《计算机网络协议开发》实验报告

第 11 次实验

TCP 协议实验

姓名：元玉慧

学号：101220151

10 级 计算机 系 4 班

邮箱：njucsyyh@gmail.com

时间：2013/05/10

一、 实验目的

通过本实验可以让学生深入理解，TCP 协议的原理，掌握连接状态控制，可靠传输等重要机制，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 TCP 协议还要向应用层提供编程接口，即网络编程中常用的 Scoket 接口，通过这些接口函数，可以加深理解网络编程的原理，提高网络程序设计和调试能力。

二、实验设计背景

从设计角度分析，TCP 协议主要要考虑如下因素：

-1-状态控制

-2-滑动窗口机制

-3-拥塞控制算法

-4-性能问题

-5-Socket 接口

我们要实现的是对上述功能的一个简化，仅实现客户端角色的，“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 段，并向应用层提供客户端需要的 Socket 接口。

简化的 TCP 协议客户端：

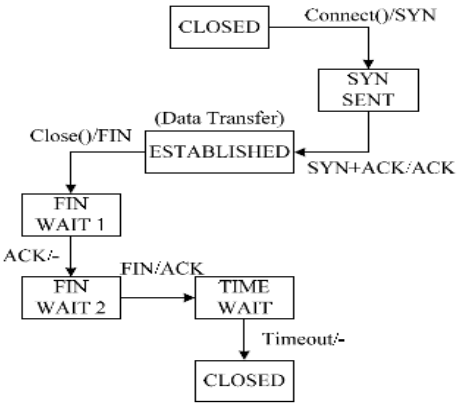
客户端角色的 TCP 只能主动的发起建立连接的请求，而不能监听端口等待其他主机的连接，因而相比于 TCP 协议的状态控制，客户端角色的 TCP 协议得到了简化；

三、实验背景

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提供端到端的、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，提供了可靠的、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、TELNET 等。

简化的 TCP 协议客户端

客户端角色的 TCP 协议只能够主动发起建立连接请求，而不能监听端口等待其它主机的连接。因而相比于标准 TCP 协议的状态控制，客户端角色的 TCP 协议的状态机得到简化，如下图所示：



从 CLOSED 状态开始，调用 Socket 函数 stud_tcp_connect ( )发送建立连接请求（SYN 报文）

-----→ 转入 SYN_SENT 状态，等待服务器端的应答和建立连接请求；

-----→收到服务器端的 SYN+ACK 后，以 ACK 应答，完成“三次握手”的过程，转为 ESTABLISHED 状态

(在 ESTABLISHED 状态，客户端与服务器端可以通过函数 stud_tcp_recv ( )和 stud_tcp_send ( )进行数据交互)

-----→ 完成数据传输后，客户端通过 stud_tcp_close ( )函数发送关闭连接请求（FIN 报文），转入 FIN-WAIT 1 状态

-----→ 收到应答 ACK 后进入 FIN-WAIT 2 状态，此时状态为半关闭，等待服务器端关闭连接

-----→ 收到服务器端的 FIN 报文后，需要发送确认 ACK，状态改变为 TIME-WAIT;

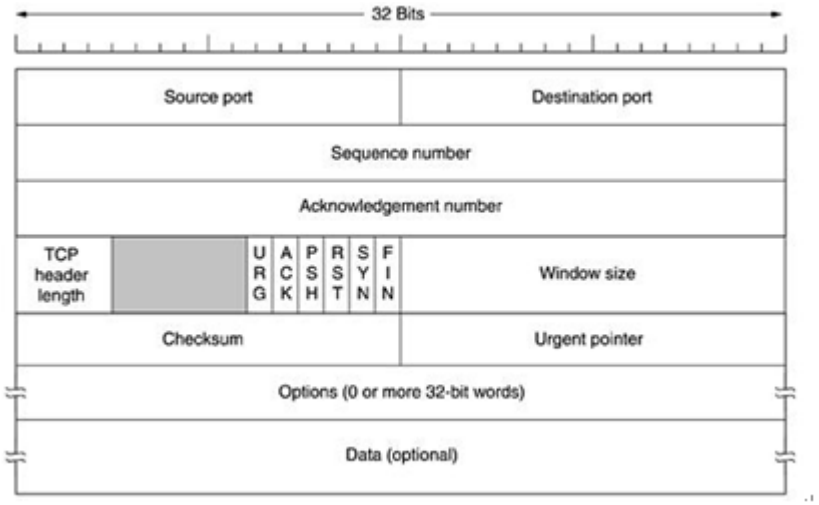
-----→ 等待一段时间后，转为初始的 CLOSED 状态，连接完全断开。

上述流程是一个客户端 TCP 建立和拆除连接的最简单的正常流程，然而实际的 TCP 协议要处理各种各样的异常情况，这也是 TCP 状态控制非常复杂的一个原因。本实验中对此不做复杂处理，对接收到的异常报文（如报文类型不符、序号错误等情况）调用系统所提供的函数 tcp_DiscardPkt( )丢弃即可。

“停 – 等”模式的 TCP 协议，其发送窗口和接收窗口大小都为 1。接收到的报文中，只有序列号正确的报文才会被处理，并根据报文内容和长度对接收窗口进行滑动。发送一个报文后就不能继续发送，而要等待对方对此报文的确认，收到确认后才能继续发送。如果一段时间后没有收到确认，则需要重传

实现一个完整的 TCP 协议，还需要向应用层提供 Socket 接口函数。作为客户端角色的 TCP 协议，必须实现的 Socket 接口函数包括：stud_tcp_socket( )、stud_tcp_connect ( )、stud_tcp_recv ( )、stud_tcp_send ( )和 stud_tcp_close ( )。实现 Socket 接口函数，主要工作是将 Socket 接口函数和 TCP 报文的接收和发送流程结合起来

TCP 头部报文格式



Source port：源端口号
Destination port：目的端口号
Sequence number：发送端序列号，用于标识从 TCP 发送端向接收端发送的数据字节流
Acknowledgement number：确认序列号，一个 TCP 客户端发起连接请求时，ack_seq 的值为 0
TCP header length：首部长度，其单位并非字节，而是 32bit
urg, ack, psh, rst, syn, fin 为 6 个标志位:

urg: 紧急指针有效
ack: 确认序号有效
psh: 接收方应该尽快将这个报文段交给应用层
rst: 复位
syn: 同步序号，用来发起一个连接
fin: 发送端完成发送任务。

Window size：16 位滑动窗口的大小，单位为字节，这个值是接收端正期望接收的字节数，最大 63353
Checksum：检验和
Urgent pointer：紧急指针，和序列号字段相加表示紧急数据最后一个字节的序列号

数据结构
TCB 数据结构来存储每一个 TCP 连接的发送和接收，TCP 为每一个活动维护了一个 TCP 结构，TCB 中包含了 2 端的 IP 地址和端口号，连接状态信息，发送和接受窗口信息等。

IPv4 报文协议的 TCP 校验和计算方法：（参考 RFC 793）

TCP pseudo-header for checksum computation (IPv4)

Bit offset	0－3	4－7	8－15	16－31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			
160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

三、实验内容

设计 TCB 数据结构：

TCP 协议的接收处理：

需要实现 stud_tcp_input() 函数，完成校验和检查，字节序转换，重点实现客户端角色的 TCP 段接收的有限状态机，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。

TCP 协议的封装：

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，要在收到对上一段的确认后才能继续发送。

TCP 协议的提供的 Socket 接口函数：

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 (), stud_tcp_close(), 将接口函数实现的内容和接收流程有机的结合起来。

测试 1：针对段交互的测试；

测试 2：socketAPI 测试；

接口函数说明以及实现：

TCP 段输入函数： 本函数完成下列段接收处理步骤

-1- 检查校验和字段

-2- 字节序转换

-3- 检查序号

-4- 将段交由输入有限状态机处理

-5- 有限状态机对端进行处理，转换状态

-6- 根据当前状态，调用 tcp_sendIpPkt()完成 TCP 连接的建立，返回 ACK, TCP, 释放连接等工作。

实现关键代码：

```
tcp_check_buf *buffer = new tcp_check_buf;
buffer->pheader = *p_header;
buffer->theader = *temp_tcp;

if(checksum((unsigned short*)buffer, sizeof(*buffer)) != 0){
    printf("warning: checksum is wrong ...\n");
    return -1;
}

tcb_now = get_tcb_port(ntohs(temp_tcp->dst_port), ntohs(temp_tcp->src_port));

return FSM(0, temp_tcp, tcb_now);
```

TCP 段输出函数：此函数中需要自行申请空间，封装 TCP 段首部和相关的数据，此函数由接收函数调用，也可以直接由解析器调用，此函数中将调用系统函数 tcp_sendIpPkt()完成段发送。

-1-flag 字段可以是如下的值：(略去)

-2-判断需要发送的段的类型，并针对不同的特定类型做相应的处理；

-3-判断是否可以发送

-4-构造 TCP 数据段并发送

-5- 填写 TCP 段中各个字段的内容和数据，转换字节序，计算校验和，然后调用发送流程函数；

实现关键代码：

【1】获取当前 TCP 连接的状态信息：

```
if( (tcb_now = get_tcb_port(srcPort, dstPort)) == NULL){
    int sockfd = stud_tcp_socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    tcb_now = get_tcb_sockfd(sockfd);
    tcb_now->src_addr = srcAddr;
    tcb_now->dst_addr = dstAddr;
    tcb_now->src_port = srcPort;
    tcb_now->dst_port = dstPort;
}
```

【2】判断消息的窗口大小是否正确

```
if(tcb_now->window_size <= 0) return;
else{
    //pseudo TCP header
    pseudo_head* p_head = new pseudo_head;
    p_head->src_addr=ntohl(srcAddr);
    p_head->dst_addr=ntohl(dstAddr);
    p_head->mbz=0;
    p_head->protocol = IPPROTO_TCP;
    p_head->tcplen = ntohs(20+len);
    //TCP
    p_tcphead tcphdr= new tcphead;
    memset(tcphdr, 0, sizeof(tcphead));
    for(int i=0; i<len; i++)
        tcphdr->data[i] = pData[i];
```

```
tcphdr->src_port = ntohs(srcPort);
tcphdr->dst_port = ntohs(dstPort);
tcphdr->seq = ntohl(tcb_now->seq);
tcphdr->ack = ntohl(tcb_now->ack);
tcphdr->THL = 0x50;
tcphdr->flags = flag;
tcphdr->window_size = ntohs(30);
tcphdr->urgent_pointer = ntohs(0);

tcp_check_buf* check_buf = new tcp_check_buf;
check_buf->pheader = *p_head;
check_buf->theader = *tcphdr;

tcphdr->check_sum = checksum((unsigned short*)check_buf, sizeof(tcp_check_buf));
//转换状态字段，修改TCB中的记录信息
FSM(flag, NULL, tcb_now);
//发送tcp报文
tcp_sendIpPkt((unsigned char*)tcphdr, 20+len, srcAddr, dstAddr, 50);//
```

关于 tcphdr->THL 赋值为 0x50 是因为包括了首部长度字段和保留字段，首部长度字段是（5*4）20，保留字段是 0；设置滑动窗口的大小为 30；调用 tcp_sendIpPkt()函数中设置最大条数 ttl 为 50；

获得 socket 描述符函数

本函数在实现是要完成以下功能：创建新的 TCB 结构；并对成员变量进行初始化；为该 TCB 结构分配唯一的套接字描述符；

函数体中并不需要用到形参，不知道这 3 个参数在这里有何用?

```
int stud_tcp_socket(int domain, int type, int protocol)
{
    tcb_now = new tcb;
    //赋值得到唯一的socket描述符
    tcb_now->sockfd = socknum++;
    //初始状态是 CLOSED
    tcb_now->state = CLOSED;
    //分配唯一的端口号
    tcb_now->src_port = gSrcPort++;
    tcb_now->dst_port = gDstPort++;
    //获取发送的源地址和目标地址
    tcb_now->src_addr = getIpv4Address();
    tcb_now->dst_addr = getServerIpv4Address();
    //
    tcb_now->seq = gSeqNum++;
    tcb_now->ack = gAckNum;
    //停等协议窗口大小为1
    tcb_now->window_size = 1;
    //从头部插入tcb节点
    tcbnode* temp = new tcbnode;
    temp->tcb = tcb_now;
    temp->next = tcb_list;
    tcb_list = temp;
    return tcb_now->sockfd;
}
1
```

TCB 建立连接函数

本函数中要求发送 SYN 段，调用系统函数 waitIpPacket()获得 SYN_ACK 段，发送 ACK 段，直接建立 TCP 连接

需要完成的步骤有：

-1-设定目的/源 IP 地址和端口

-2-设定 TCB 中的输入状态为 SYN_SENT, 及其他相关变量，准备发送 SYN 段

-3-调用发送流程下层接口函数， stud_tcp_output()发送 SYN 段（发送类型为 PACKET_TYPE_SYN）

-4-等待 3 次握手完成之后，返回或者返回出错

```
int stud_tcp_connect(int sockfd, struct sockaddr_in *addr, int addrlen)
{

    if( (tcb_now = get_tcb_sockfd(sockfd)) != NULL){
        //设定目的IP和端口号
        tcb_now->dst_addr = ntohl(addr->sin_addr.s_addr);
        tcb_now->dst_port = ntohs(addr->sin_port);
        //第一次握手
        stud_tcp_output(NULL,0,PACKET_TYPE_SYN, tcb_now->src_port,
                tcb_now->dst_port, tcb_now->src_addr, tcb_now->dst_addr);
        int len = 0;
        p_tcphead tcphdr = new tcphead;
        //接收到的TCP头部长度信息为20，否则需要再次主动接受返回的数据段
        //第二次握手
        while(len < 20){
            len = waitIpPacket((char*)tcphdr, 500);
        }
        //收到确认信息之后，
        //在调用FSM()函数中会进行第三次握手
        return stud_tcp_input((char*)tcphdr, len,
                addr->sin_addr.s_addr,htonl(getIpv4Address()));
        //完成了3此握手！！！
    }
    else{
        printf("NO EXIST ...\n");
        return -1;
    }
    return 0;
}
```

TCP 段发送函数

-1-判断是否处于 ESTABLISHED 状态；

-2-将应用层协议的数据复制到 TCB 的输入缓冲区；

-3-调整 stud_tcp_output()发送 TCP 数据段；（发送类型为 PACKET_TYPE_DATA）

```
int stud_tcp_send(int sockfd, const unsigned char *pData, unsigned short datalen, int flags)
{
    if( (tcb_now = get_tcb_sockfd(sockfd)) == NULL){
        printf("NO EXIST ...\n");
        return -1;
    }
    if(tcb_now->state == ESTABLISHED){
        stud_tcp_output((char *)pData, datalen, PACKET_TYPE_DATA, tcb_now->src_port,
                tcb_now->dst_port, tcb_now->src_addr, tcb_now->dst_addr);
        p_tcphead tcphdr = new tcphead;
        //每次发送报文都要等待接收到ACK才可以发送下一个报文段
        //即满足了停等协议的要求
        int len = 0;
        while(len < 20){
            len = waitIpPacket((char*)tcphdr, 500);
        }
        //判断收到的数据报是是ACK还是RST或者其他信息
        if(tcphdr->flags == PACKET_TYPE_ACK){
            if(ntohl(tcphdr->ack) != (tcb_now->seq+datalen)){
                printf("SEQ NUMBER ERROR ...\n");
                tcp_DiscardPkt((char *)tcphdr, STUD_TCP_TEST_SEQNO_ERROR);
                return -1;
            }
            tcb_now->seq = ntohl(tcphdr->ack);
            tcb_now->ack = ntohl(tcphdr->seq) + len-20;
        }
        return 0;
    }

    return -1;
}
```

TCP 段接收函数

判断是否处于 ESTABLISHED 状态，从 TCB 输入缓冲区中读出数据，将数据交给应用层协议，即将数据拷贝到参数 pData 中。（参数 datalen 和 flags 是用不到的）

```
/*
 * TCP段接收函数
 * 接收来自服务器的数据
 * 并回复确认的ACK
 */
int stud_tcp_recv(int sockfd, unsigned char *pData, unsigned short datalen, int flags)
{
    if( (tcb_now = get_tcb_sockfd(sockfd)) == NULL){
        printf("warning: NO EXIST ...\n");
        return -1;
    }
    if(tcb_now->state == ESTABLISHED){
        //等待接收发送的报文
        int len = 0;
        p_tcphead tcphdr = new tcphead;
        while(len < 20){
            len = waitIpPacket((char*)tcphdr, 500);
        }
        tcb_now->ack = ntohl(tcphdr->seq)+len-20;
        memcpy(pData, (unsigned char*)tcphdr+20, len-20);
        //发送ACK给服务器，确认收到了服务器发来的数据
        stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb_now->src_port,
                tcb_now->dst_port, tcb_now->src_addr, tcb_now->dst_addr);

        return 0;
    }
    return -1;
}
```

TCP 关闭连接函数

在正常情况下，进行相应的状态转换，在非正常情况下直接删除 TCB 结构并退出，调用发送流程下层接口函数，发送 FIN 段，等待回应的 ACK 段和服务器发送的 FIN 段，收到后返回 ACK 段，删除对应的 TCB 数据结构；

```
int stud_tcp_close(int sockfd)
{
    if( (tcb_now = get_tcb_sockfd(sockfd)) == NULL ){
        printf("NO EXIST ...\n");
        return -1;
    }
    //只有在ESTABLISHED 状态下才可以关闭连接
    if(tcb_now->state == ESTABLISHED){
        //第一次握手，发送关闭连接的报文,状态变为 FIN_WAIT1
        stud_tcp_output(NULL, 0, PACKET_TYPE_FIN_ACK, tcb_now->src_port,
                tcb_now->dst_port, tcb_now->src_addr, tcb_now->dst_addr);
        //第二次握手，等待接收其发来ACK
        int len;
        p_tcphead tcphdr = new tcphead;
        while(len < 20){
            len = waitIpPacket((char*)tcphdr, 500);
        }
        //接收处理ACK报文，有限状态机切换到 FIN_WAIT2
        stud_tcp_input((char *)tcphdr, len, htonl(tcb_now->dst_addr),
                htonl(tcb_now->src_addr));
        //等待接收服务器端的FIN报文
        len = 0;
        while(len < 20){
            len = waitIpPacket((char*)tcphdr, 500);
        }
        //处理接收到的FIN报文，状态变为 CLOSED 并发送ACK报文
        stud_tcp_input((char *)tcphdr, len, htonl(tcb_now->dst_addr),
                htonl(tcb_now->src_addr));

     tcbnode *p = tcb_list;
     tcbnode *q = tcb_list->next;
     while(q != NULL){
         if(q->tcb->sockfd == sockfd){
             p->next = q->next;
             delete q;
             break;
         }
         p = q;
         q = q->next;
     }
     return 0;
 }
    else{
        tcbnode *p = tcb_list;
        tcbnode *q = tcb_list->next;
        while(q != NULL){
            if(q->tcb->sockfd == sockfd){
                p->next = q->next;
                delete q;
                break;
            }
            p = q;
            q = q->next;
        }
    }
    return -1;
}
```

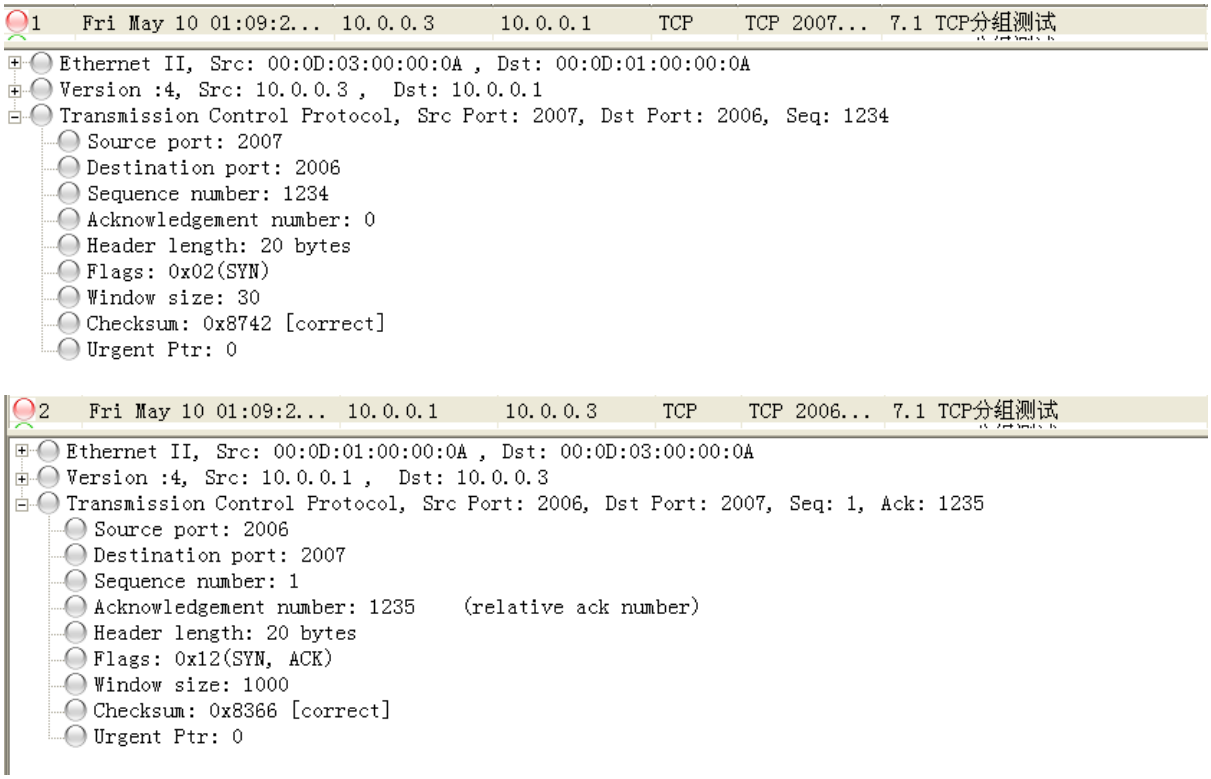
有限状态机的实现

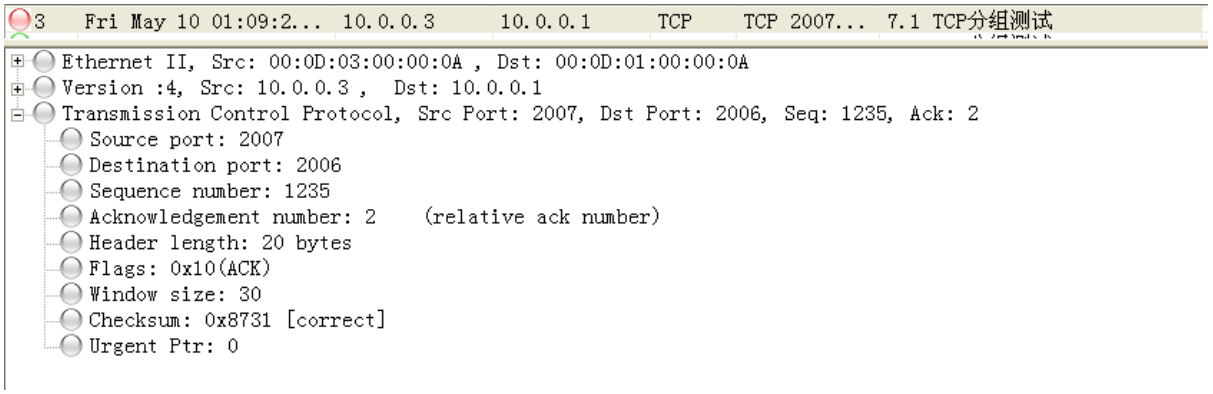
```
int FSM(unsigned char flag, p_tcphead tcp, p_tcb tcb){
      unsigned char flags;
     if(tcp != NULL)   flags = tcp->flags;
     char state = tcb->state;
     printf("The FSM state is %d...\n ", tcb->state);
     switch(state)
     {
     case CLOSED:{
          if( (flag & 0x02) == 2)   // SYN
          {
               tcb->state = SYN_SENT;
               printf("   CLOSED  --------->  SYN_SENT\n");
               return 0;
          }
         return -1;
     }

     case SYN_SENT:{
          if( (flags & 0x12) == 18 ){  //SYN ACK
               if( ntohl(tcp->ack) != tcb->seq+1){
                    printf("SEQ number error ...\n");
                    tcp_DiscardPkt((char *)tcp, STUD_TCP_TEST_SEQNO_ERROR);
                    return -1;
               }
               tcb->state = ESTABLISHED;
               tcb->ack = ntohl(tcp->seq)+1;
               tcb->seq = ntohl(tcp->ack);
               printf("   SYN_SENT  -------->  ESTABLISHED ...\n");
               stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, getIpv4Address(),
getServerIpv4Address());
               return 0;
          }
           return -1;
     }

     case ESTABLISHED:{
          if((flag & 0x01) == 1)   //FIN
          {
               tcb->state = FIN_WAIT1;
               printf("   ESTABLISHED  ------>  FIN-WAIT1\n");
               return 0;
          }
          return -1;
     }

     case FIN_WAIT1:{
          if( (flags & 0x10) == 16){  // ACK
               if( ntohl(tcp->ack) != tcb->seq+1){
                    printf("SEQ number error ...\n");
                    tcp_DiscardPkt((char *)tcp, STUD_TCP_TEST_SEQNO_ERROR);
                    return -1;
               }
               tcb->state = FIN_WAIT2;
               tcb->ack = ntohl(tcp->seq)+1;
               printf("   FIN_WAIT1  ------>  FIN_WAIT2 ...\n");
               return 0;
          }
          return -1;
     }

     case FIN_WAIT2:{
```

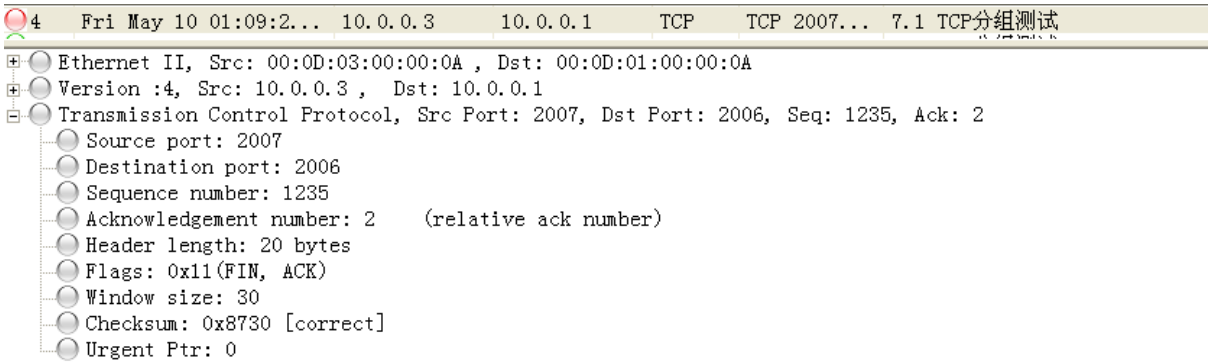
```
        if ((flags & 0x01) == 1)      //FIN
        {
             if (ntohl(tcp->ack) != tcb->seq+1)
             {
                  printf("SEQ number error.\n");
                  tcp_DiscardPkt((char *)tcp, STUD_TCP_TEST_SEQNO_ERROR);
                  return -1;
             }
             tcb->seq = ntohl(tcp->ack);
             tcb->ack = ntohl(tcp->seq) + 1;
             printf("   FIN_WAIT2  ------>  TIME_WAIT\n");
             tcb->state = TIME_WAIT;
             stud_tcp_output(NULL, 0, PACKET_TYPE_ACK, tcb->src_port, tcb->dst_port, getIpv4Address(),
getServerIpv4Address());
             return 0;
        }
        return -1;
    }

    default: return -1;
    }
}
```

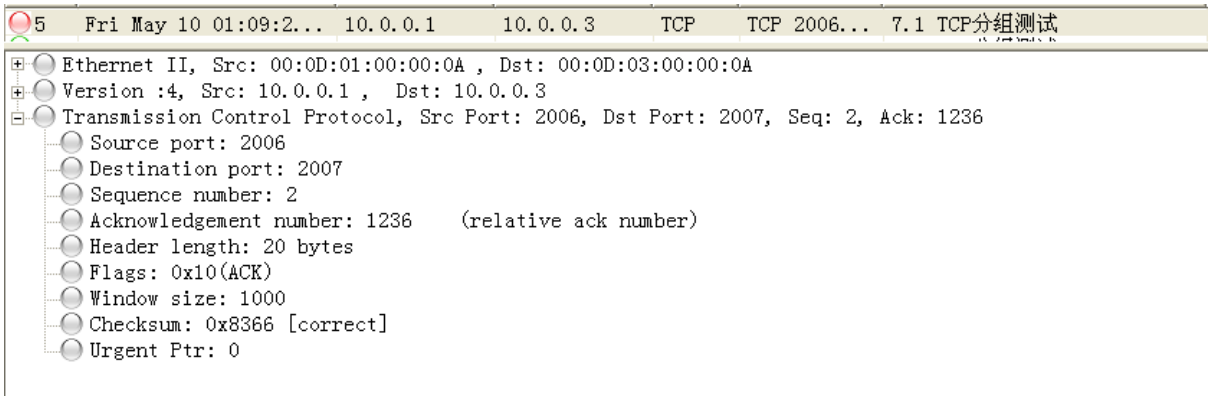
四、实验结果及报文分析

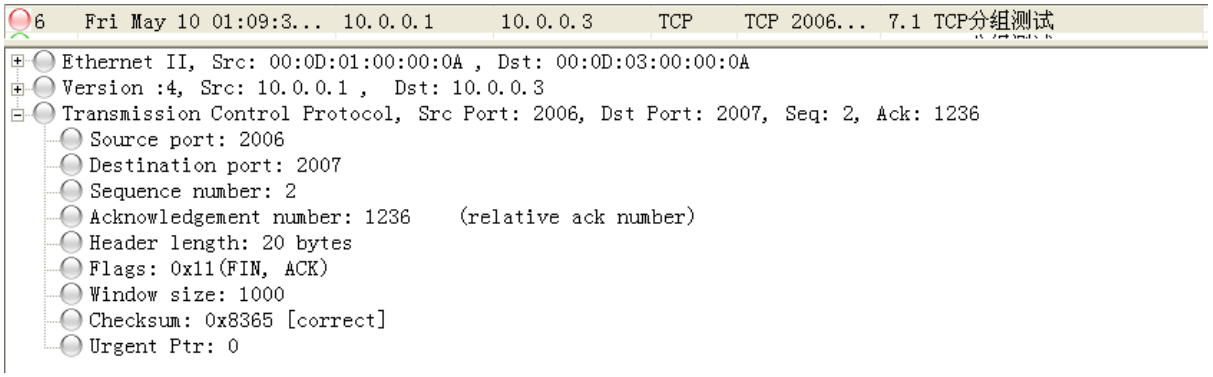
TCP 分组测试

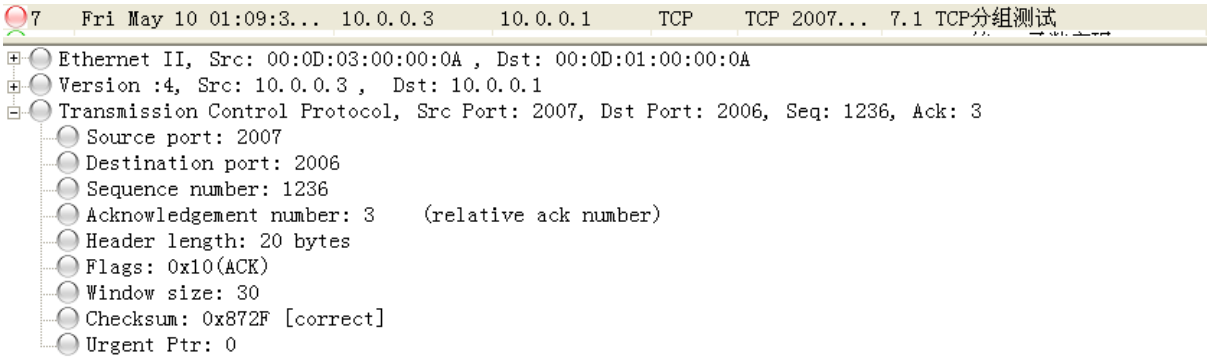



3 Fri May 10 01:09:2... 10.0.0.3 10.0.0.1 TCP TCP 2007... 7.1 TCP分组测试
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2007, Dst Port: 2006, Seq: 1235, Ack: 2
Source port: 2007
Destination port: 2006
Sequence number: 1235
Acknowledgement number: 2 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 30
Checksum: 0x8731 [correct]
Urgent Ptr: 0


4 Fri May 10 01:09:2... 10.0.0.3 10.0.0.1 TCP TCP 2007... 7.1 TCP分组测试
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2007, Dst Port: 2006, Seq: 1235, Ack: 2
Source port: 2007
Destination port: 2006
Sequence number: 1235
Acknowledgement number: 2 (relative ack number)
Header length: 20 bytes
Flags: 0x11(FIN, ACK)
Window size: 30
Checksum: 0x8730 [correct]
Urgent Ptr: 0

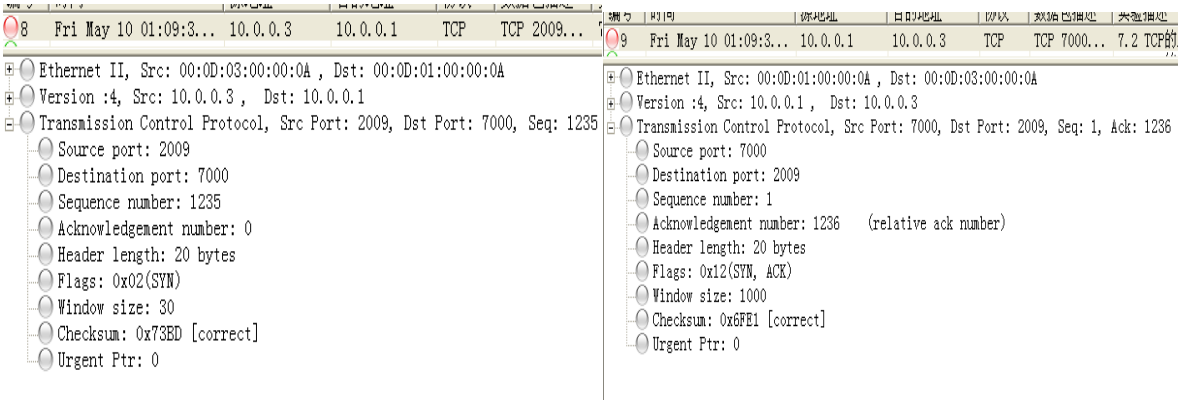

5 Fri May 10 01:09:2... 10.0.0.1 10.0.0.3 TCP TCP 2006... 7.1 TCP分组测试
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 2006, Dst Port: 2007, Seq: 2, Ack: 1236
Source port: 2006
Destination port: 2007
Sequence number: 2
Acknowledgement number: 1236 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 1000
Checksum: 0x8366 [correct]
Urgent Ptr: 0

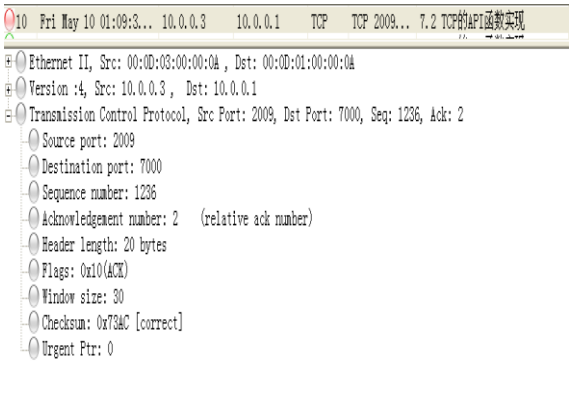

6 Fri May 10 01:09:3... 10.0.0.1 10.0.0.3 TCP TCP 2006... 7.1 TCP分组测试
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 2006, Dst Port: 2007, Seq: 2, Ack: 1236
Source port: 2006
Destination port: 2007
Sequence number: 2
Acknowledgement number: 1236 (relative ack number)
Header length: 20 bytes
Flags: 0x11(FIN, ACK)
Window size: 1000
Checksum: 0x8365 [correct]
Urgent Ptr: 0

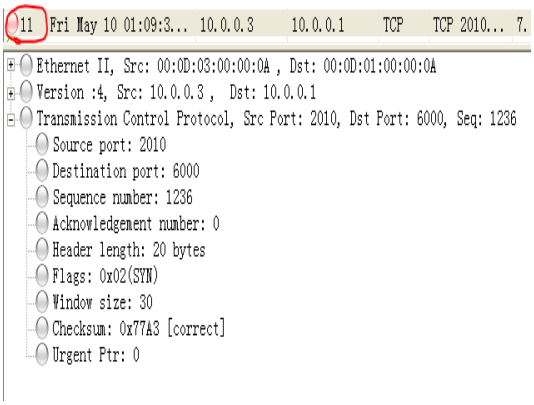


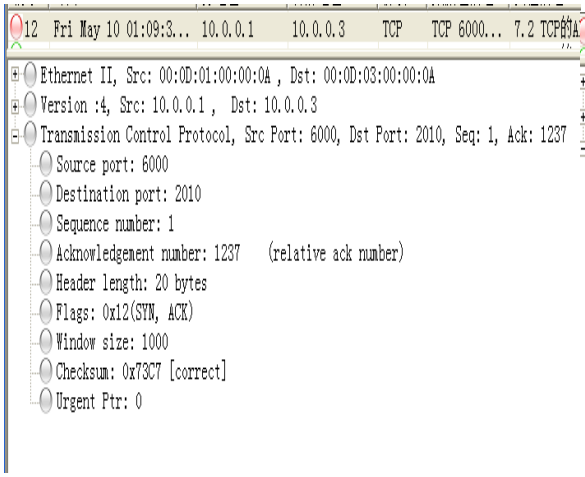
NO	Src_port	Dst_port	seq	ack	flags	描述
1	2007	2006	1234	0	SYN	本地请求连接，第一次握手
2	2006	2007	1	1235	SYN+ACK	对方对于建立连接的恢复，第二次握手
3	2007	2006	1235	2	ACK	本地返回建立连接的确认，第三次握手
4	2007	2006	1235	2	FIN+ACK	本地请求关闭连接
5	2006	2007	2	1236	ACK	服务器响应 2 关闭连接请求
6	2006	2007	2	1236	FIN+ACK	服务器处理完数据，指示本地关闭连接
7	**2007**	**2006**	**1236**	**3**	**ACK**	**本地接收指示，返回确认**

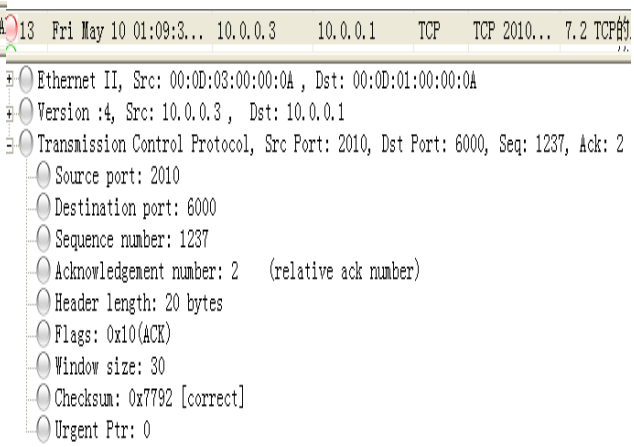
TCP API 实现

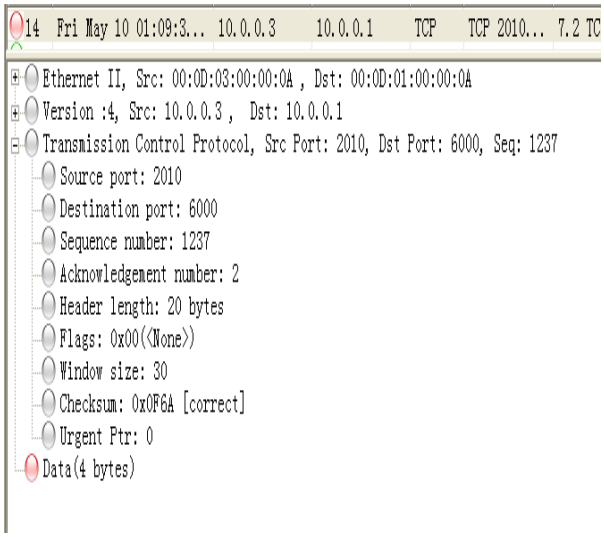


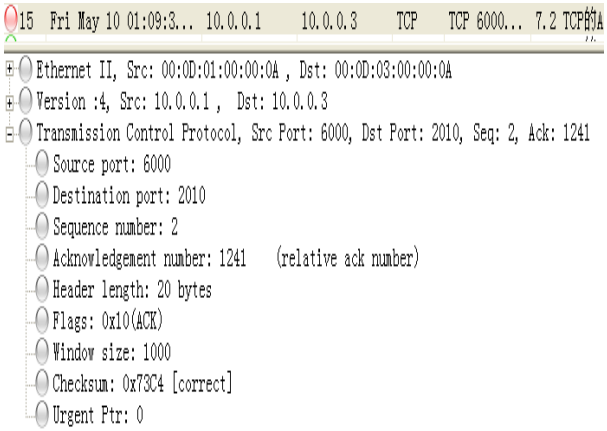

10 Fri May 10 01:09:3... 10.0.0.3 10.0.0.1 TCP TCP 2009... 7.2 TCP的API函数实现
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2009, Dst Port: 7000, Seq: 1236, Ack: 2
Source port: 2009
Destination port: 7000
Sequence number: 1236
Acknowledgement number: 2 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 30
Checksum: 0x73AC [correct]
Urgent Ptr: 0


11 Fri May 10 01:09:3... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1236
Source port: 2010
Destination port: 6000
Sequence number: 1236
Acknowledgement number: 0
Header length: 20 bytes
Flags: 0x02(SYN)
Window size: 30
Checksum: 0x77A3 [correct]
Urgent Ptr: 0


12 Fri May 10 01:09:3... 10.0.0.1 10.0.0.3 TCP TCP 6000... 7.2 TCP的A
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 6000, Dst Port: 2010, Seq: 1, Ack: 1237
Source port: 6000
Destination port: 2010
Sequence number: 1
Acknowledgement number: 1237 (relative ack number)
Header length: 20 bytes
Flags: 0x12(SYN, ACK)
Window size: 1000
Checksum: 0x73C7 [correct]
Urgent Ptr: 0


13 Fri May 10 01:09:3... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.2 TCP的
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1237, Ack: 2
Source port: 2010
Destination port: 6000
Sequence number: 1237
Acknowledgement number: 2 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 30
Checksum: 0x7792 [correct]
Urgent Ptr: 0


14 Fri May 10 01:09:3... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.2 TC
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1237
Source port: 2010
Destination port: 6000
Sequence number: 1237
Acknowledgement number: 2
Header length: 20 bytes
Flags: 0x00(<None>)
Window size: 30
Checksum: 0x0F6A [correct]
Urgent Ptr: 0
Data(4 bytes)


15 Fri May 10 01:09:3... 10.0.0.1 10.0.0.3 TCP TCP 6000... 7.2 TCP的A
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 6000, Dst Port: 2010, Seq: 2, Ack: 1241
Source port: 6000
Destination port: 2010
Sequence number: 2
Acknowledgement number: 1241 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 1000
Checksum: 0x73C4 [correct]
Urgent Ptr: 0

16 Fri May 10 01:09:3... 10.0.0.3 10.0.0.1 TCP TCP 2009... 7.2

- Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
- Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
- Transmission Control Protocol, Src Port: 2009, Dst Port: 7000, Seq: 1236
  - Source port: 2009
  - Destination port: 7000
  - Sequence number: 1236
  - Acknowledgement number: 2
  - Header length: 20 bytes
  - Flags: 0x00(<None>)
  - Window size: 30
  - Checksum: 0x0B84 [correct]
  - Urgent Ptr: 0
- Data(4 bytes)

17 Fri May 10 01:09:4... 10.0.0.1 10.0.0.3 TCP TCP 7000... 7.2 TCP

- Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
- Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
- Transmission Control Protocol, Src Port: 7000, Dst Port: 2009, Seq: 2, Ack: 1240
  - Source port: 7000
  - Destination port: 2009
  - Sequence number: 2
  - Acknowledgement number: 1240 (relative ack number)
  - Header length: 20 bytes
  - Flags: 0x10(ACK)
  - Window size: 1000
  - Checksum: 0x6FDE [correct]
  - Urgent Ptr: 0

18 Fri May 10 01:09:4... 10.0.0.1 10.0.0.3 TCP TCP 7000... 7.2 TCP

- Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
- Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
- Transmission Control Protocol, Src Port: 7000, Dst Port: 2009, Seq: 2, Ack: 1240
  - Source port: 7000
  - Destination port: 2009
  - Sequence number: 2
  - Acknowledgement number: 1240 (relative ack number)
  - Header length: 20 bytes
  - Flags: 0x10(ACK)
  - Window size: 1000
  - Checksum: 0x07A6 [correct]
  - Urgent Ptr: 0
- Data(4 bytes)

19 Fri May 10 01:09:4... 10.0.0.3 10.0.0.1 TCP TCP 2009... 7.2 TCP

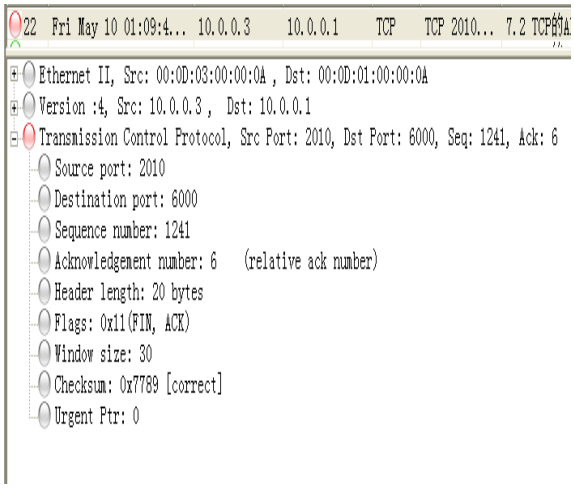
- Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
- Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
- Transmission Control Protocol, Src Port: 2009, Dst Port: 7000, Seq: 1240, Ack: 6
  - Source port: 2009
  - Destination port: 7000
  - Sequence number: 1240
  - Acknowledgement number: 6 (relative ack number)
  - Header length: 20 bytes
  - Flags: 0x10(ACK)
  - Window size: 30
  - Checksum: 0x73A4 [correct]
  - Urgent Ptr: 0

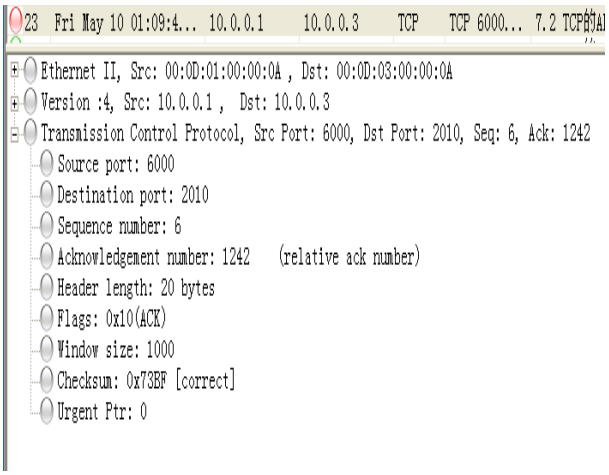
20 Fri May 10 01:09:4... 10.0.0.1 10.0.0.3 TCP TCP 6000... 7.2 TCP

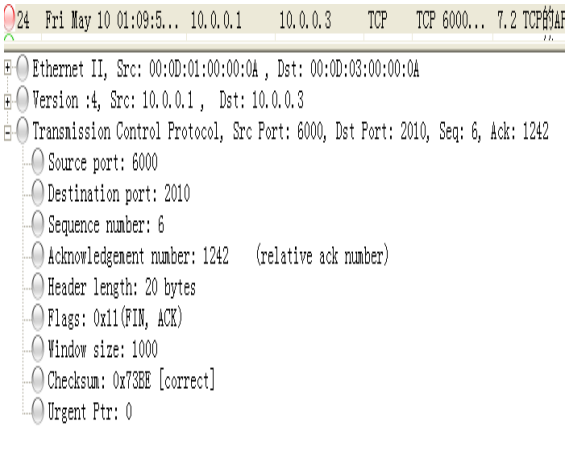
- Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
- Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
- Transmission Control Protocol, Src Port: 6000, Dst Port: 2010, Seq: 2, Ack: 1241
  - Source port: 6000
  - Destination port: 2010
  - Sequence number: 2
  - Acknowledgement number: 1241 (relative ack number)
  - Header length: 20 bytes
  - Flags: 0x10(ACK)
  - Window size: 1000
  - Checksum: 0x0B8C [correct]
  - Urgent Ptr: 0
- Data(4 bytes)

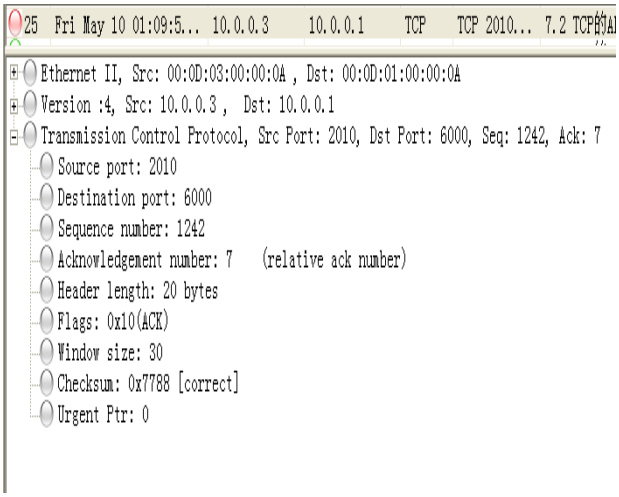
21 Fri May 10 01:09:4... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.2 TCP

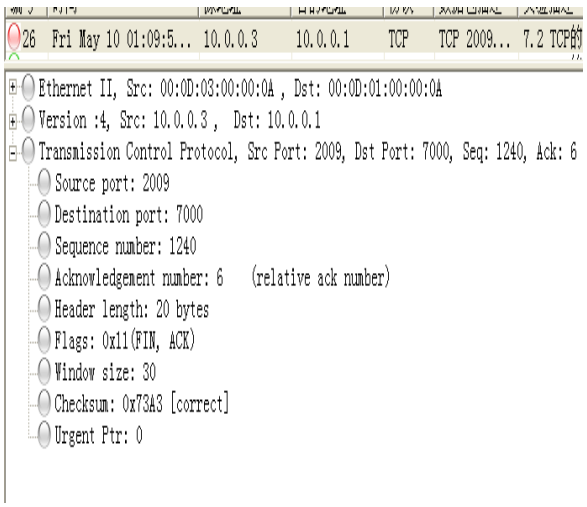
- Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
- Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
- Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1241, Ack: 6
  - Source port: 2010
  - Destination port: 6000
  - Sequence number: 1241
  - Acknowledgement number: 6 (relative ack number)
  - Header length: 20 bytes
  - Flags: 0x10(ACK)
  - Window size: 30
  - Checksum: 0x778A [correct]
  - Urgent Ptr: 0

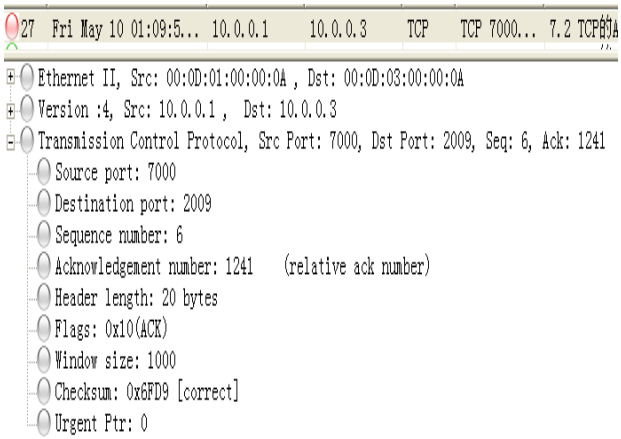

22 Fri May 10 01:09:4... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.2 TCP的A
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1241, Ack: 6
Source port: 2010
Destination port: 6000
Sequence number: 1241
Acknowledgement number: 6 (relative ack number)
Header length: 20 bytes
Flags: 0x11(FIN, ACK)
Window size: 30
Checksum: 0x7789 [correct]
Urgent Ptr: 0

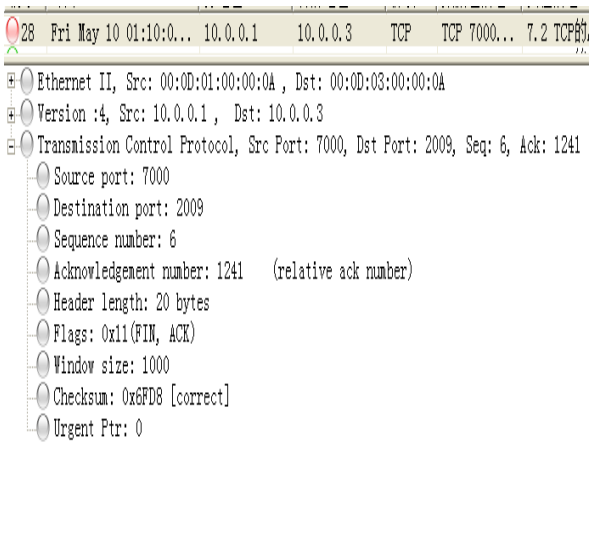

23 Fri May 10 01:09:4... 10.0.0.1 10.0.0.3 TCP TCP 6000... 7.2 TCP的AI
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 6000, Dst Port: 2010, Seq: 6, Ack: 1242
Source port: 6000
Destination port: 2010
Sequence number: 6
Acknowledgement number: 1242 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 1000
Checksum: 0x73BF [correct]
Urgent Ptr: 0

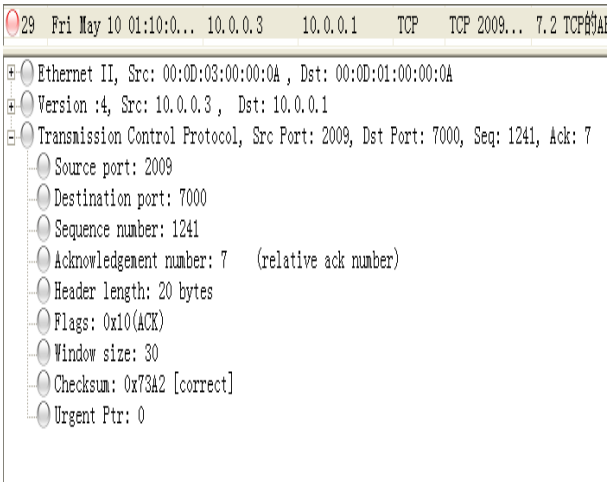

24 Fri May 10 01:09:5... 10.0.0.1 10.0.0.3 TCP TCP 6000... 7.2 TCP的AF
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 6000, Dst Port: 2010, Seq: 6, Ack: 1242
Source port: 6000
Destination port: 2010
Sequence number: 6
Acknowledgement number: 1242 (relative ack number)
Header length: 20 bytes
Flags: 0x11(FIN, ACK)
Window size: 1000
Checksum: 0x73BE [correct]
Urgent Ptr: 0


25 Fri May 10 01:09:5... 10.0.0.3 10.0.0.1 TCP TCP 2010... 7.2 TCP的AI
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2010, Dst Port: 6000, Seq: 1242, Ack: 7
Source port: 2010
Destination port: 6000
Sequence number: 1242
Acknowledgement number: 7 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 30
Checksum: 0x7788 [correct]
Urgent Ptr: 0


26 Fri May 10 01:09:5... 10.0.0.3 10.0.0.1 TCP TCP 2009... 7.2 TCP的
Ethernet II, Src: 00:0D:03:00:00:0A , Dst: 00:0D:01:00:00:0A
Version :4, Src: 10.0.0.3 , Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 2009, Dst Port: 7000, Seq: 1240, Ack: 6
Source port: 2009
Destination port: 7000
Sequence number: 1240
Acknowledgement number: 6 (relative ack number)
Header length: 20 bytes
Flags: 0x11(FIN, ACK)
Window size: 30
Checksum: 0x73A3 [correct]
Urgent Ptr: 0


27 Fri May 10 01:09:5... 10.0.0.1 10.0.0.3 TCP TCP 7000... 7.2 TCP的A
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version :4, Src: 10.0.0.1 , Dst: 10.0.0.3
Transmission Control Protocol, Src Port: 7000, Dst Port: 2009, Seq: 6, Ack: 1241
Source port: 7000
Destination port: 2009
Sequence number: 6
Acknowledgement number: 1241 (relative ack number)
Header length: 20 bytes
Flags: 0x10(ACK)
Window size: 1000
Checksum: 0x6FD9 [correct]
Urgent Ptr: 0





报文分析：

NO.	Src_port	Dst_port	Seq	Ack	Flags	Description
8	2009	7000	1235	0	SYN	本地请求建立连接，第一次握手
9	7000	2009	1	1236	SYN+ACK	对方对于建立连接的回复，第二次握手
10	2009	7000	1236	2	ACK	本地返回建立连接确认，第三次握手
11	2010	6000	1236	0	SYN	本地请求连接，第一次握手
12	6000	2010	1	1237	SYN+ACK	对方对于建立连接的回复，第二次握手
13	2010	6000	1237	2	ACK	本地返回建立连接确认，第三次握手
14	2010	6000	1237	2	ACK	本地发送数据，数据长度为 4 个字节
15	6000	2010	2	1241	ACK	对方收到数据，返回 ACK

16	2009	7000	1236	2	ACK	本地发送数据，数据长度为 4 个字节
17	7000	2009	2	1240	ACK	对方收到数据，返回 ACK
18	7000	2009	2	1240	ACK	对方发送数据，数据长度为 4 个字节
19	2009	7000	1240	6	ACK	本地收到数据，返回 ACK
20	6000	2010	2	1241	ACK	对方发送数据，数据长度为 4 个字节
21	2010	6000	1241	6	ACK	本地收到数据后，返回 ACK
22	2010	6000	1241	6	FIN+ACK	本地请求关闭连接
23	6000	2010	6	1242	ACK	对方响应关闭连接请求
24	6000	2010	6	1242	FIN+ACK	对方处理完数据后，指示本地关闭连接
25	2010	6000	1242	7	ACK	本地接收指示，然后返回确认
26	2009	7000	1240	6	FIN+ACK	本地请求关闭连接
27	7000	2009	6	1241	ACK	对方响应关闭连接请求
28	7000	2009	6	1241	FIN+ACK	对方处理完数据，指示本地关闭连接
29	2009	7000	1241	7	ACK	本地接收指示，返回确认

五、实验中遇到的问题

【1】TCP 校验和计算的问题：

刚开始以为和 IP 协议相同那样计算，后来维基了一下，学习了 TCP 校验和计算的时候需要添加一个 TCP pseudo_header 伪头部字段。

【2】TCP 关闭连接的时候是采用了四次握手，中间存在 TIME_WAIT 状态，是为了保证网络中参与的或者重复的数据包传送到服务器端得到处理，避免其对后面的 TCP 连接造成影响。

TIME_WAIT 状态下等待时间是 MSL 时间的 2 倍，MSL 是一个数据报在网络中单向发出到认定丢失的时间，一个数据报有可能发送途中或者是其影响过程中成为残余数据报，确认一个数据包及其响应的丢弃的需要 2 倍 MSL 时间，将会变成 CLOSED 状态。

一个成功的连接的建立，必须是先前网络中残余的数据报都被丢弃掉。

六、思考问题

-1-总结 TCP 协议中的序号的处理方式。

主要分析确认号和序列号的作用和处理方式：
序列号是为了保证 TCP 传输数据包的顺序性，确认号是用于判定能否在规定的时间内收到确认信息，如果没有的话需要重新发送。

TCP 连接建立的时候：

-1-客户端发送同步数据包请求建立连接，产生的数据包随机产生 SYN，ACK 为 0；

-2-服务器收到同步请求数据包之后，会对客户端进行一个同步确认。确认号 ACK 是客户端的 SYN+1; 和一个随机产生的 SYN

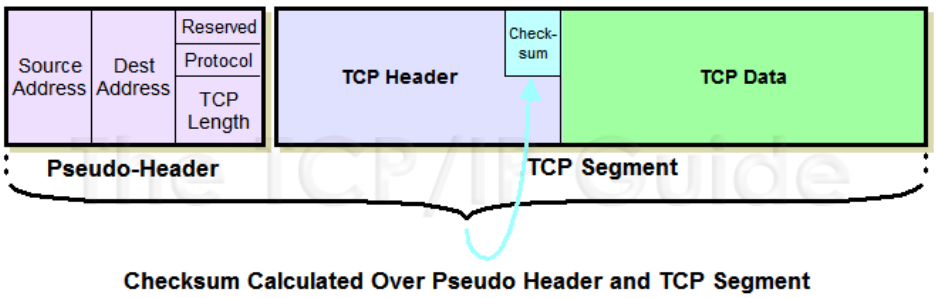
-3-客户端收到同步数据包之后，再给服务器发送确认帧，序列号是上次的请求序列号 SYN+1 得到的。

TCP 关闭连接的时候也需要做类似 的对序号的处理；

在调用 stud_tcp_send()函数的时候，内部需要判断收到的 ACK 信号是否是对于前面发送的所有报文的确认信号。

尤其在 FSM()状态机转换函数中需要对需要进行处理和判断。

-2-试比较 TCP 校验和的计算与前面的 IPv4 转发/收发实验中的校验和的计算有何异同。



Checksum Calculated Over Pseudo Header and TCP Segment

1) 除了 TCP 包本身,TCP 校验数据块还包括源 IP 地址,目的 IP 地址, TCP 包长度, TCP 协议号组成的 12 字节伪首部，见上图；

2) TCP 包的校验方式和 IPv4 的校验方式一致，即使用反码求和校验。校验算法具体如下：

发送方

i) 将校验和字段置为 0,然后将 IP 包头按 16 比特分成多个单元,如包头长度不是 16 比特的倍数,则用 0 比特填充到 16 比特的倍数;

ii) 对各个单元采用反码加法运算(即高位溢出位会加到低位,通常的补码运算是直接丢掉溢出的高位),将得到的和的反码填入校验和字段;

iii) 发送数据包.

接收方

i) 将 IP 包头按 16 比特分成多个单元,如包头长度不是 16 比特的倍数,则用 0 比特填充到 16 比特的倍数;

ii) 对各个单元采用反码加法运算,检查得到的和是否符合是全 1(有的实现可能对得到的和会取反码,然后判断最终值是不是全 0);

iii) 如果是全 1 则进行下步处理,否则意味着包已变化从而丢弃之.需要强调的是反码和是采用高位溢出加到低位的,如 3 比特的反码和运算:100b+101b=010b(因为 100b+101b=1001b,高位溢出 1,其应该加到低位,即 001b+1b(高位溢出位)=010b)

七、 实验的启示/意见和建议

此次实验大约花费了 24+小时吧！包括写代码+调代码+写报告 ，通过此次实验确实从 2 方面加深了对于 TCP 报文段收发函数的理解以及通过实现 socket API 函数也深入理解了 API 设计时的大体设计框架，当然处理的客户端的状态机流转是比实际应用中的简单很多的，但是还是学习到了很多细节和知识，纠正了很多自己以前不完整的认识。