
            } while (res == -1);
            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;
}
```