TCP协议实验

1600013019 张智涵

一、实验内容

本实验要求模拟客户端角色的TCP协议，包括TCP报文的接收和发送，以及传输层向应用层提供的socket接口。本实验将模拟TCP连接从建立到数据传输再到关闭的全部过程，并且通过实现TCP协议的有限状态机控制TCP连接的状态。

本实验中的TCP协议为简化版的停等模式，要求客户端和服务器通过收发特殊报文的方式建立或拆除连接。客户端建立连接时需要发送SYN报文，等待服务器接收并回发SYN_ACK报文，再发送ACK报文确认收到数据，才可以建立连接。连接建立后的数据传送过程包括向服务器发送数据以及接收服务器传来的数据。拆除连接时，客户端需要主动发送FIN报文，等待服务器接收并回发ACK报文确认，此时连接进入半关闭状态。客户端需要等待服务器发送FIN报文关闭连接，并发送ACK确认，等待一段TIMEOUT时间之后连接被解除。

此外，本实验要求实现传输层向应用层提供的socket接口，包括维护传输控制块、分配套接字、控制建立和解除连接等操作，为上层应用进一步抽象连接的建立、解除和数据收发过程。

二、代码实现

TCP协议中的一对网络连接端口被抽象为套接字，而在实验中可能出现和服务器的不同端口进行连接的情况，所以我们需要为每一个连接建立一个TCB并用一个数据结构来保存所有TCB。在代码实现中，采用vector数组保存所有TCB，并建立TCB结构如下：

```
struct TCB
{
    unsigned int src_addr;
    unsigned int dst_addr;
    unsigned short src_port;
    unsigned short dst_port;
    unsigned short status;
    unsigned short src_seq;
    unsigned short dst_seq;
    int sockfd;
};
```

其中，sockfd作为套接字的标号，可作为区分TCB的属性。

当stud_tcp_socket被调用时，新建一个TCB并存入tcb_table数组中。当调用stud_tcp_connect函数建立连接时，根据sockfd从tcb_table中找到对应的TCB，先调用stud_tcp_output函数发送SYN报文，再调用waitIpPacket接口函数等待服务器发回SYN_ACK报文确认，收到确认报文之后调用stud_tcp_input函数接收报文并且发送ACK确认（在stud_tcp_input函数中实现），完成建立连接的“三次握手”过程。

当调用stud_tcp_send函数发送数据时，根据sockfd从tcb_table中找到对应的TCB，再调用stud_tcp_output发送报文，同时调用waitIpPacket接口函数等待服务器发回ACK报文确认，收到确认报文之后调用stud_tcp_input函数处理。调用stud_tcp_recv函数接收报文时，将waitIpPacket函数接收到的报文中的TCP头部剥离，并将剩余数据填入pData缓冲区中，交由上层应用处理。传输层层面，调用stud_tcp_output函数发送ACK确认报文。

当调用stud_tcp_close函数断开连接时，根据sockfd从tcb_table中找到对应的TCB，调用stud_tcp_output发送FIN报文，同时调用waitIpPacket接口函数等待服务器发回FIN_ACK报文确认，收到确认报文后交由stud_tcp_input函数处理，并继续等待服务器的下一个FIN报文，收到后调用stud_tcp_output发送ACK确认，连接断开。

下层接口函数中，stud_tcp_input函数首先根据源地址、源端口、目标地址、目标端口从tcb_table中找到这四项都匹配的TCB，用接收到的报文序列号和TCB中允许接收的序列号相比较，若不相等则丢弃；若相等，则根据当前TCB的状态和收到报文的flags更改连接状态，实现有限状态机的状态转移。如果需要回发ACK报文，则调用stud_tcp_output发送，必要时更改TCB中的seq和ack序号（src_seq和dst_seq）。

stud_tcp_output函数同样需要先找到匹配的TCB，但与上一个函数不同的是，在测试用例1（分组交互测试）中，TCB不会通过stud_tcp_socket函数创建，所以tcb_table初始为空数组。这种情况下就需要手动建立一个TCB供测试用例1完成数据收发。在测试用例2中，采用与stud_tcp_input类似的方式找到匹配的TCB。接下来函数根据不同的flags填写TCP头部，如果有数据需要发送则和数据通过memcpy连接起来。TCP协议的校验和需要通过头部、TCP伪头部和数据三部分算出，所以为了计算校验和额外定义了伪头部的数据结构fake_head，在每次计算校验和之前填充好。最后调用tcp_sendIpPkt接口函数发送报文即可。

实验代码：

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

#include "sysInclude.h"
#include <iostream>
#include <winsock.h>
#include <vector>
using namespace std;

const unsigned short CLOSED = 0;
const unsigned short SYN_SENT = 1;
const unsigned short ESTABLISHED = 2;
const unsigned short FIN_WAIT_1 = 3;
const unsigned short FIN_WAIT_2 = 4;
const unsigned short TIME_WAIT = 5;
short gSrcPort = 2005;
short gDstPort = 2006;
int global_sockfd = 0;
/*
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
*/

struct tcp_head
{
    unsigned short src_port;
    unsigned short dst_port;
    unsigned int seq;
    unsigned int ack;
    unsigned short length_flags;
    unsigned short window;
    unsigned short checksum;
    unsigned short urgent_ptr;
};

struct fake_head
{
    unsigned int src_addr;
    unsigned int dst_addr;
    unsigned short protocol;
    unsigned short length;
};

struct TCB
{
    unsigned int src_addr;
    unsigned int dst_addr;
    unsigned short src_port;
    unsigned short dst_port;
    unsigned short status;
    unsigned short src_seq;
    unsigned short dst_seq;
    int sockfd;
};

vector<TCB*> tcb_table;

unsigned short compute_checksum(unsigned short *pBuffer, int length, unsigned short
*fake_head, int fake_length)
{
    unsigned int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += ntohs(pBuffer[i]);
        sum = (sum >> 16) + (sum & 0xffff);
    }
    for (int i = 0; i < fake_length; i++) {
        sum += ntohs(fake_head[i]);
        sum = (sum >> 16) + (sum & 0xffff);
    }
    return sum;
};


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 stud_tcp_input(char *pBuffer, unsigned short len, unsigned int srcAddr, unsigned int
dstAddr)
{
    tcp_head* header = new(tcp_head);
    header = (tcp_head*)pBuffer;
    unsigned short dst_port = ntohs(header->src_port);
    unsigned short src_port = ntohs(header->dst_port);

    // find the correct tcb
    TCB* tcb = NULL;
    for (int i = 0; i < tcb_table.size(); i++) {
        if (tcb_table[i]->dst_addr == ntohl(srcAddr) && tcb_table[i]->src_addr ==
ntohl(dstAddr)
            && tcb_table[i]->src_port == src_port && tcb_table[i]->dst_port == dst_port)
            tcb = tcb_table[i];
    }
    if (tcb == NULL) return -1;

    // check the sequence number
    unsigned int seq = ntohl(header->seq);
    if (seq != tcb->dst_seq) {
        tcp_DiscardPkt(pBuffer, STUD_TCP_TEST_SEQNO_ERROR);
        return -1;
    }

    //change TCP status
    unsigned short flags = ntohs(header->length_flags) & 0x3f;
    if (tcb->status == SYN_SENT && flags == 0x12) {
        tcb->src_seq++;
        tcb->dst_seq++;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, tcb-
>src_addr, tcb->dst_addr);
        tcb->status = ESTABLISHED;
    }
    if (tcb->status == ESTABLISHED && flags == 0x10) {
        unsigned int ack = ntohl(header->ack);
        tcb->src_seq = ack;
        tcb->dst_seq = seq + 1;
    }
    if (tcb->status == FIN_WAIT_1 && flags == 0x10) {
        tcb->status = FIN_WAIT_2;
    }
    if (tcb->status == FIN_WAIT_2 && flags == 0x11) {
        tcb->src_seq++;
        tcb->dst_seq++;
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, tcb-
>src_addr, tcb->dst_addr);
        tcb->status = TIME_WAIT;
    }
    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)
{
    // for lab 1, there is no existing tcb_table
    TCB* tcb = NULL;
    if (tcb_table.size() == 0) {
        tcb = new TCB;
        tcb_table.push_back(tcb);
    }
    else {
        for (int i = 0; i < tcb_table.size(); i++) {
            if (tcb_table[i]->dst_addr == dstAddr && tcb_table[i]->src_addr == srcAddr
                && tcb_table[i]->src_port == srcPort && tcb_table[i]->dst_port == dstPort)
                tcb = tcb_table[i];
        }
    }

    if (tcb == NULL) return;

    tcp_head* header = new(tcp_head);
    header->src_port = htons(srcPort);
    header->dst_port = htons(dstPort);
    header->window = htons(1);
    fake_head* fake_header = new(fake_head);
    fake_header->src_addr = htonl(srcAddr);
    fake_header->dst_addr = htonl(dstAddr);
    fake_header->protocol = htons(6);
    fake_header->length = htons(20+len);

    if (flag == PACKET_TYPE_SYN) {
        header->seq = htonl(1);
        header->ack = htonl(0);
        header->checksum = htons(0);
        header->length_flags = htons(0x5002);
        header->checksum = htons(~compute_checksum((unsigned short *)header, 10, (unsigned
short *)fake_header, 6));
        tcp_sendIpPkt((unsigned char*)header, len + 20, srcAddr, dstAddr, 255);

        tcb->dst_addr = dstAddr;
        tcb->dst_port = dstPort;
        tcb->src_addr = srcAddr;
        tcb->src_port = srcPort;
        tcb->src_seq = 1;
        tcb->dst_seq = 1;
        tcb->status = SYN_SENT;
    }
    if (flag == PACKET_TYPE_ACK) {
        header->seq = htonl(tcb->src_seq);
        header->ack = htonl(tcb->dst_seq);
        header->checksum = htons(0);
        header->length_flags = htons(0x5010);
        header->checksum = htons(~compute_checksum((unsigned short *)header, 10, (unsigned
short *)fake_header, 6));
        tcp_sendIpPkt((unsigned char*)header, len + 20, srcAddr, dstAddr, 255);
    }
    if (flag == PACKET_TYPE_FIN_ACK) {
        header->seq = htonl(tcb->src_seq);
        header->ack = htonl(tcb->dst_seq);
        header->checksum = htons(0);
        header->length_flags = htons(0x5011);
        header->checksum = htons(~compute_checksum((unsigned short *)header, 10, (unsigned
short *)fake_header, 6));
        tcp_sendIpPkt((unsigned char*)header, len + 20, srcAddr, dstAddr, 255);

        if (tcb->status == ESTABLISHED)
            tcb->status = FIN_WAIT_1;
    }
    if (flag == PACKET_TYPE_DATA) {
        header->seq = htonl(tcb->src_seq);
        header->ack = htonl(tcb->dst_seq);
        header->checksum = htons(0);
        header->length_flags = htons(0x5000);
        memcpy((char *)header + 20, pData, len);
        header->checksum = htons(~compute_checksum((unsigned short *)header, 10 + len/2,
(unsigned short *)fake_header, 6));
        tcp_sendIpPkt((unsigned char*)header, len + 20, srcAddr, dstAddr, 255);
    }
}

int stud_tcp_socket(int domain, int type, int protocol)
{
    global_sockfd++;
    TCB *tcb = new TCB;
    tcb->sockfd = global_sockfd;
    tcb_table.push_back(tcb);
    return global_sockfd;
}

int stud_tcp_connect(int sockfd, struct sockaddr_in *addr, int addrlen)
{
    unsigned int src_addr = getIpv4Address();
    unsigned int dst_addr = ntohl(addr->sin_addr.s_addr);
    unsigned short dst_port = ntohs(addr->sin_port);

    // find the correct socket
    TCB *tcb = NULL;
    for (int i = 0; i < tcb_table.size(); i++) {
        if (tcb_table[i]->sockfd == sockfd)
            tcb = tcb_table[i];
    }
    tcb->src_addr = src_addr;
    tcb->dst_addr = dst_addr;
    tcb->src_port = gSrcPort;
    tcb->dst_port = dst_port;

    stud_tcp_output(NULL, 0, PACKET_TYPE_SYN, gSrcPort, dst_port, src_addr, dst_addr);
    char* pBuffer = new char[100];
    int length = waitIpPacket(pBuffer, 255);
    stud_tcp_input(pBuffer, length, htonl(dst_addr), htonl(src_addr));
    return 0;
}

int stud_tcp_send(int sockfd, const unsigned char *pData, unsigned short datalen, int flags)
{
    // find the correct socket
    TCB *tcb = NULL;
    for (int i = 0; i < tcb_table.size(); i++) {
        if (tcb_table[i]->sockfd == sockfd)
            tcb = tcb_table[i];
    }

    stud_tcp_output((char*)pData, datalen, PACKET_TYPE_DATA, tcb->src_port, tcb->dst_port,
tcb->src_addr, tcb->dst_addr);
    char* pBuffer = new char[100];
    int length = waitIpPacket(pBuffer, 255);
    if (length == -1) return -1;
    stud_tcp_input(pBuffer, length, htonl(tcb->dst_addr), htonl(tcb->src_addr));

    return 0;
}

int stud_tcp_recv(int sockfd, unsigned char *pData, unsigned short datalen, int flags)
```

```
{
    TCB *tcb = NULL;
    for (int i = 0; i < tcb_table.size(); i++) {
        if (tcb_table[i]->sockfd == sockfd)
            tcb = tcb_table[i];
    }
    if (tcb->status != ESTABLISHED)
        return -1;

    char* pBuffer = new char[100];
    int length = waitIpPacket(pBuffer, 255);
    if (length == -1) return -1;

    tcp_head *header = new tcp_head;
    memcpy((char*)header, pBuffer, 20);
    memcpy(pData, (unsigned char *)pBuffer + 20, length - 20);
    tcb->src_seq = ntohl(header->ack);
    tcb->dst_seq = ntohl(header->seq) + length - 20;
    stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, tcb->src_addr,
tcb->dst_addr);

    return 0;
}

int stud_tcp_close(int sockfd)
{
    TCB *tcb = NULL;
    for (int i = 0; i < tcb_table.size(); i++) {
        if (tcb_table[i]->sockfd == sockfd)
            tcb = tcb_table[i];
    }

    // send FIN
    stud_tcp_output(NULL, 0, PACKET_TYPE_FIN_ACK, tcb->src_port, tcb->dst_port, tcb-
>src_addr, tcb->dst_addr);
    tcb->src_seq++;
    tcb->dst_seq++;

    // wait for ACK
    char* pBuffer = new char[100];
    int length = waitIpPacket(pBuffer, 255);
    if (length == -1) return -1;
    stud_tcp_input(pBuffer, length, htonl(tcb->dst_addr), htonl(tcb->src_addr));

    // wait for FIN
    length = waitIpPacket(pBuffer, 255);
    if (length == -1) return -1;

    // send ACK
    stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, tcb->src_addr,
tcb->dst_addr);
    return 0;
}
```