Fancy Quiz Server
Part I. Architecture and Design

Explains your key data items and data structures, and how your server is going manage them to deliver the quizzes to the groups, and give good, fair service to the clients. Focus on technical details. This should help you plan your own work and serve as a blueprint for your program. This doesn't mean you shouldn't program during the first week.

Implementation of the server for group quizzes will combine as required using of multiplexing and threads. Since the main goal of any useful application with client-server architecture is to be able to handle multiple client requests at the same time it is necessary to use one of the above techniques. I decided instead of creating a separate thread for each connecting client to use select system call and to use an example code for echoserver as a starting point. When a new client arrives it is added to the array of ready file descriptors, so no blocking is ensured.

In order to organize my codes I will explain firstly main data structures that will be used and then how each separate command will be handled.

Data structures:

Essentially they are two main data items involved in this program: group and client. Therefore for each object two new data types were created, as shown in the parts of code bellow.

```
typedef struct{
    int score;
    char name[30];
    int groupName[30];
    int isLeader;
}client;

typedef struct{
    char topic[30];
    int size;
    int currSize;
    char groupName[30];
    client *members[1010];
}group;
```

To store all groups and clients pointer to arrays were created:

```
client *clients[1010];
group *groups[32];
```

Each time when a new group or client is added the memory is dynamically allocated for each element using socket number, which is a file descriptor as an index:

```
clients[fd] = (client*)malloc(sizeof(client));
groups[fd] = (group*)malloc(sizeof(group));
```

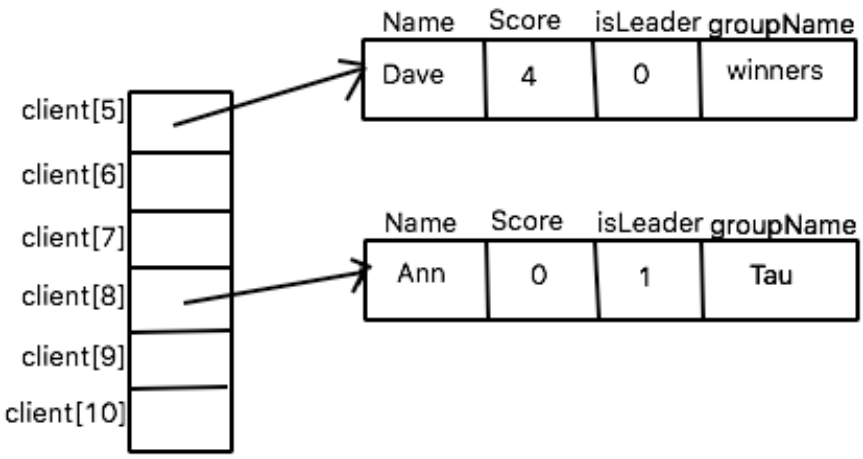

Figure 1. Array diagram for clients

As it is illustrated in Figure 1 each element in clients[ ] points to one client which has attributes: Name, Score, Leader flag and the name of the group to which the client belongs.

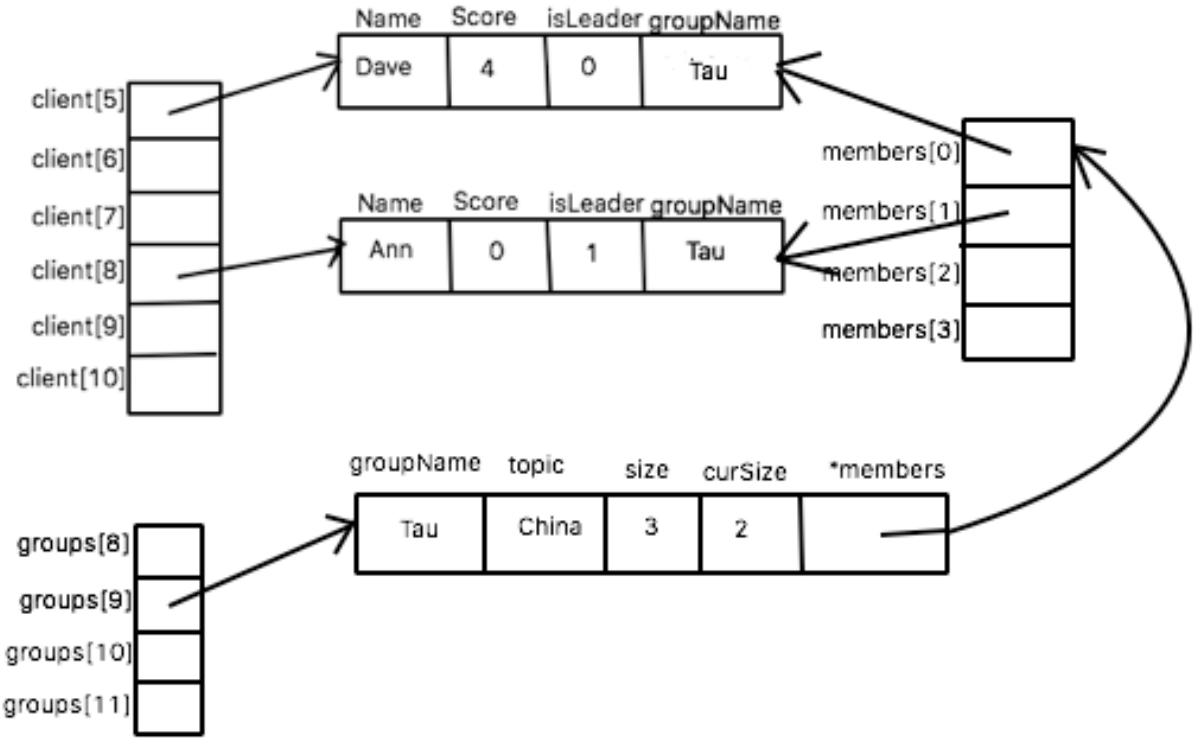

Figure 2. Array diagram for groups/clients

The similar diagram is shown in Figure 2 for groups[ ] array but the last attribute of the groups has client type and it is an array of pointers to the same memory location as from clients[ ] array.

Commands from client:

As it was mentioned above in this implementation multiplexing is used to handle a new connection with the client, however to start a quiz in different groups at the same time, threads will be used. Therefore one thread is created per one group.

When the client connects to the server the new element in clients[ ] is allocated. Then the server is expecting the answer from the client. Handling of different commands will be done within if-else blocks, comparing the first token in the client answer's string to existing commands:

1) GROUP command.

Firstly the third token in the string will be checked to ensure that the groupname is not already in use. I am planning here to call the function, which will iterate through the array of groups and return 1 if the groupname is used by other group. If it not there, the client (leader) creates a new element in groups[ ] array, adds topic, groupname and groupsize. Current groupsize is set to 1. Thread for group is started and quiz is played in the thread.

2) JOIN command

Client provides name, groupname to which it wants to join and score attribute is set to zero. At the same time a new client should be pointed by the element in members[ ] in that group. To find the group I am planning plan to call here function findByGroupname(), which will be searching from the groups[ ] the group with that name and return -1 if no matching group is found. NOGROUP message is sent. Otherwise, it will return the socket number of the group leader. Before updating the current size and adding groupname attribute to the client. If the currSize == size then no other clients can be accepted and FULL message is sent.

```
    int findByGroupname(char *grname){
    printf("Join??");
    fflush( stdout );
     for(int i = 0; i < 32; i++){
         if(groups[i] == NULL){ //skip not used socket
             continue;
         }
         if(strcmp(groups[i]->groupName, grname) == 0){
             printf("Found!\n");
             fflush( stdout );
             return i;
     }
}
     return -1;
}
```

Though, there is a condition that no complicated looping structures are allowed in the main thread are allowed, therefore the way to solve this issue should be rethought.

3) GETOPENGROUPS command

The list of all open group will be send iterating through groups[ ] array.

4) LEAVE command

Groupname attribute of the client will be cleaned and the member[ ] is also will be cleaned in groups[ ]. Current size will be decremented. To find the member findByGroupname is again called.

5) CANCEL command

If the quiz has not yet started group[fd] will be cleaned where fd is a socket of the leader.

6) QUIZ command

Each quiz is handled inside the thread. When the message SENDQUIZ is sent each thread expects that the client will start to send the quiz probably part by part. Therefore client's response will be read in the loop until the quiz size will be reached. Before the text of the quiz its size sent in the message QUIZ|size|text…..

I will have a variable remSize, which is the quiz's size minus the length of the part of the quiz text sent by the client for the first time.

```
int remSize;

        if(strcmp(splitted[2], NULL) == 0){
            remSize = quizSize - strlen(splitted[2]);
            printf("%s\n", splitted[2]);
            fwrite(splitted[2] , 1 , strlen(splitted[2]) , fp );
        }else{
            remSize = quizSize;
        }
```

So while remSize is greater than zero server will read from client and each time write the client's response to the temporary file.

Afterwards as it was done in the previous homework by fgets the content of the temporary file will be written into array of lines and quiz will be started sending the participants one question per time.

Concurrency inside the quiz will be provided this time by using multiplexing, i.e. the new array of descriptors to hold the participating clients will be created and the timer value will be set to 1 minute. If the client does not answer the same leaving procedure will be repeated.

Messages send by the server:

7) ENDGROUP message

This message is sent by the server to the clients in the group if leader sends CANCEL message or when the quiz has ended. Group[ ] item is cleaned as well as groupname attribute for all clients who were in this group.

8) OPENGROUPS message

This command is just reply for GETOPENGROUPS command.