QuizzGame: Implementarea unui joc de tip quizz

Marian Cosmin Constantin

Universitatea Alexandru Ioan Cuza

Rezumat Acest raport prezintă o imagine de ansamblu asupra proiectului QuizzGame (B), incluzând implementarea, funcționalitatea și o demonstrație a utilizării acestuia. Voi discuta tehnologiile aplicate, structura aplicației, detalii cheie de implementare și posibile îmbunătățiri viitoare.

1 Introducere

În acest raport voi prezenta o imagine de ansamblu a proiectului QuizzGame (B) [Propunere Continental]. Voi discuta despre implementare, funcționalitate și voi prezenta un exemplu de utilizare. Am ales acest proiect pentru că mi s-a părut interesantă ideea de a crea un quizz cu un timp limită pentru fiecare răspuns și mi-am dorit să explorez cum se poate construi un astfel de sistem. De asemenea, am în vedere adăugarea mai multor funcționalități și opțiuni pentru quiz în versiunea finală a proiectului.

2 Tehnologii aplicate

În implementarea proiectului, am decis să folosesc un server TCP concurent. Concurența este realizată folosind multithreading, care este o opțiune potrivită pentru acest tip de aplicație. Am ales TCP pentru a asigura transmiterea informațiilor fără pierderi de date, lucru esențial într-un quiz, unde integritatea întrebărilor este crucială. Am folosit mutexuri pentru protecție și pentru a evita condițiile de tip "race condition". Pentru a implementa un cronometru pentru fiecare întrebare, am folosit o structură de tip timeval în cadrul funcției select, pe care am denumit-o timpr, și care poate fi setată la orice timp dorit. Pentru testare, am folosit 10 secunde. Intrebarile si raspunsurile sunt stocate folosind un fisier xml pentru usurinta si eficienta si pentru a respecta cerinta.De asemenea pentru a permite orice numar de clienti am folosit o alocare dinamica pentu vector si am folosit optiunea SOMAXCONN de la listen care permite un numar maxim de clienti rezolvand astfel aceasta problema.

3 Structura aplicației

Aplicația funcționează după cum urmează: Se pornește serverul și se specifică numărul de jucători care vor participa. Acest număr poate fi ales liber. După

ce este specificat, serverul așteaptă ca toți jucătorii să se înregistreze. Fiecare jucător începe cu un scor de 0. Se creează un thread separat pentru fiecare jucător, iar comunicarea este gestionată independent. După ce toți jucătorii s-au conectat și s-au sincronizat, jocul începe cu comanda start, care eliberează firele jucătorilor care stăteau anterior în bucla `while (!game_started)`. Intrebarile sunt stocate intr-un fisier xml si la fiecare pas se extrage intrebarea noua si raspunsul ei (si le memoram in 2 variabile intrebare,raspuns) care mai apoi este trimisa catre fiecare jucator. Jucătorii au un timp limitat pentru a răspunde (stocat în variabila `timpr`). Răspunsurile sunt gestionate astfel:

- Dacă răspunsul este corect, scorul crește.
- Dacă nu există răspuns sau răspunsul este greșit, scorul rămâne neschimbat.
- Dacă un jucător se deconectează, acesta este eliminat din joc (acest lucru este realizat prin verificarea a ce s-a primit după `recv` și marcarea jucătorului ca inactiv dacă s-a deconectat).

La finalul jocului, după ce toate întrebările au fost parcurse, se anunță scorul fiecărui jucător, iar câștigătorul este anunțat tuturor jucătorilor fiind gestionate si posibilitati de scoruri egale.

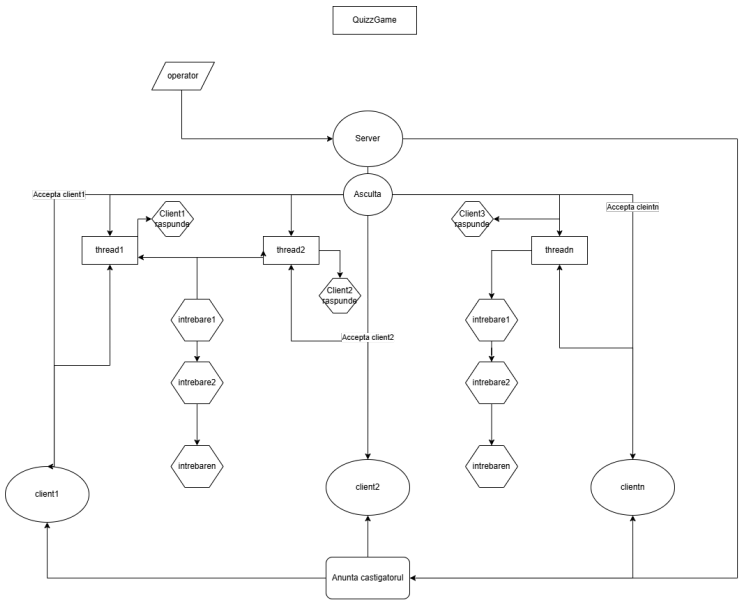


Fig. 1. Diagrama

4 Aspecte de implementare

Este important de precizat că rolul clientului în această aplicație este minimal, având doar sarcina de a răspunde la întrebări în timpul acordat, majoritatea logicii fiind realizată pe server.

```
1  int* client_index = malloc(sizeof(int));
2  *client_index = client_count;
3  pthread_create(&threads[client_count], NULL, handle_client, client_index);
4
```

Fig. 2. Codul pentru crearea unui thread pentru fiecare client.

Un aspect semnificativ al implementării este metoda prin care am transmis indexul clientului către thread-ul care îl gestionează, economisind astfel resurse prin evitarea căutărilor suplimentare pentru a obține indexul.

```
1  pthread_mutex_lock(&lock);
2  clients[client_count].socket = client_socket;
3  clients[client_count].scor = 0;
4  clients[client_count].activ = 1;
5  client_count++;
6  pthread_mutex_unlock(&lock);
```

Fig. 3. Codul pentru sincronizarea accesului la structura de date a clienților.

Un alt detaliu important al implementării este utilizarea mutexurilor pentru a evita condițiile de tip ”race condition” și pentru a preveni accesul simultan la resursele partajate.În exemplul nostru, două fire pot încerca simultan să modifice `client_count`, ceea ce ar putea duce la coruperea datelor sau la inconsistențe. Folosind mutexuri, ne asigurăm că actualizările sunt realizate secvențial, menținând consistența datelor și prevenind erorile.

```c
fd_set readfds;
struct timeval timpr;
FD_ZERO(&readfds);
FD_SET(client_socket, &readfds);
timpr.tv_sec = timpr;
ttimpr.tv_usec = 0;

int activity = select(client_socket + 1, &readfds, NULL, NULL, &timpr);
if (activity > 0 && FD_ISSET(client_socket, &readfds))
{
    //raspuns
} else
{
    // nu a raspuns la timp
}

```

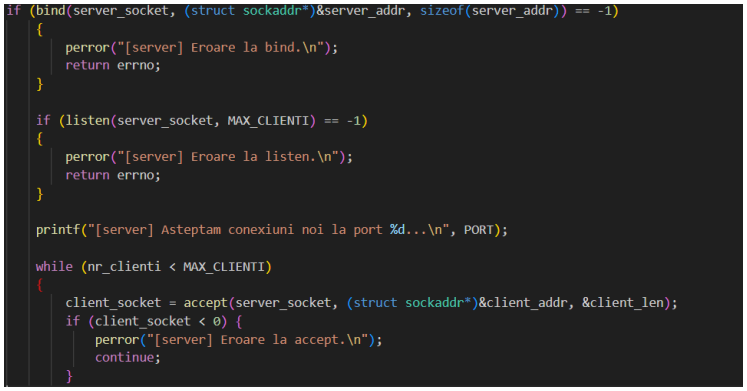
Fig. 4. Codul pentru implementarea unui timer de răspuns pentru fiecare întrebare.

Funcționalitatea timpului de răspuns pentru fiecare întrebare este gestionată printr-o instrucțiune `select`. Serverul așteaptă răspunsul de la client în cadrul unui timp specificat. Dacă clientul nu răspunde la timp, se trece automat la următoarea întrebare, astfel încât quizz-ul să nu rămână blocat.

```c
if (bytes <= 0)
{
    printf("[server] Client %d deconectat.\n", client_index);
    pthread_mutex_lock(&lock);
    clients[client_index].activ = 0;
    pthread_mutex_unlock(&lock);
    break;

}
```

Fig. 5. Codul pentru gestionarea deconectării unui client.

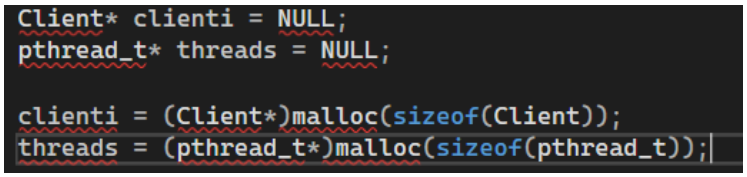
Atunci când un client se deconectează, jucătorul este marcat ca inactiv (cu `active = 0`), iar jocul continuă fără probleme, iar serverul nu se oprește.



```
if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1)
    {
        perror("[server] Eroare la bind.\n");
        return errno;
    }

    if (listen(server_socket, MAX_CLIENTI) == -1)
    {
        perror("[server] Eroare la listen.\n");
        return errno;
    }

    printf("[server] Asteptam conexiuni noi la port %d...\n", PORT);

    while (nr_clienti < MAX_CLIENTI)
    {
        client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &client_len);
        if (client_socket < 0) {
            perror("[server] Eroare la accept.\n");
            continue;
        }
```

Fig. 6. Codul pentru protocol TCP.

Protocolul utilizat este TCP întrucât folosim listen,accept ce nu se regăsesc într-un protocol UDP.În codul de mai sus este exemplificat acest lucru.



```
Client* clienti = NULL;
pthread_t* threads = NULL;

clienti = (Client*)malloc(sizeof(Client));
threads = (pthread_t*)malloc(sizeof(pthread_t));
```

Fig. 7. Codul pentru alocare dinamica.

O alta problema pe care o intampinam era cea a numarului de clienti care se pot conecta intrucat se doreste ca acesta sa fie nelimitat.Am hotarat astfel sa rezolv aceasta problema printr-o alocarea dinamica a vectorului.Astfel numarul de clienti poate fi oricare.

```
<quizz>
<question id = "1">
<text>Cat este radical din 225 ?</text>
< answer>15 < answer >
</question>
```

Fig. 8. Codul pentru xml.

```
void IntrebaresiRaspuns(xmlNode* node, int intrebarei, char* intrebare, char* raspuns) {
    int count = 0;
    for (xmlNode* cur_node = node; cur_node; cur_node = cur_node->next) {
        if (cur_node->type == XML_ELEMENT_NODE && strcmp((const char*)cur_node->name, "question") == 0) {
            if (count == intrebarei) {
                for (xmlNode* child = cur_node->children; child; child = child->next) {
                    if (child->type == XML_ELEMENT_NODE) {
                        if (strcmp((const char*)child->name, "text") == 0) {
                            strcpy(intrebare, (const char*)xmlNodeGetContent(child));
                        }
                        else if (strcmp((const char*)child->name, "answer") == 0) {
                            strcpy(raspuns, (const char*)xmlNodeGetContent(child));
                        }
                    }
                }
                return;
            }
            count++;
        }
        IntrebaresiRaspuns(cur_node->children, intrebarei, intrebare, raspuns);
    }
}
```

Fig. 9. Codul pentru xml.

Folosirea unui fisier xml este o alta imbunatatire adusa proiectului.Intreabrile si raspunsurile sunt stocate intr-un fisier iar cautarea se face folosind o functie.Cand se gaseste intrebarea curenta obtinem informatia (intrebarea si raspunsul) si le stocam apoi in 2 variabile.Astfel putem avea oricate intrebari dorim si le putem accesa cu usurinta.

Un scenariu real de utilizare pentru această aplicație ar putea fi simularea unui test școlar. Profesorul poate pregăti un test cu mai multe întrebări, al cărui răspuns îl știe doar el, împreună cu timpul alocat fiecărei întrebări. Astfel, poate evalua eficient și fără probleme elevii.

5 Concluzii

Acest proiect poate fi îmbunătățit constant, iar acesta ar outea fi continuat cu urmatoarele comenzi: o comandă de pauză pentru a opri temporar jocul, o comandă de oprire a jocului (stop), sau o comandă de kick pentru a elimina un jucător din joc. O altă idee ar fi implementarea unei funcționalități de tip ”hint” prin care jucătorii pot primi indicii pentru întrebările dificile. Proiectul mi s-a parut unul interesant si am invatat multe lucruri noi mai ales cum se lucreaza cu fisiere xml dar si cu thread-uri.

6 Referințe bibliografice

1. https://www.geeksforgeeks.org/thread-functions-in-c-c/
2. https://stackoverflow.com/questions/3838112/how-c-select-function-works-in-unix-oss
3. https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
4. https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/
5. https://edu.info.uaic.ro/computer-networks/cursullaboratorul.php