# 实验A.1：简单可靠传输层协议的实现

## 一：实验简介

在我们的课堂上已学习了传输层协议的基本原理，在此将尝试基于已学习的原理来编写程序实现传输层的相应功能。我们知道传输层之上是应用层，而应用层与传输层之间是Socket接口，传输层之下则是网络层。本实验要实现的目标是实现一个兼容TCP协议的简单TCP（Simple TCP）协议，使用STCP可以支持Socket的相应功能，并提供可靠的、面向连接的、有序的以及端到端全双工的交付机制。

为了降低实验难度，本实验已提供了一个框架，包括Socket接口、网络层、功能缺失的运输层以及客户端和服务端程序，并且已提供了除拥塞控制之外的所有功能。也就是说，我们只需实现传输层，让框架里的客户端和服务器工作在一个可靠的传输层。这样我们就无需关注丢包、重排序、重传以及超时等问题了。

## 二：实验环境

一台Linux系统的机器，需要包含gcc、Makefile工具

## 三：相关背景介绍

### 1. 各协议层代码结构

网络层【network.c】：为了简单起见，网络层是基于常规的传输层之上，被设计成可靠传输的。因此可以得知，本实验的协议均是用户态协议。

传输层【transport.c】：该层提供了最基本的框架，我们需要实现其中缺失的功能才能让客户端和服务器工作，这是本实验的核心。

应用层【ref\_client, ref\_server】：该层提供了仅有简单收发功能的客户端和服务端。

### 2. STCP数据结构

typedef uint32\_t tcp\_seq;

struct tcphdr {

uint16\_t th\_sport; /\* source port \*/

uint16\_t th\_dport; /\* destination port \*/

tcp\_seq th\_seq; /\* sequence number \*/

tcp\_seq th\_ack; /\* acknowledgment number \*/

#ifdef \_BIT\_FIELDS\_LTOH

u\_int th\_x2:4, /\* (unused) \*/

th\_off:4; /\* data offset \*/

#else

u\_int th\_off:4, /\* data offset \*/

th\_x2:4; /\* (unused) \*/

#endif

uint8\_t th\_flags;

#define TH\_FIN 0x01

#define TH\_SYN 0x02

#define TH\_RST 0x04

#define TH\_PUSH 0x08

#define TH\_ACK 0x10

#define TH\_URG 0x20

uint16\_t th\_win; /\* window \*/

uint16\_t th\_sum; /\* checksum \*/

uint16\_t th\_urp; /\* urgent pointer \*/

/\* options follow \*/

};

typedef struct tcphdr STCPHeader;

在本实验中只需要实现可靠传输功能，因此不需要处理所有的域，其中th\_sport, th\_dport, and th\_sum根据网络层的头填充，th\_urp在此则不使用。此外本实验对TH\_RST, TH\_PUSH, and TH\_URG这些标志位对应功能不作要求。

本实验中需要操作的域如下表所示。需要注意的是，所有的域都需要使用【htonl/ntohl 或者 htons/ntohs】来进行字节序的转换。

|  |  |  |
| --- | --- | --- |
| 域 | 数据类型 | 描述 |
| th\_seq | tcp\_seq | 与数据包相关的序列号 |
| th\_ack | tcp\_seq | ACK数据包的ack值 |
| th\_off | 4 bits | 数据包中数据起始位置的偏移量，单位为32bit的字 |
| th\_flags | uint8\_t | 各种控制位，如TH\_FIN, TH\_SYN等 |
| th\_win | uint16\_t | 接收方窗口大小的字节数，如发送数据包的主机愿意接受的未完成数据量。 |

### 3. 序列号

序列号是保证TCP各项功能的核心，STCP需要为应用程序数据流分配序列号，其方法是对字节进行编号。其中规则如下：

* 序列号以从0到255之间随机选择的序列号开始，依次对SYN标志，FIN标志和数据字节进行编号。
* SYN和FIN指示符都与序列空间的一个字节相关联，这使序列号确认机制可以处理SYN和不承载数据的FIN数据包。
* 传输层为每个连接管理两个流：传入和传出数据。这些流的序列号彼此独立。
* 应当为每个数据包都设置序列号。如果该数据包是纯ACK数据包（即无数据，并且未设置SYN / FIN标志），则应将序列号设置为下一个未发送的序列号。
* 应该可以随机地选择并设置初始序列号。

### 4. 数据报

以下规则适用于STCP数据包：

* 负载最大长度为536字节。
* 数据包头中的th\_seq字段包含有效负载中第一个字节的序列号。
* 只要应用程序中有可用数据，就会发送数据包。STCP不对首次传输时存在多个小包数据做任何优化，如将它们组合再发送。

### 5. ACK数据包

为了保证可靠传输，必须对收到的数据予以确认，对此STCP确认数据的规则是：

* 通过设置STCP的头的标志域里的ACK比特来确认数据。如果该比特被设置了，那么th\_ack将包含接收方所期望的下一次收到的数据字节序列号。
* 数据可能会随确认数据包一起传输。虽然STCP不要求生成这样的数据包，但要求需要处理这样的数据包。
* STCP在接收到数据包时将立即予以确认。

### 6. 滑动窗口

有两个窗口需要处理，即接收方和发送方的窗口。接收方窗口是接收方在任何给定时刻愿意接受的序列号范围。该窗口可确保发送方不会发送超出接收方处理能力的数据。

跟TCP一样，STCP也使用滑动窗口，发送方所发送数据的序列号受窗口值所限制。窗口在数据被确认后将进行“滑动”。其中窗口设置规则为：

* 本地接收方窗口的固定大小为3072字节。
* 拥塞窗口的固定大小为3072字节。STCP不做任何自适应拥塞控制。
* 发送方窗口大小由另一方通告的接收方窗口和拥塞窗口中的最小值所决定。
* 请注意，接收到的数据可能会越过当前接收方窗口的两端。在这种情况下，数据将被分为两部分，每一部分都需经过适当处理。
* 所发送数据不要超出发送方的窗口值。
* 所有窗口的第一个字节始终是最后一个确认数据的字节。例如，对于接收方窗口，如果最后一次确认的序列号是8192，则接收方愿意接受序列号8192到11263（8192 + 3072-1）的数据。

### 7. 重传

由于本实验STCP的网络层是在可靠的传输层之上实现的，因此不存在丢包和超时的情况，因此在本实验无需实现重传功能。

## 四：实验功能要求

### 1. 完善transport\_init()

本实验中唯一需要修改的文件是transport.c，其中包括transport\_init()、generate\_initial\_seq\_num()和control\_loop()需要修改。transport\_init()用于初始化传输层，开启一个循环control\_loop()来处理对端或者本地应用的任何数据。每个连接对应一个线程，当且仅当连接关闭时循环才退出。

完善内容：该函数中实现了基本框架，但是在其中需要根据参数【is\_active】来实现不同的功能，若是TRUE则生成序列号并发送SYN数据包，若是FALSE则等待连接的到来。

void transport\_init(mysocket\_t sd, bool\_t is\_active){

context\_t \*ctx;

ctx = (context\_t \*) calloc(1, sizeof(context\_t));

assert(ctx);

generate\_initial\_seq\_num(ctx);

ctx->connection\_state = CSTATE\_ESTABLISHED;

stcp\_unblock\_application(sd);

control\_loop(sd, ctx);

free(ctx);

}

### 2. 实现generate\_initial\_seq\_num()

完善内容：该函数比较简单，仅需对ctx->initial\_sequence\_num填充一个随机值即可。

static void generate\_initial\_seq\_num(context\_t \*ctx){

assert(ctx);

#ifdef FIXED\_INITNUM

ctx->initial\_sequence\_num = 1;

#else

/\*ctx->initial\_sequence\_num =;\*/

#endif

}

### 3. 完善control\_loop()

TCP有限状态机是传输层协议运行的保证，而函数control\_loop()则需要实现大多数“事件驱动”的STCP传输有限状态机。通过事件驱动，使用stcp\_wait\_for\_event()函数接收来自应用程序层的信号以接收数据或关闭连接，并接收来自网络层的数据包。control\_loop()的每次循环都将处理当前的挂起的事件集，并相应地更新有限状态机的状态。

static void control\_loop(mysocket\_t sd, context\_t \*ctx){

assert(ctx);

while (!ctx->done){

unsigned int event;

event = stcp\_wait\_for\_event(sd, 0, NULL);

if (event & APP\_DATA){

/\* 该应用程序已请求发送数据 \*/

}

/\* etc. \*/

}

}

完善内容：

* 处理对端发送过来的数据。
* 处理应用程序通过mywrite()函数传入的数据。
* 处理应用通过myclose()函数关闭Socket。
* 处理超时。

## 五：实验评分

由助教进行检查评分。