NetRiver 4–TCP 协议实验

张煌昭，1400017707，元培学院

摘要—本次作业实现了 TCP 协议的有限状态机，并基于其实现了 TCP 协议的接受和封装处理；此外还实现了 TCP 协议上的 Socket 函数接口，允许 Socket 通信。实验于 NetRiver 平台进行提交和测试。本次报告使用 Overleaf L^AT_EX 在线平台编写[1]，作业源码附于报告之后。

I. 介绍

本次作业，要求使用 C/C++ 语言，针对客户端角色的“停-等”模式的 TCP 协议，完成接受和发送流程的设计和实现。具体需要完成 TCP 报文接收的有限状态机，TCP 报文发送的封装处理，以及实现客户端 Socket 接口。

II. 实验原理

A. TCP 协议用户状态机

由于完全实现 TCP 协议十分复杂且难度较大，本次实验实现采用“停-等”模式的 TCP 协议的客户端部分，由于采用停等协议，发送窗口和接收窗口大小均为 1。由于客户端不能监听端口等待连接，而只能主动发起建立连接的请求，因此其有限状态机可以表示为如图 1所示的环。

状态机由关闭状态起，客户端使用 Connect() 函数发送 SYN 到达 SYN 已发送的等待状态，接收 SYN+ACK 并回复 ACK 至连接建立状态，在连接建立状态下，客户端和服务器可以进行自由的 Socket 通信。通信结束后客户端使用 Close() 函数发送 FIN 至 FIN 等待状态，之后收到 ACK 回复进入半关闭状态，在收到服务器关闭 FIN 并回复 ACK 确认后至超时等待状态，等待一段时间超时后到达关闭状态，关闭 TCP 连接。如上即为 TCP 协议的三次握手建立连接，四次挥手关闭连接。

[1]本报告源码可通过以下 git 命令获得，
git clone https://git.overleaf.com/15853721gmnmbcjkdywj

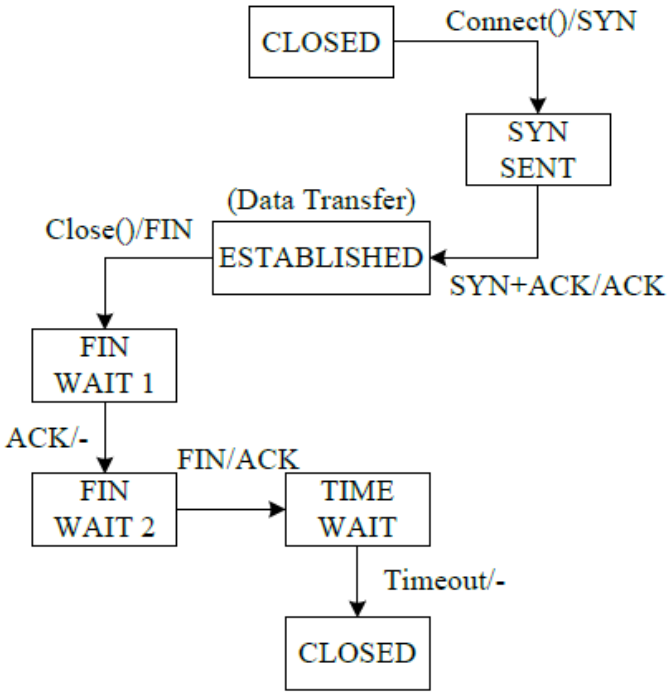

图 1. TCP 协议客户端有限状态机。FSM 成环，由关闭状态起，客户端使用 Connect() 函数发送 SYN 到达 SYN 已发送的等待状态，接收 SYN+ACK 并回复 ACK 至连接建立状态，客户端使用 Close() 函数发送 FIN 至 FIN 等待状态 1，之后收到 ACK 回复至等待状态 2，在收到 FIN 并回复 ACK 确认后至超时等待状态，超时后到达关闭状态，完成一次完整的 TCP 连接。

III. Socket 接口

TCP 层需要通过 Socket 接口向应用层提供支持。Socket 接口包括：1) 创建 Socket 接口，创建新的 TCB，即创建一个新的未进行连接的 TCP 连接；2) TCP 连接接口，正确填写 TCB，进行 TCP 三次握手建立连接；3) 发送消息接口，发送消息并进行接收确认；4) 接收消息接口，接收消息并进行接收确认；5) 关闭 TCP 连接接口，进行 TCP 四次挥手建立连接。

IV. 数据结构

A. 传输控制块

传输控制块（TCB）为 TCP 协议控制每个 TCP 连接发送和接收动作的数据结构，所有 TCP 连接对应的 TCB 可以组织成链表结构或树结构进行管理。TCB

中包含源地址和端口，目标地址和端口，连接状态，序列号和确认号，Socket 连接文件号等信息。

本次实验中的 TCB 数据结构如下。TCB 表通过 C++ STL 的 map 实现，键为 Socket 描述符，值为 TCB 指针，具体如下。

```
typedef struct __TCB__
{
    unsigned int srcAddr;    // 源IP
    unsigned int dstAddr;    // 目标IP
    unsigned short srcPort; // 源端口
    unsigned short dstPort; // 目标端口
    unsigned int seq;        // 序列号
    unsigned int ack;        // ACK号
    int sockfd;      // Socket连接号
    BYTE state;      // 连接状态
    unsigned char* data;     // 数据
    void init();     // 初始化函数
} TCB;
map<int, TCB*> TCBTable;
```

其中初始化函数，设置 TCP 连接状态为关闭（CLOSED），设置自增的 Socket 连接号和自增的源端口号。

B. TCP 报文头部

TCP 报文头部类似于 IPv4 头部，具有源端口，目标端口，序列号，确认号，头部长度，类型标签（SYN，ACK 等），窗口大小，头部检查和等字段，如图 2所示。

本次实验中的 TCP 报文头部的数据结构如下。

```
typedef struct __TCPHead__
{
    UINT16 srcPort;      // 源端口
    UINT16 destPort;     // 目标端口
    UINT32 seqNo;        // 序列号
    UINT32 ackNo;        // 确认号
    UINT8 headLen;       // 头部长度
    UINT8 flag;          // 类型标签
    UINT16 windowsize;   // 窗口大小
    UINT16 checksum;     // 校验和
    UINT16 urgentPointer;
    char data[];         // 数据段
    void ntoh();     // 小端转换为大端
```

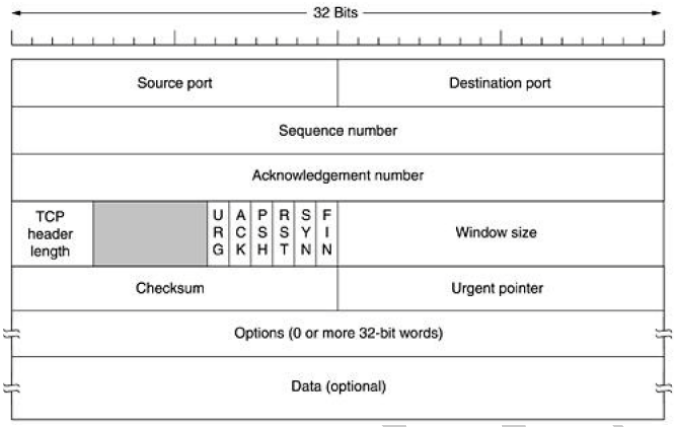

图 2. TCP 报文结构。一行为一个 32bit 字，从上向下从左到右，报文头部内容包括源端口，目标端口，序列号，确认号，头部长度，类型标签（SYN，ACK 等），窗口大小，检查和等字段。

```
    // 计算校验和
    unsigned int CheckSum();
} TCPHead;
```

其中 ntoh 函数将 x86 机器小端法存放的数据转换为网络传输时大端法存放的数据，CheckSum 函数接收源和目标地址，类型和长度，计算需要更新的检验和。

V. 实现

A. TCP 层实现

TCP 层需要实现 stud_tcp_input() 和 stud_tcp_output() 两个函数。

stud_tcp_input() 函数接收 TCP 报文，之后需要转化大小端，检查 TCP 头部校验和，检查序列号，进行状态转换。stud_tcp_input() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_input(char *pBuffer,
                   unsigned short len,
                   unsigned int srcAddr,
                   unsigned int dstAddr)
{
    // 大小端转换

    // 检查校验和
    // 检查序列号

    // 状态转换
        // SYN_SEND状态下，发送ACK，转移至ESTABLISHED
        // FIN_WAIT1状态下，转移至FIN_WAIT2
        // FIN_WAIT2状态下，发送ACK，转移至TIME_WAIT

    return 0;
}
```

stud_tcp_output() 函数发送 TCP 报文，判断是否可以发送，填写 TCP 报文头部字段，进行状态转换。stud_tcp_output() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
void stud_tcp_output(char *pData,
                     unsigned short len,
                     unsigned char flag,
                     unsigned short srcPort,
                     unsigned short dstPort,
                     unsigned int srcAddr,
                     unsigned int dstAddr)
{
    // 若为新连接，创建新的TCB

    // 创建新TCP报文头部，并填写各字段
    // 大小端转换
    // 发送TCP报文

    // 状态转换
        // CLOSED状态下，发送SYN，转移至SYN_SENT
        // ESTABLISHED状态下，发送FIN，转移至FIN_WAIT1
}
```

B. Socket 接口实现

Socket 接口函数包括 stud_tcp_socket() 函数创建 Socket，stud_tcp_connect() 函数建立 TCP 连接，stud_tcp_send() 函数发送消息，stud_tcp_recv() 函数接收消息和 stud_tcp_close() 函数关闭 TCP 连接。

stud_tcp_socket() 函数创建 Socket 接口，其中需要创建 TCB 并初始化，并将 TCB 加入 TCB 表中，返回 Socket 描述符。stud_tcp_socket() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_socket(int domain,
                    int type,
                    int protocol)
{
    // 创建TCB并初始化
    // 将新创建的TCB加入TCB表
    // return socket fd;
}
```

stud_tcp_connect() 函数建立 TCP 连接，其中需要从 TCB 表中利用 Socket 描述符查找 TCB 并填写各个字段，完成三次握手并进行相应的状态转换。stud_tcp_connect() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_connect(int sockfd,
                     struct sockaddr_in *addr,
                     int addrlen)
{
    // 利用sockfd，查找TCB
    // 填写目标地址和端口等字段
    // 大小端转换

    // 发送SYN，开始三次握手
    // 等待SYN+ACK
    // 更新序列号等，回复ACK
    // 状态到达ESTABLISHED
    return -1;
}
```

stud_tcp_send() 函数通过建立的 TCP 连接发送消息，其中需要检查连接状态，填写并发送 TCP 报文头部，等待 ACK 回复并更新序列号。stud_tcp_send() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_send(int sockfd,
                  const unsigned char *pData,
                  unsigned short datalen,
                  int flags)
{
    // 利用sockfd，查找TCB
    // 检查TCB状态是否为ESTABLISHED
        // 否则报错

    // 发送数据
    // 等待ACK并检查
        // 若序列号或确认号出错则丢弃报错
}
```

stud_tcp_recv() 函数通过建立的 TCP 接收消息，其中需要检查连接状态，读出数据并回复 ACK，将其交给应用层。stud_tcp_recv() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_recv(int sockfd,
                  unsigned char *pData,
                  unsigned short datalen,
                  int flags)
{
    // 利用sockfd，查找TCB
    // 检查TCB状态是否为ESTABLISHED
        // 否则报错

    // 接受数据
    // 回复ACK
}
```

stud_tcp_close() 函数关闭 TCP 连接，若 TCB 为 ESTABLISHED 状态则进行正确的状态转换，否则直接删除 TCB，正常状态转换时发送 FIN 后等待 ACK。stud_tcp_close() 函数的定义，及 C 语言风格伪代码如下。实现详情请见附件源码。

```
int stud_tcp_close(int sockfd)
{
```

```
    // 利用sockfd，查找TCB
    // 检查TCB状态是否为ESTABLISHED
        // 否则报错

    // 若不为ESTABLISHED
        // 则直接删除退出
    // 发送FIN，状态转移至FIN_WAIT1

    // 等待回复ACK，状态转移至FIN_WAIT2
    // 等待FIN，回复ACK，状态转移至TIME_WAIT
}
```

VI. 代码

本次作业详细代码请见附录部分。

附录

```
/*
* 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
typedef 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;
    }
} TCB;

// TCP header
typedef 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;
    }
} TCPHead;


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;
}
```