《Linux 操作系统》
大作业报告

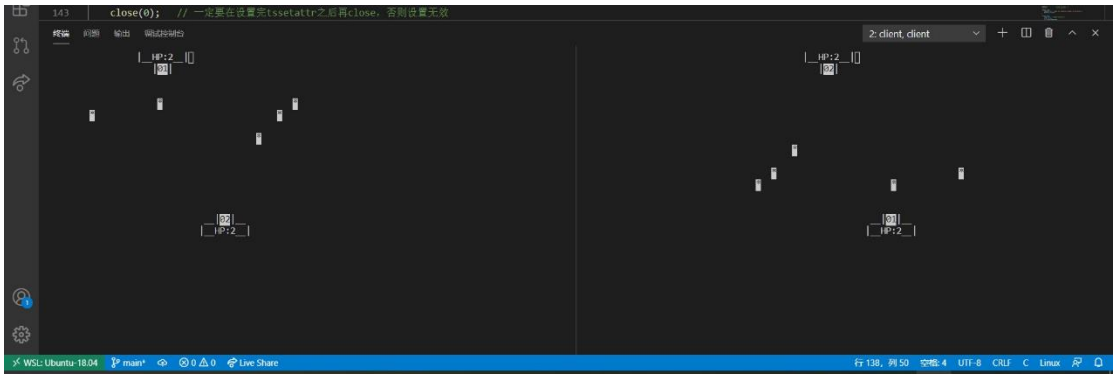
姓　　名			
班　　级			
实 验 名 称			
开 设 学 期			
开 设 时 间			
报 告 日 期			
评 定 成 绩			

东北大学软件学院

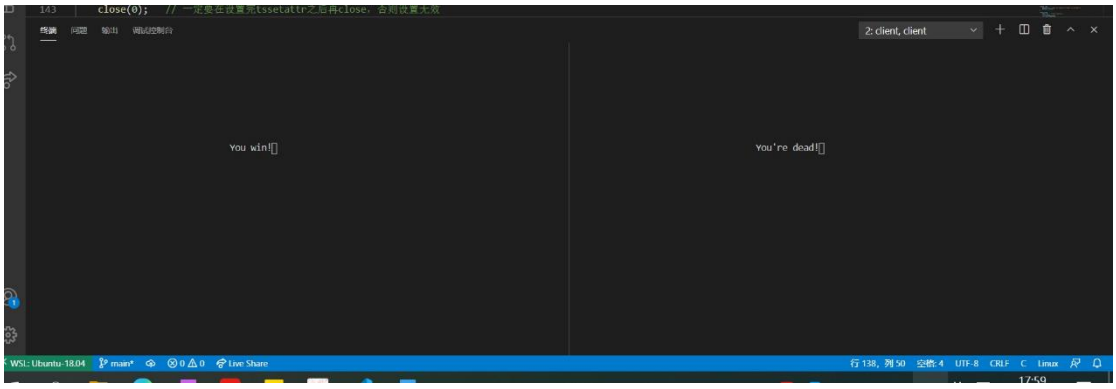
一、 程序使用说明

本次大作业的内容为一个在 Linux 环境下终端运行的在线多人小游戏。游戏内容为两个相对的平台互相发射子弹攻击，被击中者要扣生命值，生命值最先为 0 者被判定为失败。

游戏截图：



游戏结束时，胜者显示“You win!”，败者显示“You’re dead!”：



运行方式：

首先编译服务端程序：

```
cc server.c -o server
```

运行 server 参数格式为：“./server 端口号”，例如

```
./server 8080
```

然后编译客户端

```
cc client.c -o client -lcurses
```

在两个终端分别运行 client，格式为“./client 服务器 IP 端口号”，例如：

./client 127.0.0.1 8080

操作方式：WSAD 分别控制上下左右，空格键发射子弹。

二、 设计思路

1. 状态同步

目前游戏服务器的设计可以分为两类，分别是**帧同步**和**状态同步**。这里使用的是**状态同步**的思路。

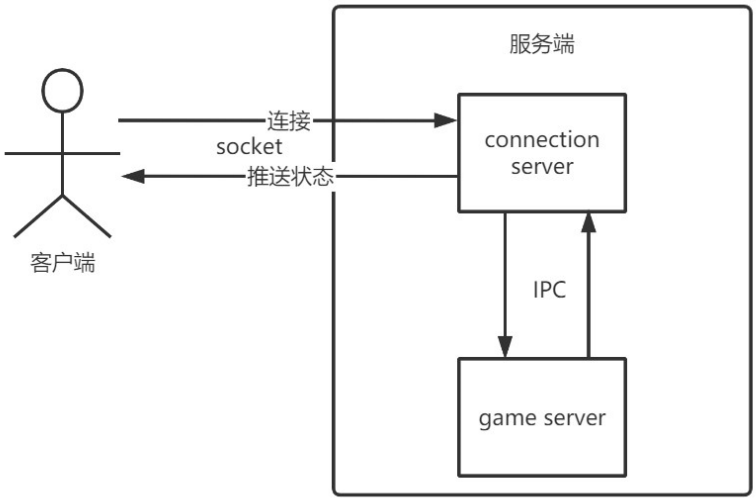
状态同步的含义为服务器向用户同步的是游戏的状态。玩家将所做的操作上传到服务器，由服务器统一计算，再将游戏状态（例如角色位置、生命值等）传回客户端。它的好处是，需要的网络带宽较小，而且在网络环境较差时，虽然客户间的画面可能不能完全同步，但是每个客户端上画面都是**按时序逻辑播放的**。此外，这样易于实现客户端和设计传输协议。

在本次大作业中，游戏状态指的是两个玩家平台的位置、子弹的位置、生命值和能量值。

2. 服务器设计

参考现实的游戏服务器设计，将游戏逻辑抽离出来单独作为一个进程运行，而 connection server 只负责对信息的接收和发送，connection server 和 game server 通过 IPC 的方式连接。具体服务器内进程间的关系如图所示：

（后来发现有 bug，但是全删掉有点可惜了。。）



3. 协议设计

在本游戏中使用的协议总体格式为“报文类型+内容”

具体实现如下：

格式	含义
POST 游戏状态的编码	推送游戏状态
OPT [W/S/A/D/Q]	用户操作。分别表示：上/下/左/右/退出
EXIT [0/1/2/3/4]	退出，同时说明退出原因。分别表示：主动/被动/拒绝访问/获胜/失败
WAIT	等待其他玩家

4. 客户端-服务端通信与状态转移

为使设计逻辑更为清晰，通过状态来分析两者的交流过程。这里的状态具体来说便是：

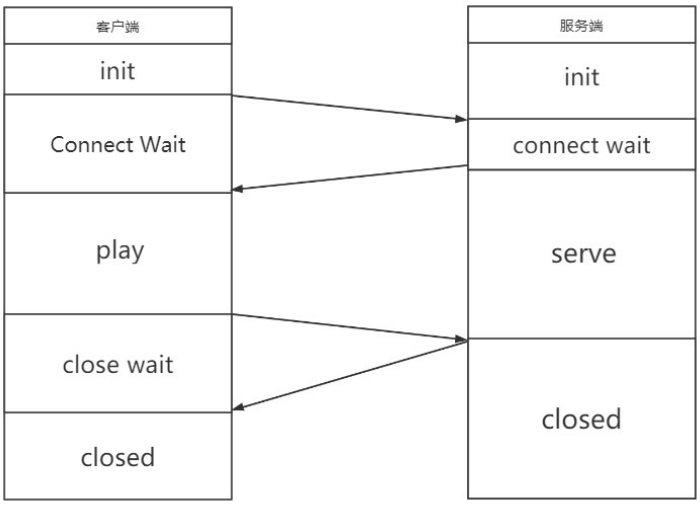
- 当且仅当特定的情况改变状态；

- 当且仅当处于特定的状态才允许实施某种行为。

在代码中使用枚举来表示：

```
enum client_phase{
    init, conn_wait, play, close_wait, closed
};
```

下图为正常情况下客户端和服务端之间的通信过程：



客户处于init状态时,向服务端连接并进入connect wait状态。服务端连接第一个客户端后进入connect wait状态，直到全部用户连接成功，进入serve状态，同时向客户端发送第一个POST报文，客户端接到第一个POST报文后进入play状态。在play状态客户会通过OPT报文发送操作，服务端在serve状态会向客户端发送POST更新游戏状态。最后客户通过OPT Q提出退出，服务端向所有客户发送EXIT推出指令。或者游戏结束，服务器通知客户端游戏结果。

5.

三、 采用技术

1. Linux网络编程

充分利用了Linux相关系统调用来实现。

下图中为初始化 socket 连接，如果绑定不成功，向像消息队列中传入关闭的信息。

```c
void init_socket(int *listen_sock, char port[], int msgqid){
    struct sockaddr_in addr;

    *listen_sock  =  socket(AF_INET,  SOCK_STREAM, 0);

    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(port));
    addr.sin_addr.s_addr = INADDR_ANY;
    if(bind(*listen_sock, (const struct sockaddr *)&addr, sizeof(struct sockaddr_in))==-1){
        struct operation opt_st;
        opt_st.dtype=0;
        if(-1==msgsnd(msgqid, &opt_st, sizeof(struct operation), IPC_NOWAIT)){…
        perror("cannot bind");
        exit(1);
    }

    listen(*listen_sock, 1);
}
```

2. I/O 复用（epoll）

这里在服务端和客户端均使用了 epoll 来同时处理多个文件描述符。

在服务端，主要通过 epoll 同时与两个客户端进行通信：

等待两个玩家全部连接成功：

```c
void wait_players(int epollfd, struct epoll_event *events,\
                  int listen_sock, int *conn_sock, enum server_phase *phase){
    int nfds;
    struct epoll_event ev;
    int i;

    for(i=0;i<2;i++){
        char wating_msg[10] = "WAIT\0";
        nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);

        if (nfds == -1) {…
        conn_sock[i] = accept(listen_sock, NULL, NULL);
        // 另一个客户端未连接，第一个客户端已断开：close并重连（i--）
        if(conn_sock[i] == -1 && errno == EAGAIN){…
        else if(conn_sock[i] == -1){…
        setnonblocking(conn_sock[i]);
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = conn_sock[i];
        if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock[i], &ev) == -1) {…
        if(-1==write(conn_sock[i], wating_msg, strlen(wating_msg))){…
        printf("connected %d\n",conn_sock[i]);
        *phase = conn_wait;
    }
}
```

处理客户端的请求：

```
while(phase != closed){
    int i;
    nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);

    if (nfds == -1 && errno != EINTR) {
        perror("epoll_pwait");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < nfds && phase != closed; ++i) {
        if (events[i].data.fd == listen_sock) {
            int conn_sock = accept(listen_sock, NULL, NULL);
            char refuse_msg[10] = "EXIT 2\0";
            if(-1==write(conn_sock, refuse_msg, strlen(refuse_msg))){
                perror("write");
                exit(EXIT_FAILURE);
            }
        } else {
            handle_msg(events, sts ,i,&phase);
        }
    }
}
return 0;
```

3. 通过 curses 绘制图形

这里通过 curses 实现游戏图形界面的绘制。在代码中 graph.h 文件中是专门用于绘制图形的函数。

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>

#include "data.h"

int draw_self(int l,int c, int hp, int ep){…

int draw_other(int l, int c, int hp, int ep){…

void draw_bullets(const struct pos *bullets, const int size){…

int erase_plane(int l,int c){…

int draw_status(struct status *sts, struct status *old, int log_fd){…

void show_end(const char *msg){…
#endif
```

绘制敌方平台：

```c
int draw_other(int l, int c, int hp, int ep){
    char hp_msg[100];
    char ep_msg[100];

    if(l < 0 || l >= LINES){…
    if(c < 0 || c >= COLS){…
    sprintf(hp_msg, "|__HP:%1d__|", hp);
    sprintf(ep_msg, "%02d", ep);

    move(l+1,c);
    standend();
    addstr("   |");
    standout();
    addstr(ep_msg);
    standend();
    addstr("|  ");
    move(l,c);
    addstr(hp_msg);
    // addstr("|________|");
    standend();

    // refresh();          /* update the screen    */
    return 0;
}
```

绘制图形最重要的是根据整个画面的长宽计算游戏对象的位置。在 data.h 中通过宏定义实现了一些常用的位置计算模板：

```
#define MAX_HP 2
#define MAX_EP 2
#define RELOAD_ITR 3

#define SCREEN_C 80
#define SCREEN_L 16

#define PL_HIGHT 2      // DO NOT CHANGE
#define PL_WIDTH 10      // DO NOT CHANGE
#define BLT_HIGHT 1     // DO NOT CHANGE
#define BLT_WIDTH 1     // DO NOT CHANGE

#define PL_C_UPR (SCREEN_C - PL_WIDTH)              // upper band
#define PL_L_UPR (SCREEN_L - PL_HIGHT)
#define PL_C_LWR 0                                 // lower band
#define PL_L_LWR 0

#define BLT_C_UPR (SCREEN_C - BLT_WIDTH)            // bullet coor upper band
#define BLT_L_UPR (SCREEN_L - BLT_HIGHT - PL_HIGHT)
#define BLT_C_LWR 0                                // lower
#define BLT_L_LWR PL_HIGHT

#define CENTRAL_SYMMETRY(V, LOW, UP) ((LOW) + (UP) - (V))
```

4. 对终端输入的设置

正常的情况下(规范模式下)，键盘输入是需要在输入一段字符后，按下回车键，这些字符才会传到进程中，在此之前驱动只是在屏幕中返送输入的字符。由于游戏操作的需要，需要取消规范模式和返送。在游戏结束时需要再将设置恢复。

开始时的设置：

```
}
tcgetattr(0,&init_tty);
tcgetattr(0,&new_tty);
new_tty.c_lflag &= ~(ECHO|ICANON);
new_tty.c_cc[VMIN]=1;
new_tty.c_cc[VTIME]=0;
tcsetattr(0,TCSAFLUSH,&new_tty);
}
```

如上图开始时保存了初始的设置，结束时将 init_tty 传入即可：

```c
void std_exit(int flg, int sock_fd, int epollfd){
    close(sock_fd);
    close(epollfd);
    init_tty.c_cc[VMIN]=1;
    tcsetattr(0,TCSAFLUSH,&init_tty);
    close(0);   // 一定要在设置完tssetattr之后再close，否则设置无效
    clear();
    endwin();           /* reset the tty etc    游戏退出时调用*/
    exit(flg);
}
```

5. 间隔计时器和信号处理

这里在两处使用了间隔计时器。首先在 game server 使用间隔计时器控制游戏的时间周期，例如周期性的子弹移动、弹药填充、生命值检查等；另一个计时器用来控制刷新率，即向客户推送状态的频率。当用户的操作频率过快时，服务端只会保持最大刷新率不会继续上升。这样保证了游戏不会因为网络带宽有限而严重不同步，甚至崩溃。

Connect server 的计时器：

```c
> void post_to_client(int signum){···

void init_internal_fresh(){
    struct itimerval itval;
    if(SIG_ERR==signal(SIGALRM, post_to_client)){
        perror("signal");
        exit(EXIT_FAILURE);
    }
    itval.it_value.tv_sec = 1;
    itval.it_value.tv_usec = 0;
    itval.it_interval.tv_sec = 1;
    itval.it_interval.tv_usec = 0;
    if(-1 == setitimer(ITIMER_REAL, &itval, NULL)){
        perror("setitimer");
        exit(EXIT_FAILURE);
    }
}
```

Game server 中的计时器：

```c
> void move_bullets(int signum){···

  void init_mv_bullets_timer(){
      struct itimerval itval;
      if(SIG_ERR==signal(SIGALRM, move_bullets)){
          perror("signal");
          exit(EXIT_FAILURE);
      }
      itval.it_value.tv_sec = 2;
      itval.it_value.tv_usec = 0;
      itval.it_interval.tv_sec = 2;
      itval.it_interval.tv_usec = 0;
      if(-1 == setitimer(ITIMER_REAL, &itval, NULL)){
          perror("setitimer");
          exit(EXIT_FAILURE);
      }
  }
```

四、 遇到的问题和解决办法

1. 代码逻辑混乱

例如有时为了进行一个简单的指令，需要通过多个变量的值判断能否执行该指令。这样容易出错且代码混乱、可读性差。

通过枚举量抽象表示客户端和服务端状态，可以解决这个问题。只在特定情况改变它的值，并用它来判断。

2. 游戏结束后无法停留在预设的结束画面

正常情况下，使用 getch 函数可以等待用户输入以暂停画面，但在非规范模式下不起作用，需要在恢复键盘输入设置后再使用 getch。

3. 游戏结束后客户端终端无法恢复正常

在使用 tcsetattr 设置之前不可将对应的文件描述符关闭。

```c
void std_exit(int flg, int sock_fd, int epollfd){
    close(sock_fd);
    close(epollfd);
    init_tty.c_cc[VMIN]=1;
    tcsetattr(0,TCSAFLUSH,&init_tty);
    close(0);   // 一定要在设置完tssetattr之后再close，否则设置无效
    clear();
    getch();
    endwin();           /* reset the tty etc    游戏退出时调用*/
    exit(flg);
}
```

除此之外，更重要的是分析所有可能的结束节点，包括正常的和异常的结束，并在结束前保证将终端复原。

当服务端异常关闭时(std_exit 为封装的在结束前执行的函数)：

```
memset(buf, 0, sizeof(buf));

if(-1==(nread = read(sock_fd,buf,sizeof(buf)))){
    perror("read");
    std_exit(EXIT_FAILURE,sock_fd,epollfd);
}
if(nread == 0){
```

```
while(phase != closed){
    int nfds = epoll_wait(epollfd, events,2,-1);
    int i;

    if(nfds == -1){
        if(errno == EINTR) continue;
        perror("epoll_wait");
        std_exit(EXIT_FAILURE,sock_fd,epollfd);
    }
    for(i=0;i<nfds;i++){
```

上述下图在客户端中断（例如 Ctrl+C）时也能起作用。

4. 游戏画面绘制经常出错

通过定义宏来规范游戏画面的计算，减少出错。

```
#define MAX_HP 2
#define MAX_EP 2
#define RELOAD_ITR 3

#define SCREEN_C 80
#define SCREEN_L 16

#define PL_HIGHT 2      // DO NOT CHANGE
#define PL_WIDTH 10      // DO NOT CHANGE
#define BLT_HIGHT 1     // DO NOT CHANGE
#define BLT_WIDTH 1     // DO NOT CHANGE

#define PL_C_UPR (SCREEN_C - PL_WIDTH)              // upper band
#define PL_L_UPR (SCREEN_L - PL_HIGHT)
#define PL_C_LWR 0                              // lower band
#define PL_L_LWR 0

#define BLT_C_UPR (SCREEN_C - BLT_WIDTH)            // bullet coor upper band
#define BLT_L_UPR (SCREEN_L - BLT_HIGHT - PL_HIGHT)
#define BLT_C_LWR 0                             // lower
#define BLT_L_LWR PL_HIGHT

#define CENTRAL_SYMMETRY(V, LOW, UP) ((LOW) + (UP) - (V))
```

5. 服务端在处理指令的过程中，同时以固定的频率移动子弹

使用间隔计时器自动地以固定周期执行某些指令。同时在代码中，间隔计时器地设置类似于一种配置，只在程序开始时设置即可，不影响代码主体地其他逻辑。

设置间隔定时器和相应的信号处理函数：

```
> void move_bullets(int signum){···

  void init_mv_bullets_timer(){
      struct itimerval itval;
      if(SIG_ERR==signal(SIGALRM, move_bullets)){
          perror("signal");
          exit(EXIT_FAILURE);
      }
      itval.it_value.tv_sec = 1;
      itval.it_value.tv_usec = 0;
      itval.it_interval.tv_sec = 1;
      itval.it_interval.tv_usec = 0;
      if(-1 == setitimer(ITIMER_REAL, &itval, NULL)){
          perror("setitimer");
          exit(EXIT_FAILURE);
      }
  }
```

在程序开始时设置（init_mv_bullets_timer）：

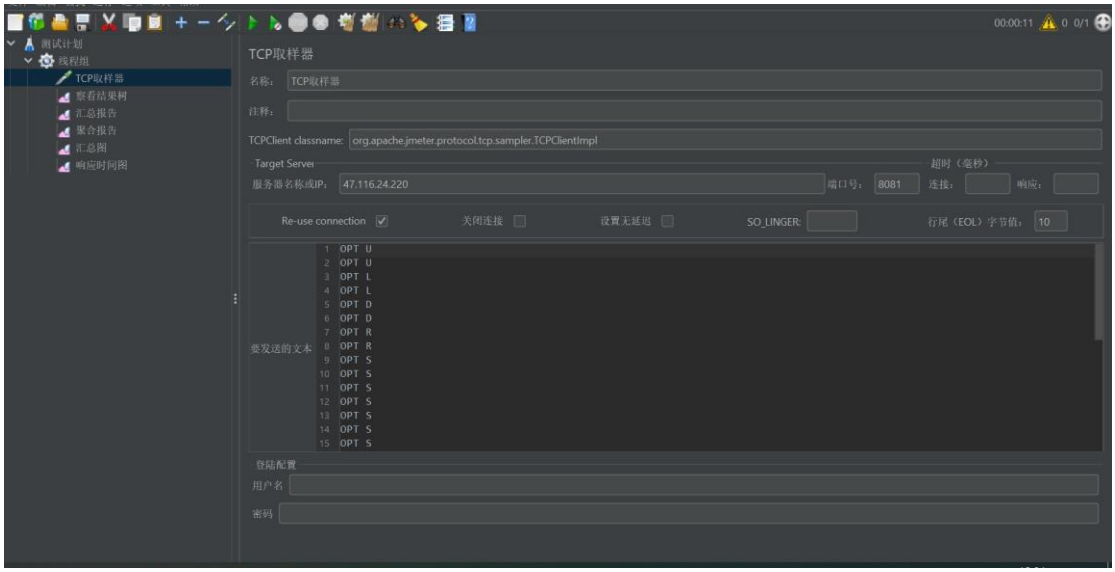
```
int main(int ac, char *av[]){
    int sockfd;
    struct epoll_event events[MAX_EVENTS];
    int listen_sock, nfds, epollfd;

    if(ac < 2){
        fprintf(stderr,"usage: portnumber needed.");
        exit(EXIT_FAILURE);
    }

    init_socket(&listen_sock,av[1]);
    init_epoll(&epollfd,listen_sock);
    wait_players(epollfd, events,listen_sock,player_fds,&phase);
    init_status_world(sts, player_fds);
    init_mv_bullets_timer();
```

五、 选做部分和加分点

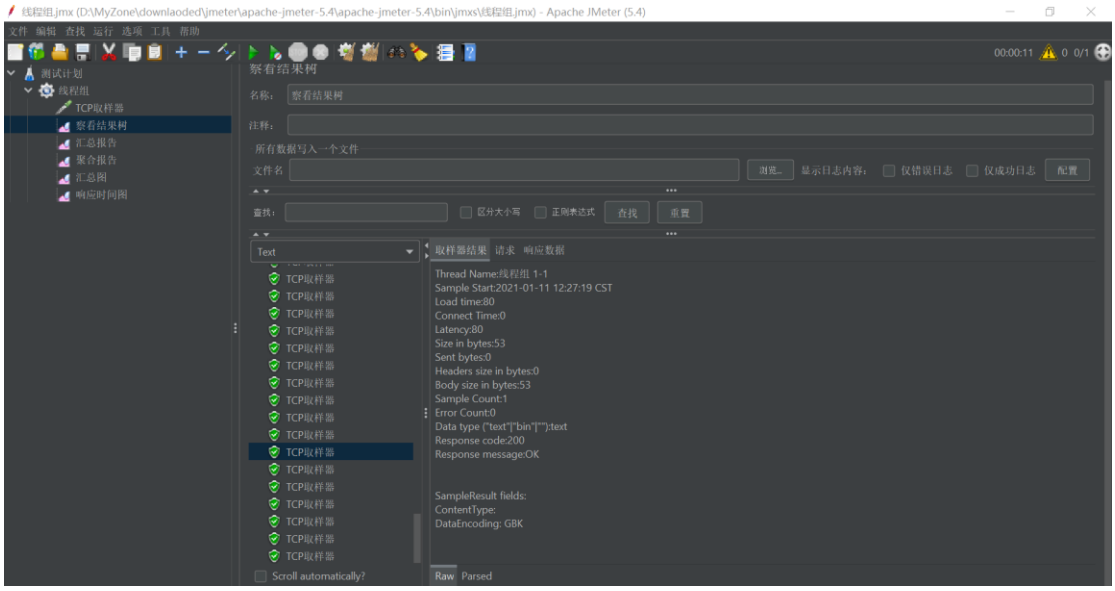
1. 压力测试

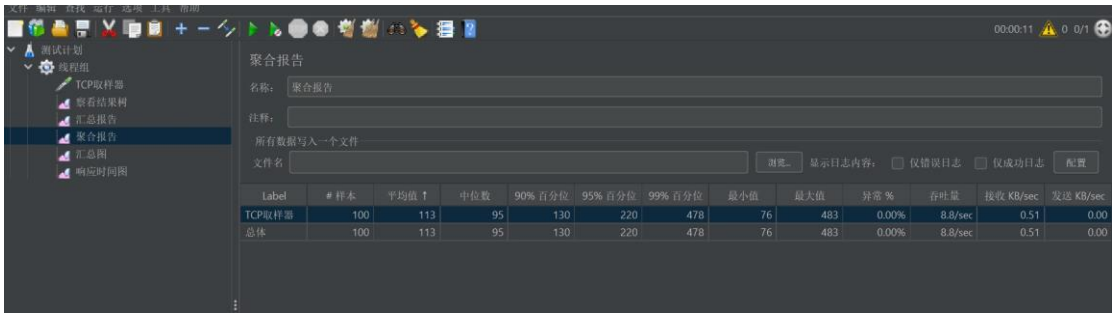
测试工具：JMeter

取样器设置



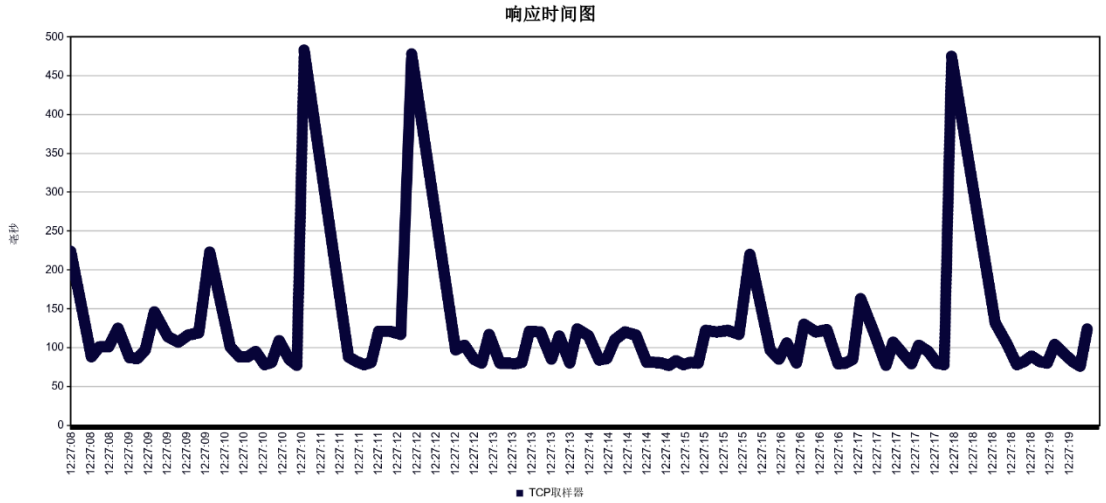
测试结果截图：





上图显示了测试的定量结果，包含响应时间地最大值、最小值、平均值、中位数和各分位点的值，此外还包括异常率、吞吐量、接收/发送速度等。

响应时间波动图：

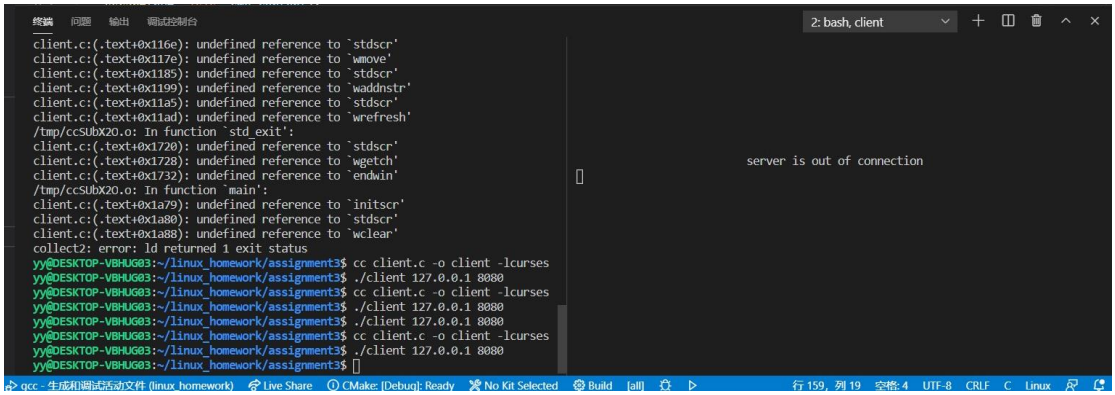


2. 健壮性测试

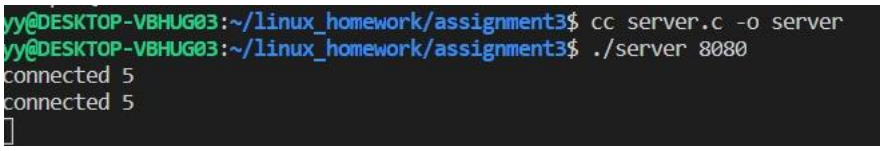
测试用例	预期结果
服务端异常关闭	客户端全屏显示服务器断开提示，等待用户输入后退出
客户端异常断开	服务端断开，另一个客户端显示服务器断开
第二个客户端未连接时第一个客户端已断开	服务端释放文件描述符，等待重新练接。

测试一结果：

测试二结果：



测试三，服务端在重新连接时重用了文件描述符：



评价表格

考核标准	得分
（1）正确理解和掌握实验所涉及的概念和原理（20%）；	18
（2）按实验要求合理设计数据结构和程序结构（20%）；	18
（3）认真记录实验数据，原理及实验结果分析准确（20%）；	18
(4)实验过程中,具有严谨的学习态度和认真、踏实、一丝不苟的科学作风(10%);	9
（5）所做实验具有一定的创新性（10%）；	9
（6）实验报告规范（20%）。	18