# 《网络文字游戏对战实验》实验报告

**学号：201220131**

**姓名：胡增杰**

## 1.实验目的

编写多用户的网络文字对战程序，服务器和客户端的功能需要满足实验手册的要求。

## 2.协议设计

协议内容已经在协议的说明文档中给出，不在此赘述。

下面主要说明一下自己的协议设计思路和历程：

首先是分析实验的需求，可以发现客户端和服务器需要交互很多的信息，而且服务器有时还需要处理多个客户端的数据，然后再根据处理的具体内容返回给客户端信息。

最初自己的设计思路是设计很多个包，比如表示请求登录和退出的包，表示请求对战和回应的包，表示对战内容的包，表示对战结果信息的包，通知用户登录和退出信息的包等。但是这种实现中，接受数据的包就变得很不统一，感觉很麻烦。因此又联想到实验二中解析的那个数据包的第一个字段是两个的type字段，而且在实验二的协议中也使用type字段就包含了很多的交互的信息内容本身，因此自己也仿照这种思路，只使用一个数据包，利用type字段来表达不同的信息，这样再结合具体的实验要求，就设计出了现有的这份协议。

## 3.编程实现

使用到的数据报文和其他数据结构已经在协议的说明文档中给出，不再赘述。

### 3.1客户端

首先设置有三个全局的状态量：

- login，为1表示用户已经登录，为0表示用户还没有登录。
- battling，为1表示用户正处于对战中，为0表示用户没有处在对战中。
- needaccept,为1表示用户接收到了来自别的用户的对战请求，用户需要输入yes或no进行答复，为0表示用户没有接收到来自别的用户的对战请求，不要进行答复。

这三个变量都有对应的锁，分别用来对这三个状态量进行并发读写。

同时还有两个全局变量curusername和rivalsname,分别用来存储当前登录用户的名称和与当前用户对战的用户的名称（当前用户处于对战状态才有意义）。

使用两个线程，handle_input和handle_network,

- handle_input专门处理标准输入，接受来自终端的标准输入，再根据login，battling，needaccept的值来进行处理相应的数据交互和异常处理。
- handle_network,专门负责接受来自服务器端的数据包，再根据不同的type字段进行相应的处理，并要是更改login,battling,needaccept这三个状态量和curusername，rivalsname数据量。

设置login，battling，needaccept这三个状态量的原因就在于，自己的程序是线程并行程序，收到数据包和对数据包作出回应的这两个过程是并行和分离的，因此需要这三个状态量来指示程序现在处于什么状态，应该做出怎样的回应，这样才能使程序运行出的逻辑是正确的。

## 3.2服务器端

同样使用多线程，对于每个和服务器建立TCP连接的客户端，就创建一个线程recv_handler对其进行处理，在每个线程中，根据接收到的包的type字段进行相应处理。另外还有一个battle_worker的线程处理对战信息的数据包。

使用到三个数据结构：

**用户列表**：

```
typedef struct UserInfo{
    int connectfd; //to find the connection bewteen server and this user
    User user; //state of user contains whether this UserInfo is used or not
} UserInfo;
#define USERLIST_SIZE 100
UserInfo userlist[USER_NUM];
pthread_mutex_t userlist_mutex;
```

保存已经建立连接的用户，connectfd是该用户与服务器的连接套接字，user就是登录的用户的姓名和状态，User的结构如下：

```
#pragma pack(1)
typedef struct User{ //20
    char name[NAME_SIZE];
    uint8_t state; /*
        0-not used, for example, a user has logined once but logouted.
        1-vacant
        2-battling
    */
    //uint8_t blood;
} User;

```

state为0表示这个用户没有在线，1表示在线没有对战，2表示在线且对战中。

对应的互斥锁userlist_mutex是为了实现对userlist的并行读写。

**对战关系列表**

```
typedef struct BattleInfo{
    uint8_t used; //0-not used, 1-used, for battlelist
    //uint8_t round; //the number of rounds each side has engaged
    char username1[NAME_SIZE];
    int sockfd1;
    uint8_t blood1;
    char username2[NAME_SIZE];
    int sockfd2;
    uint8_t blood2;
} BattleInfo;
#define BATTLELIST_SIZE 100
BattleInfo battlelist[BATTLELIST_SIZE];
pthread_mutex_t battlelist_mutex;
```

每一个BattleInfo，都表示已经建立对战关系的两个用户，used字段用来指示在battlelist中这个下标位置的数据是不是空，每次在battlelist中存放BattleInfo时都采用first-fit的方法，找到第一个为空的位置进行存放。username,sockfd,blood分别表示两个用户的名字、与服务器的连接套接字、剩余血量。

对应的互斥锁battlelist_mutex用来并发读写battlelist。

**对战数据包缓冲区**

```
typedef struct BattlePkt{
    uint8_t used; //0-not used, 1-used, for battlebuffer
    PKT pkt;
} BattlePkt;
#define BATTLEBUFFER_SIZE 100
BattlePkt battlebuffer[BATTLEBUFFER_SIZE];
pthread_mutex_t battlebuffer_mutex;

int pktcnt = 0;
pthread_mutex_t pktcnt_mutex;
pthread_cond_t pktcnt_cond;
```

因为在对战过程中，涉及两个客户端的数据交互，但是服务器分别与这两个客户端建立了连接，并使用两个相同的线程分别处理与两个客户端的连接，这两个连接是分离的，因此需要处理对战信息的过程不能放在上面的处理连接的线程中，否则会被重复处理，因此需要将对战信息的数据包单独分离出来用一个线程来处理。

凡是遇到服务器接收到客户端的type为0x0301的数据包，也就是包含对战选择信息的数据包，都将他们原封不动的放在battlebuffer中，并使用一个信号量pktcnt_cond来通知线程battle_worker缓冲区中有包可供处理。

值得特别提出的就是battle_worker的处理过程，它首先根据battlelist，因为battlelist中存放的是已经建立对战关系的用户组，因此遍历battlelist，对其中每一个队长用户组，在battlebuffer中根据包的srcname和dstname找到这一对战组的交战报文，然后进行比较处理，再根据保存的sockfd回发给对应的客户端。在自己的实现中，battlebuffer的存放也是使用first-fit，其找到第一个空闲的位置进行存放，寻找过程也是first-fit，因此用户作战过程中用户其实可以连续多次输入自己的选择，服务器会按照输入顺序进行处理。

# 4.遇到的困难

个人感觉遇到的困难是相当之多！

## 4.1 数据报文的设计

遇到的问题在上述的【2.协议设计】已经说明，这个还是相当比较好解决的问题。

## 4.2 服务器的跨链接处理

自己所谓跨链接处理，指的是，如两个用户和服务器分别建立一个连接，但是当请求对战和对战信息处理的时候，服务器需要分别从两个连接中读取信息，然后进行处理，如果把这个处理过程放在处理每个连接处理线程中，就会发生重复。因此需要分离，并记录每个连接的套接字，这样在别的连接处理线程中发送给另外的连接数据。

## 4.3 并行化程序的编写

因为这个实验其实是自己实际意义上写的第一个并行程序，所以刚开始写代码的过程中有很多串行的思想遗留，比如，自己最初对于客户端请求对战的处理方法就是，在客户端的向服务器发送请求报文，然后等待服务器发送回应报文，服务器收到有请求对战的报文，就通知给目的client，然后等待目的client发送回应报文才能发回给请求方，自己最初把这些过程全部都写在一段代码中，导致思考不清楚怎么正确实现并行化。

## 4.4 客户端程序的输入分配

客户端的输入大致可以分为三种类型，打开程序的login,logout,veiw,request这类指令，对于别的用户的对战请求的回应（yes/no)，对战过程中的输入选择（r,s,p), 最初自己对于这三个输入的处理是放在不同线程中的，结果导致不同的线程会争夺标准输入，无法判断应该将输入分配给哪个线程，哪怕使用互斥锁也不能控制好这个选择，因为等待输入是很慢的，但是互斥锁的上锁和解锁相对于人为的输入是很快的。

自己最后的解决办法就是将标准输入的过程全部都放在一个线程中，并且使用login,battling,needaccept来判断输入对应于那种情况，另外一个线程只接受服务器的数据包，这样就很好的完成了实验的功能。

# 5. 实验总结

这个实验花费了大量的实验，但是从零开始，自己一步步从设计报文格式和协议内容，然后编程实现过程遇到的一系列线程程序编写、如何通过并行程序实验自己想要的功能等等的问题，代码也重构了三四次，纠正了很多对线程程序理解不到位的地方，感觉收获颇丰。