《计算机网络协议开发》实验报告

第 12 次实验

STCP 协议:信号实现

姓名：元玉慧
学号：101220151
10 级 计算机 系 4 班
邮箱：njucsyyh@gmail.com
时间：2013/05/19

一、 实验目的

通过本实验熟悉传输控制层协议的设计，实现。熟悉传输控制协议是如何建立的，以及如何解决信号数据包的丢失和损坏。

二、实验设计背景

从 SimpleNet 通信协议允许客户和服务器创建套接字，打开连接以进行可靠的数据传输。SimpleNet 是一个分层架构，第 n-1 层为第 n 层提供服务和 API,例如 SCTP 为应用层提供可靠的字节流传输。SIP 为传输层提供类似 IP 的端系统之间的连接，重叠网络层 SON 为运行 SimpleNet 的端主机建立网络。TCP/IP 运行在重叠网络之上，类比于：SON+SIP 是网络层，TCP/IP 是数据链路层和物理层。

STCP 的一些假设：
-1- 单向传输：数据段从客户端流向服务器端；
-2- 连接管理：由客户端发起和关闭连接
-3- 不支持流控或者是拥塞控制

STCP 需要实现：
连接管理：连接的建立和关闭；
校验和： 接收端用它来检测被破坏的数据；
实现 GBN，协议以保证数据的可靠传输，使用序列号来检测丢失的数据并按序接收数据。
重传：发送端使用超时和用于确认收到数据

本次实验的个人理解：
说白了我们所谓的 STCP 的通信还是依靠 TCP 的 socket 函数，只不过对其基本的 2 个操作（send(), recv() ）进行了封装到不同的函数中，创建了 TCP 层次的套接字连接，STCP 层要记住客户端和服务器端对应的套接字描述符，sockfd 和 connfd，然后在重叠的 STCP 层次上实现了初始化，分配 STCP 套接字（即 TCB 数组的下标），建立连接初始化，发送报文函数（本次不要求实现）， 关闭连接，释放套接字等操作。
STCP 的连接建立和连接关闭都醉 TCP 做了简化，STCP 连接的建立是 2 路握手，客户发送 SYN，然后服务器响应 SYNACK 后，连接就建立成功。
但是控制信息可能会丢失，客户端知道它发送了 SYN，但是服务器没有收到，或者是服务器发送了 SYNACK，但是客户端没有收到，都是交给客户端 stcp_client 处理，通过设置一个 SYN_TIMEOUT, 如果它在这段时间内收到 SYNACK, 它就再次发送 SYN 并成功新设置定时器，它重复此操作，直到它收到一个 SYNACK 或者重试次数过多之后。就会放弃连接。
关闭连接时的情况是：客户端发送 FIN,然后服务器端回复 FINACK。

主要的收发操作都是在客户端和服务器端的线程中，需要根据客户端和服务器端的收到的数据段中携带的 type 信息做不同的状态流转。

三、实验内容

简单传输控制协议，STCP

STCP 客户端函数原型：

void stcp_client_init(int conn)

```
void stcp_client_init(int conn) {
  //printf("Client: Init STCP client successfully ...\n");
  int i;
    for(i=0; i<TCB_MAX; i++){
    if(tcblist[i].tcb != NULL) free(tcblist[i].tcb);
    tcblist[i].tcb = NULL;
    tcblist[i].tag = 0;
  }

  STCP_client = conn;

  pthread_t tid;
    pthread_create(&tid, NULL, seghandler, (void *)conn);
    return;
}
```

Int stcp_client_sock(unsigned int client_port)

```
int stcp_client_sock(unsigned int client_port) {
  int i;
  for(i=0; i<TCB_MAX; i++){
    if(tcblist[i].tag == 0) {
       tcblist[i].tcb = malloc(sizeof(client_tcb_t));
       tcblist[i].tcb->state = CLOSED;
       tcblist[i].tcb->client_portNum = client_port;
       tcblist[i].tcb->server_nodeID = 0;
       tcblist[i].tcb->server_portNum = 0;
       tcblist[i].tcb->client_nodeID = 0;
       tcblist[i].tcb->next_seqNum = 0;
       tcblist[i].tcb->bufMutex = NULL;
       tcblist[i].tcb->sendBufHead = NULL;
       tcblist[i].tcb->sendBufunSent = NULL;
       tcblist[i].tcb->sendBufTail = NULL;
       tcblist[i].tcb->unAck_segNum = 0;
       tcblist[i].tag = 1;
      // printf("Client: Creat STCP client tcb item successfully ...\n");
       break;
     }
  }
  if(i==TCB_MAX)  return -1;
    return i;
}
```

Int stcp_client_connect(int sockfd, unsigned int server_port)

```
int stcp_client_connect(int sockfd, unsigned int server_port) {
  tcblist[sockfd].tcb->server_portNum = server_port;
  seg_t *send_seg = malloc(sizeof(seg_t));
  send_seg->header.type = SYN;
  send_seg->header.src_port = tcblist[sockfd].tcb->client_portNum;
  send_seg->header.dest_port = server_port;

  sip_sendseg(STCP_client, send_seg);

  tcblist[sockfd].tcb->state = SYNSENT;
  printf("\n**********************\n");
  printf("SYN : client ---> server\n");
  printf("**********************\n");
  //printf("Client: Now state is SYNSENT , waiting for the SYNACK ...\n");

  int count;
  for(count=0; count < SYN_MAX_RETRY; count++){
    usleep(SYN_TIMEOUT/1000);
      if(tcblist[sockfd].tcb->state == CONNECTED){
      printf("Client: Receive the SYNACK successfully ...\n");
          return 1;
    }
      else{
      printf("resend the message ...\n");
          sip_sendseg(STCP_client, send_seg);
    }


  if(count == SYN_MAX_RETRY){
    printf("Client: Having tried to connect for many times, but failed ...\n");
    tcblist[sockfd].tcb->state = CLOSED;
      return -1;
  }
}
```

Int stcp_client_disconnect()

```
int stcp_client_disconnect(int sockfd) {
    seg_t *send_seg = malloc(sizeof(seg_t));
    send_seg->header.src_port = tcblist[sockfd].tcb->client_portNum;
    send_seg->header.dest_port = tcblist[sockfd].tcb->server_portNum;
    send_seg->header.type = FIN;
    sip_sendseg(STCP_client, send_seg);
    printf("Client: +++  CONNECTED ------>  FINWAIT\n");
    tcblist[sockfd].tcb->state = FINWAIT;
    //printf("Client: Now state is FINWAIT, waiting for the FINACK ...\n");
    printf("\n**********************\n");
    printf("FIN: client ---> server\n");
    printf("**********************\n");
    int count;
```

```
  for(count=0; count < FIN_MAX_RETRY; count++){
    usleep(FIN_TIMEOUT/1000);

      if(tcblist[sockfd].tcb->state == CLOSED){
      printf("Client: Receive the FINACK successfully ...\n");
      //printf("Client: Close the connect to the server successfully ...\n");
          return 1;
    }
      else{
          sip_sendseg(STCP_client, send_seg);
printf("resend the message ...\n");
    }
  }

  if(count == FIN_MAX_RETRY){
    tcblist[sockfd].tcb->state = CLOSED;
    printf("Client: Having tried to close connect for many times, but failed ...\n");
      return -1;
  }
}
```

Int stcp_client_close()

```
int stcp_client_close(int sockfd) {
//printf( "THE CURRENT STATE IS %d\n", sockfd);
  int state = tcblist[sockfd].tcb->state;
  if(tcblist[sockfd].tcb != NULL)
      free(tcblist[sockfd].tcb);
  tcblist[sockfd].tcb = NULL;
  tcblist[sockfd].tag = 0;
  if(state==1) {
    printf("Client: Free the client tcb item successfully ...\n");
      return 1;
  }
  else {
    printf("Client: Failed to free the client tcb item successfully ...\n");
      return -1;
  }
}
```

Void* seghandler(void * arg)

```
void *seghandler(void* arg) {
  char buffer[sizeof(seg_t)];

  while(1){
    //printf("on going\n");
    memset(buffer, 0, sizeof(seg_t));
    int test = sip_recvseg(STCP_client, (seg_t*)buffer);
    if(test == -1){
         pthread_exit(NULL);
    }
    //
      if(test == 1){
      //printf("Client: The seg info is partly lost ... \n");
      //drop out the packet ...
        continue;
    }
//~ printf("after continue ...\n");
    int server_port = ((seg_t*)buffer)->header.src_port;
    int client_port = ((seg_t*)buffer)->header.dest_port;
    int server_type = ((seg_t*)buffer)->header.type;
    int state;
    int i;
    int index;

 for(i=0; i<TCB_MAX; i++){
     if( (tcblist[i].tag==1)&& (tcblist[i].tcb->server_portNum==server_port) ){
     state = tcblist[i].tcb->state;
     printf("find the tcb ...\n");
     index = i;
   }
 }

 switch(state){
     case CLOSED:{
     break;
   }
   case SYNSENT:{
     if(server_type==SYNACK){
         tcblist[index].tcb->state = CONNECTED;
         printf("\n***********************\n");
         printf("SYNACK: server ---> client\n");
         printf("***********************\n");
         printf("Client: +++  SYNSENT ------>  CONNECTED\n");
     }
     break;
   }
```

```
        case FINWAIT:{
          if(server_type==FINACK){
             tcblist[index].tcb->state = CLOSED;
             printf("\n***********************\n");
             printf("FINACK: server ---> client\n");
             printf("***********************\n");
             printf("Client: +++  FINWAIT ------>  CLOSED\n");
            }
            break;
        }
          default:{
              printf("Client: The state is wrong ...\n");
              break;
          }
      }
    }
}
```

STCP 服务端函数原型

Void stcp_server_init (int conn)

```
void stcp_server_init(int conn) {
  //printf("Server: Init STCP server successfully ...\n");
  int i;
    for(i=0; i<MAX_TRANSPORT_CONNECTIONS; i++){
    if(tcblist[i].tcb != NULL) free(tcblist[i].tcb);
    tcblist[i].tcb = NULL;
    tcblist[i].tag = 0;
  }
  //初始一个全局变量？？
  //client_tcb_t *tp = new client_tcb_t;
  STCP_server = conn;
  pthread_t tid;
    pthread_create(&tid, NULL, seghandler, (void *)conn);
}
```

Void stcp_server_sock (int conn)

```
int stcp_server_sock(unsigned int server_port) {
  int i;
  for(i=0; i<MAX_TRANSPORT_CONNECTIONS; i++){
    if(tcblist[i].tag == 0) {
       tcblist[i].tcb = (server_tcb_t *)malloc(sizeof(server_tcb_t));
       tcblist[i].tcb->state = CLOSED;
       tcblist[i].tcb->server_portNum = server_port;
       tcblist[i].tcb->server_nodeID = 0;
       tcblist[i].tcb->client_portNum = 0;
       tcblist[i].tcb->client_nodeID = 0;
       tcblist[i].tcb->expect_seqNum = 0;
       tcblist[i].tcb->recvBuf = NULL;
       tcblist[i].tcb->usedBufLen = 0;
       tcblist[i].tcb->bufMutex = NULL;
       tcblist[i].tag = 1;
       break;
     }
  }
```

```
  if(i==MAX_TRANSPORT_CONNECTIONS)  return -1;
    return i;
}
```

Void stcp_server_accept (int sockfd)

```
int stcp_server_accept(int sockfd) {
  //printf("Server: Waiting for the  STCP client connect to the server ...\n");
  tcblist[sockfd].tcb->state = LISTENING;
  //
  while(1){
     usleep(ACCEPT_POLLING_INTERVAL / 1000);
     if(tcblist[sockfd].tcb->state==CONNECTED){
       //printf("Server: STCP client connect to the server successfully ...\n");
         break;
     }
  }
  return 1;
}
```

Void stcp_server_recv (int sockfd, void *buf, unsigned int length)

Int stcp_server_close(int sockfd)

```
int stcp_server_close(int sockfd) {
  sleep(CLOSEWAIT_TIMEOUT);
  printf("Server: +++  CLOSEWAIT ------>  CLOSED\n");
    tcblist[sockfd].tcb->state = CLOSED;
  int state = tcblist[sockfd].tcb->state;
  if(tcblist[sockfd].tcb != NULL)
       free(tcblist[sockfd].tcb);
  tcblist[sockfd].tag = 0;
  return 1;
}
```

服务器端的处理线程函数实现：

```
void *seghandler(void* arg) {
  char buffer[sizeof(seg_t)];

  while(1){
    memset(buffer, 0, sizeof(seg_t));
    int t = sip_recvseg(STCP_server, (seg_t*)buffer);
    if(t == -1){
        pthread_exit(NULL);
    }

    if(t == 1){
      //printf("Server: The seg info is partly lost ... \n");
      //drop out the packet ...
        continue;
    }
    int client_port = ((seg_t*)buffer)->header.src_port;
```

```
int server_port = ((seg_t*)buffer)->header.dest_port;
int client_type = ((seg_t*)buffer)->header.type;
int state;
int i;
int index;
for(i=0; i<10; i++){
    if(tcblist[i].tag==1 && tcblist[i].tcb->server_portNum == server_port){
    state = tcblist[i].tcb->state;
    index = i;
   }
}

switch(state){
    case CLOSED:{
    break;
  }

   case LISTENING:{
     if(client_type==SYN){
       printf("Server: +++  LISTENING ------>  CONNECTED\n");
           tcblist[index].tcb->state = CONNECTED;
           tcblist[index].tcb->client_portNum = client_port;
         seg_t* temp = (seg_t *)malloc(sizeof(seg_t));
       temp->header.type = SYNACK;
       temp->header.src_port = server_port;
       temp->header.dest_port = client_port;
       sip_sendseg(STCP_server, temp);
       printf("\n**********************\n");
       printf("SYNACK: server ---> client\n");
       printf("**********************\n");
     }
     break;
   }

   case CONNECTED:{
     if(client_type==FIN){
       printf("Server: +++  CONNECTED ------>  CLOSEWAIT\n");
       tcblist[index].tcb->state = CLOSEWAIT;
         seg_t* temp = (seg_t *)malloc(sizeof(seg_t));
       temp->header.type = FINACK;
       temp->header.src_port = server_port;
       temp->header.dest_port = client_port;
       sip_sendseg(STCP_server, temp);
```

```
        printf("\n**********************\n");
        printf("FINACK: server ---> client\n");
        printf("**********************\n");
        }
      //send data

      if(client_type==SYN){
        printf("Server: +++  CONNECTED ------>  CONNECTED\n");
        tcblist[index].tcb->state = CONNECTED;
        seg_t* temp = (seg_t *)malloc(sizeof(seg_t));
        temp->header.type = SYNACK;
        temp->header.src_port = server_port;
        temp->header.dest_port = client_port;
        sip_sendseg(STCP_server, temp);
        printf("\n**********************\n");
        printf("SYNACK: server ---> client\n");
        printf("**********************\n");
      }

      break;
    }
```

```
      case CLOSEWAIT:{
        if(client_type==FIN){
          tcblist[index].tcb->state = CLOSEWAIT;
          printf("Server: +++  CLOSEWAIT ------>  CLOSEWAIT\n");
          seg_t* temp = (seg_t *)malloc(sizeof(seg_t));
          temp->header.type = FINACK;
          temp->header.src_port = server_port;
          temp->header.dest_port = client_port;
          sip_sendseg(STCP_server, temp);
          printf("\n**********************\n");
          printf("FINACK: server ---> client\n");
          printf("**********************\n");
         }
         break;
      }

       default:{
          printf("Server: The state is wrong ...\n");
          break;
       }
    }
  }
}
```

SIP 函数原型分析：

Int sip_sendseg(int connection, seg_t * segPtr)

```
int sip_sendseg(int connection, seg_t* segPtr)
{
  if( send(connection, "!", 1, 0) == -1)     return -1;
  if( send(connection, "&", 1, 0) == -1)     return -1;
  //发送方是确定了数据的大小的
  if( send(connection, segPtr, sizeof(seg_t), 0) != sizeof(seg_t))  return -1;
  if( send(connection, "!", 1, 0) == -1)     return -1;
  if( send(connection, "#", 1, 0) == -1)     return -1;
    return 1;
}
```

Int sip_recvseg(int connection , seg_t* segPtr)

```
int sip_recvseg(int connection, seg_t* segPtr)
{
  int state = SEGSTART1;
  int test = 0;
  int index = 0;
  char buffer[sizeof(seg_t)];
  char c;
  memset(buffer, 0, sizeof(seg_t));
  while(1){
    if(test==1) break;
      switch(state){
    case SEGSTART1:{
      //printf("a");
      if(recv(connection, &c, 1, 0)==-1) return -1;
      if(c=='!')
         state = SEGSTART2;
      break;
    }


    case SEGSTART2:{
    //printf("sip_recvseg state 2\n");
    if(recv(connection, &c, 1, 0)==-1) return -1;
    if(c=='&')
       state = SEGRECV;
    else
       state = SEGSTART1;
    break;
  }

    case SEGRECV:{
    //printf("sip_recvseg state 3\n");
    while(1){
      if(recv(connection, &c, 1, 0)==-1) return -1;
      if(c=='!'){
        state = SEGSTOP1;
          break;
      }
      else
          buffer[index++] = c;
     }
    break;
  }
```

```
    case SEGSTOP1:{
    recv(connection, &c, 1, 0);
    if(c == '#'){
       //printf("receive all the message \n");
       test = 1;
      }
      if(c != '#'){
       state = SEGRECV;
       buffer[index++] = '!';
       buffer[index++] = c;
    }
    break;
  }
    default:{
      //printf("The state is wrong ...\n");
  }
  }
  }
   int tt = seglost(segPtr);
   if(tt == 0){
   printf("the seg is complete ...\n");
   memcpy(segPtr, buffer, sizeof(seg_t));
   return 0;
}
else if(tt ==1){
   printf("the seg is partly lost ...\n");
     return 1;
}
```

重传机制的实现，主要是在服务器和客户端的接收线程中接收报文的时候，需要对 seg_lost() 的返回值进行判断。
同时重发机制的话也是在客户端和服务器端的 FSM 图中可以体现出来的。

四、实验结果及报文分析

```
//数据包丢失率为30%
#define PKT_LOSS_RATE 0.3
//SYN_TIMEOUT值，单位为纳秒
#define SYN_TIMEOUT 100000000
//FIN_TIMEOUT值，单位为纳秒
#define FIN_TIMEOUT 100000000
//stcp_client_connect()中的最大SYN重传次数
#define SYN_MAX_RETRY 15
//stcp_client_disconnect()中的最大FIN重传次数
#define FIN_MAX_RETRY 15
```

测试配置是：

服务器：114.212.190.188

客户端：114.212.190.186

为了方便观察发生数据包丢失时能否正确处理，如下测试是在丢包率为 30%, SYN / FIN 最大重传此时为 15 的情况下测试的：

2 次握手建立连接：

客户端发送 SYN 报文，然后收到服务器的 SYNACK 的流程

客户端部分：其中红色部分是发生报文丢失或者是段损坏的情况下，seg 是不完整的需要重传，打印出来的提示信息；

```
b101220151@csnetlab_2:~/lab10/client$ ./lab10_client

Client: +++  CLOSED ------>  SYNSENT

**********************
srcport: 87
destport: 88
type: SYN
**********************
the seg is complete ...

**********************
srcport: 88
destport: 87
type: SYNACK
**********************
Client: +++  SYNSENT ------>  CONNECTED
Client: Receive the SYNACK successfully ...
client connect to server, client port:87, server port 88

   Client: +++  CLOSED ------>  SYNSENT

   **********************
   srcport: 89
   destport: 90
   type: SYN
   **********************
   Resend the SYN ...
   the seg is partly lost ...
   Resend the SYN ...
   Resend the SYN ...
   the seg is complete ...

   **********************
   srcport: 90
   destport: 89
   type: SYNACK
   **********************
   Client: +++  SYNSENT ------>  CONNECTED
   Client: Receive the SYNACK successfully ...
   client connect to server, client port:89, server port 90
```

服务器部分

```
b101220151@csnetlab_4:~/lab10/server$ ./lab10_server
TCP: Server running ... wait for connections ...
TCP: Received request ...

Server: +++  CLOSED ------>  LISTENING
the seg is complete ...
Server: +++  LISTENING ------>  CONNECTED

**********************
srcport: 88
destport: 87
type:  SYNACK
**********************

Server: +++  CLOSED ------>  LISTENING
the seg is partly lost ...
the seg is complete ...
Server: +++  LISTENING ------>  CONNECTED

**********************
srcport: 90
destport: 89
type:  SYNACK
**********************
```

发送 FIN/FINACK 关闭 STCP 连接

客户端连接 1:

```
Client: +++  CONNECTED ------>  FINWAIT

**********************
srcport: 87
destport: 88
type: FIN
**********************
the seg is complete ...

**********************
srcport: 88
destport: 87
type: FINACK
**********************
Client: +++  FINWAIT ------>  CLOSED
Client: Receive the FINACK successfully ...
Client: Free the client tcb item successfully ...
```

客户端连接 2:

由于报文丢失重传了 4 此报文

```
Client: +++  CONNECTED ------>  FINWAIT

**********
srcport: 89
destport: 90
type: FIN
**********
Resend the FIN ...
Resend the FIN ...
Resend the FIN ...
Resend the FIN ...
the seg is complete ...

**********
srcport: 90
destport: 89
type: FINACK
**********
Client: +++  FINWAIT ------>  CLOSED
Client: Receive the FINACK successfully ...
Client: Free the client tcb item successfully ...
b101220151@csnetlab_2:~/lab10/client$
```

最后 2 个端口上的连接都被正常关闭掉，并且收到服务器的 FINACK 的确认，最后是释放 tcb 中对应的数据

服务器部分：
红色部分是接收是段发生丢失的情况下，打印提示信息：

```
**********
srcport: 90
destport: 89
type:  SYNACK
**********
the seg is complete ...
Server: +++  CONNECTED ------>  CLOSEWAIT

**********
srcport: 88
destport: 87
type:  FINACK
**********
the seg is partly lost ...
the seg is partly lost ...
the seg is partly lost ...
the seg is partly lost ...
the seg is complete ...
Server: +++  CONNECTED ------>  CLOSEWAIT

**********
srcport: 90
destport: 89
type:  FINACK
**********
Server: +++  CLOSEWAIT ------>  CLOSED
Server: +++  CLOSEWAIT ------>  CLOSED
b101220151@csnetlab_4:~/lab10/server$
```

六、思考问题

由于开始时对实验中一些字节考虑的不清楚，遇到了一些 bug，比如说线程在服务器和客户端扮演什么角色，其他 API 接口函数都是根据线程中的执行情况来做不同处理的，同时在实现 sip_recvseg()的时候遇到了不同的 BUG,主要是接收的时候一些状态流转的细节没有搞清楚。

然后是在实现客户端和服务器的 seghandler()函数的时候没有搞清楚其中的一些状态流转以及收发报文的逻辑关系，然后调了 8+小时的 BUG,还是很蛋疼的，跟有些同学一个下午搞定，感觉自己还是弱爆啦!!

不过偶会继续努力，有一天偶也会成为众人膜拜的大神的。

七、 实验的启示/意见和建议

实验花费了 3 个晚上+1 个早上+2 个下午 == 20+小时。