Proiect RC. Mersul trenurilor

Mazilu Cosmin-Alexandru

[1] Universitatea Alexandru Ioan Cuza, Iași
[2] Facultatea de informatică
[3] https://www.info.uaic.ro/

Abstract. În acest raport tehnic este prezentat proiectul "Mersul trenurilor", modul în care a fost proiectat, tehnologiile folosite, implementarea lor și utilitatea proiectului în mare.

Keywords: Server · Computer Networks · Threads · Sockets - TCP · Multiplexing · Threads

1 Introducere

Scopul principal al proiectului este de a creea un sistem care ajută la gestionarea trenurilor, orarul în care circulă, rutele pe care le parcurg și stațiile/gările pe care le parcurg.

2 Tehnologii aplicate

Proiectul se bazează pe o arhitectură de tip client - server. Protocolul de comunicație utlizat este TCP (Transmion Control Protocol) [3] pentru evitarea pierderii de date importante despre statusul unui tren, poziția acestuia, existența unei posibile întârzieri etc. Serverul este unul concurent pentru a putea primi cereri de la mai mulți clienți simultan, concurența acestuia find realizată prin multiplexare [5] și ajutorul threadurilor [1]. Threadurile (fire de execuție) pot fi văzute asemenea unor programe aflate în execuție, dar fară spațiu de adresă proprie. Prin multiplexare se face posibilă monitorizarea mai multor descriptori ai clienților. Un descriptor este selectat de server atunci când clientul este pregătit de a transmite informații. Un "thread pool" [4] conține un număr finit de threaduri și se ocupă cu descriptorii selectați prin multiplexare. Avantajul acestui thread pool este acela că serverul nu trebuie să creeze câte un thread pentru fiecare client în parte, ceea ce ar duce la un consum crescut de resurse pentru un număr ridicat de clienți, deoarece threadurile din thread pool sunt capabile să se mute pe alt descriptor după ce îl termină de servit pe cel anterior. Datele despre trenuri și rutele acestora, vor fi salvate în fișiere de tip XML [2], pentru o mai bună organizare a informațiilor.

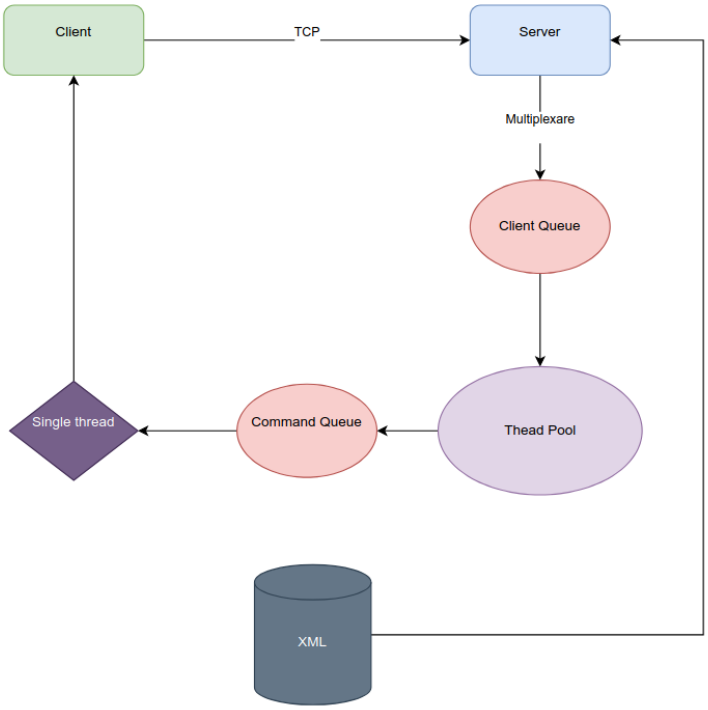

Fig. 1. Arhitectura client-server.

3 Structura aplicației

Conexiunea dintre client și server se face prin protocolul de comunicare TCP. Clienții sunt selectați prin multiplexare doar atunci cand aceștia sunt pregătiți să transfere date. Descriptorii clienților selectați sunt puși intr-o coadă pentru a fi procesați de threadurile din thread pool. Thread poolul este format din n threaduri (testat cu $n = 30$) care au calitatea de a se putea ocupa de un alt client după ce termin cel anterior.Pentru a se asigura faptul că două threaduri nu aleg același client din coadă, este folosit un mutex care blochează toate celelalte threaduri din a alege un client.De asemeanea, pentru a optimiza serverul și pentru ca threadurile să încerce să preia un client din coadă, deși coada este goală, este folosit un "condition variable" care blochează threadurile din a alege clienti din coadă până când starea cozii se schimbă.
Odată ce un thread a ales un client, acesta preia comanda de la client și o procesează. Procesarea constă în încapsularea într-un obiect al comenzii, obiect care v-a conține descriptorul clientului, un "flag" cu ceea ce își dorește clientul (de exemplu: flag = 1, dacă clientul dorește să afle informații despre un anumit tren), durata în minute a întârzierii unui tren, id-ul trenului etc.Acest obiect v-a fi pus într-o altă coadă cu comenzi.

Pentru a se asigura că comenzile sunt procesate în orindem un singur thread, diferit de cele din thread pool, se va ocupa de cererile din coada cu comenzi. Acest thread, preia obiectul din coadă și caută/modifică în fișierul XML ceea ce este cerut de comandă și trimite răspunsul către descriptorul din obiect.

4 Aspecte de implementare

Listing 1.1. Selectarea clienților de către threaduri

```
struct cl{
    int client_fd;
    char command_buff[BUFFER_SIZE];
};

void* client_handler(void* arg) {
    while(true) {
        pthread_mutex_lock(&client_mutex);

        while(clients_queue.empty()) {
            pthread_cond_wait(&client_condition,
                              &client_mutex);
        }
        cl client = clients_queue.front();
        clients_queue.pop();
        pthread_mutex_unlock(&client_mutex);

        handle_client(client);
    }
    return nullptr;
}
```

Cu ajutorul mutexurilor, se asigură că threadurile nu selectează același client, evitând astfel problema de race condition.De asemenea, prin condition variable, threadurile nu caută încontinuu clienți în ce coada este goală, reducând major resursele hardware consumate de către server.

Listing 1.2. Creearea serverului și a socketului din server

```
    if((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("[Server]Error at socket()\n");
        exit(EXIT_FAILURE);
    }

    if(setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR |
                SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("[Server]Error at setsockopt()\n");
        exit(EXIT_FAILURE);
    }

```

```
12      address.sin_family = AF_INET;
13      address.sin_addr.s_addr = INADDR_ANY;
14      address.sin_port = htons(PORT);
15
16      if(bind(server_fd, (struct sockaddr*)&address,
17              sizeof(address)) < 0) {
18          perror("[Server]Error at bind()\n");
19          exit(EXIT_FAILURE);
20      }
21
22      if(listen(server_fd, MAX_CLIENTS) < 0) {
23          perror("[Server]Error at listen()");
24          exit(EXIT_FAILURE);
25      }
```

Listing 1.3. Creearea clientului și conectarea la server

```
1   if((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
2           perror("[Client]Error at socket().\n");
3           return errno;
4       }
5
6       server.sin_family = AF_INET;
7       server.sin_addr.s_addr = inet_addr(argv[1]);
8       server.sin_port = htons(port);
9
10      //connecting to server
11      if(connect(sd, (struct sockaddr*) &server,
12              sizeof(struct sockaddr)) == -1){
13          perror("[Client]Error at connect().\n");
14          return errno;
15      }
```

Listing 1.4. Command Thread

```
1   void* command_handler(void* arg) {
2       char file[] = "trains.xml";
3       XMLClass xml_file(file);
4       while(true) {
5           pthread_mutex_lock(&task_mutex);
6           while(tasks_queue.empty()) {
7               pthread_cond_wait(&task_condition, &task_mutex);
8           }
9
10          Command task = tasks_queue.front();
11          tasks_queue.pop();
12          pthread_mutex_unlock(&task_mutex);
13
14          // Process the command
15          if(task.GetFlag() == 1){
```

```
16              char response[1000];
17              strcpy(response, xml_file.GetTodayTrains());
18              send_message(task.GetClient(), response);
19          } else if(task.GetFlag() == 2){
20              char response[1000];
21              strcpy(response,
22                  xml_file.GetThisHourDepartures(
23                      task.GetCurrentHour()));
24              send_message(task.GetClient(), response);
25          } else if(task.GetFlag() == 3){
26              char response[1000];
27              strcpy(response,
28                  xml_file.GetThisHourArrivals(
29                      task.GetCurrentHour()));
30              send_message(task.GetClient(), response);
31          } else if(task.GetFlag() == 4){
32              char response[1000];
33              strcpy(response, xml_file.showInfo(
34                  task.GetTrainId()));
35              send_message(task.GetClient(), response);
36          } else if(task.GetFlag() == 5){
37              char response[1000];
38              strcpy(response, xml_file.addDelay(
39                  task.GetTrainId(),
40                  task.GetDelay(), task.GetStation()));
41              send_message(task.GetClient(), response);
42          } else if(task.GetFlag() == 6) {
43              char response[1000];
44              strcpy(response,
45                  xml_file.addEarly(task.GetTrainId(),
46                  task.GetDelay(), task.GetStation()));
47              send_message(task.GetClient(), response);
48          } else {
49              char response[1000];
50              strcpy(response, "Invalid␣command.\n");
51              send_message(task.GetClient(), response);
52          }
53      }
54      return nullptr;
55  }
```

5 Scenariu real de utilizare

Aplicația ar putea fi utilizată de personalul feroviar pentru o organizare mai bună. De exemplu, personalul dintr-o gară ar pute cere informații despre trenurile care ajung în gară pentru a știi cum să le aranjeze pe linii, în functie de numărul trenurilor, întârzieri și de capacitatea gării. Acest lucru ar putea evita accidentele, de exemplu de a pune 2 trenuri pe aceeași linie dar pe sensuri opuse.

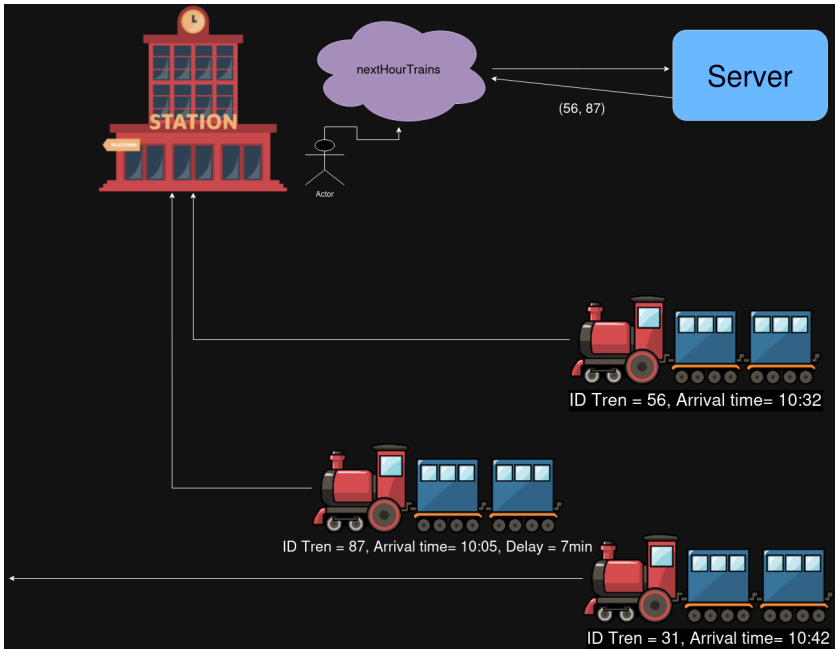

Fig. 2. Use case.

6 Concluzii

O potențială îmbunătățire ar fi crearea unei interfațe grafice pentru o utilizare mai ușoară. De asemenea, o integrare cu toate gările și stațiile pentru o eficientizare a trenurilor. O altă posibilă îmbunătățire ar fi ca și pasagerii să poată folosi aplicația pentru a ști când ajunge un tren și când să își cumpere bilete.

References

1. Alboaie, L.: Programare in retea (III) (2024), https://edu.info.uaic.ro/computer-networks/files/7rc$_{Programarea In Retea III_{RO}.pdf}$
2. AWS: What is XML? (2024), https://aws.amazon.com/what-is/xml/
3. GeeksforGeeks: What is TCP (Transmission Control Protocol)? (2024), https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/
4. Sorber, J.: How to write a multithreaded webserver using condition variables (Part 3) (2019), https://www.youtube.com/watch?v=P6Z5K8zmEmct=382s
5. TechTarget: Multiplexing (2024), https://www.techtarget.com/searchnetworking/definition/multiplexing