Raport tehnic - Proiect “Mersul trenurilor”

1.Introducere

Scopul acestui proiect este de a avea un server care are acces la un fișier de tip XML, ce contine date despre mersul trenurilor, și care actualizează datele(intarzieri si estimare sosire) la cererea clienților. Toata logica va fi realizata in server, clientul doar cere informații despre plecari/sosiri și trimite informații la server despre posibile intarzieri si estimari sosiri, la baza fiind un protocol de tip TCP, concurent (se pot conecta mai mulți clienți la server).

2.Tehnologii utilizate

2.1 TCP

TCP (Transmission Control Protocol) este un protocol fundamental al suitei de protocoale internet (Internet Protocol Suite) utilizat pentru transmiterea datelor în rețelele de calculatoare. Este un protocol orientat pe conexiune, ceea ce înseamnă că stabilește o conexiune fiabilă între două dispozitive înainte de a începe transferul de date. TCP asigură transmiterea corectă a datelor. Înainte de a începe transferul de date, TCP inițiază un proces de "strângere de mână în trei pași" (three-way handshake) pentru a stabili o conexiune între expeditor și receptor.

2.2 Parserul XML folosind Libxml2

Libxml2 este o bibliotecă scrisă în C pentru procesarea fișierelor XML. Aceasta oferă funcționalități puternice pentru analizarea, manipularea și traversarea documentelor XML. Este bazată pe modelul DOM (Document Object Model), ceea ce permite accesarea elementelor XML ca noduri într-un arbore ierarhic. Biblioteca este extrem de eficientă și suportă standardele XML, inclusiv XPath pentru interogarea datelor complexe.

2.3 Sockets

Un socket este un punct final de comunicare utilizat pentru a transmite date între două noduri (procese) dintr-o rețea. Este o interfață software care permite aplicațiilor să trimită și să primească date prin rețea utilizând protocoale precum TCP/IP sau UDP.

2.4 Threads

Un thread este cea mai mică unitate de execuție dintr-un proces, reprezentând o cale independentă de execuție a codului. Fiecare proces poate conține unul sau mai multe threads care rulează în paralel și împart aceeași memorie.

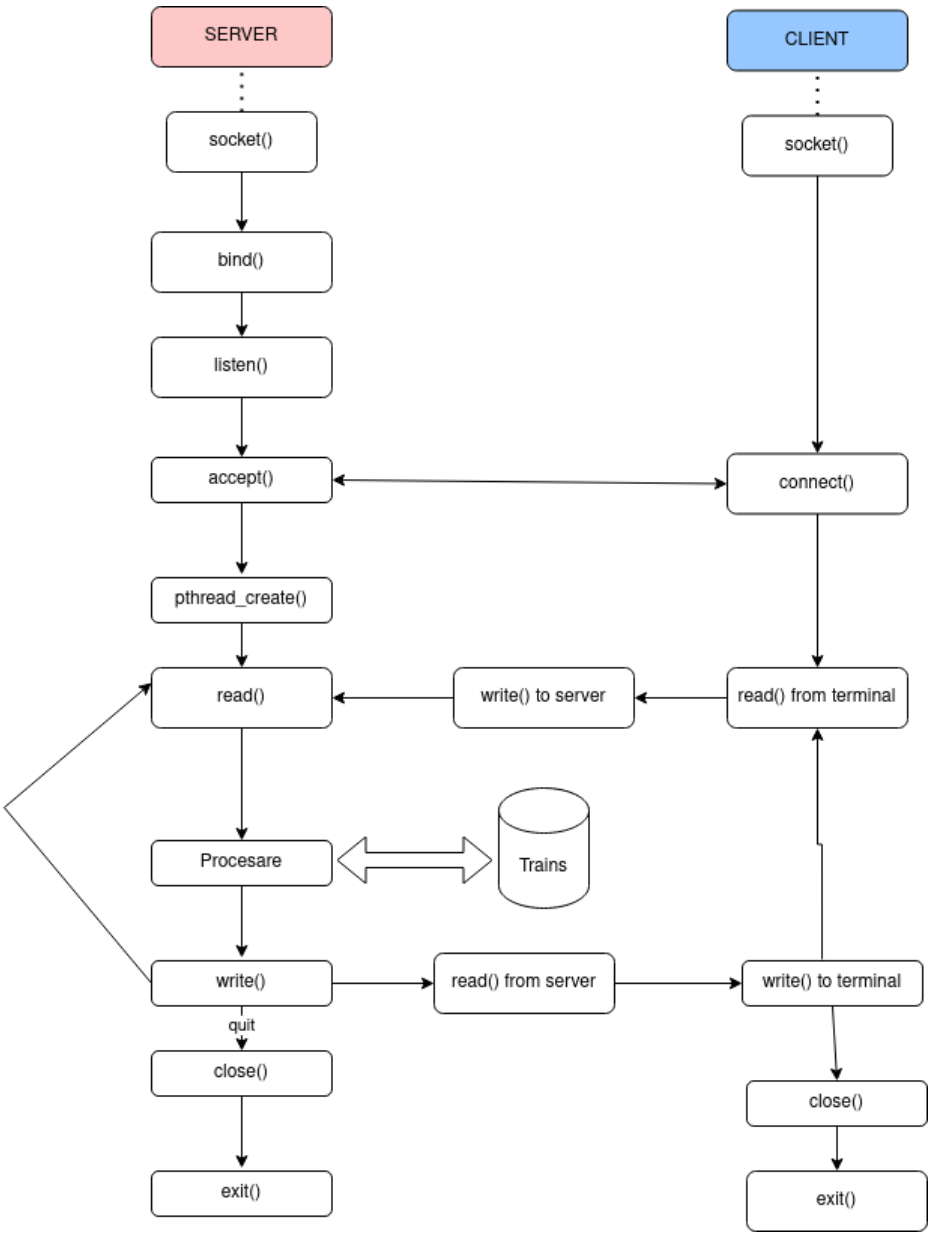
3.Structura aplicației

Aplicația este formată din două componente principale:

1. **Serverul** – gestionează cererile clienților, actualizează datele trenurilor și răspunde în timp real.
2. **Clienții** – trimit cereri pentru a primi informații despre trenuri sau pentru a raporta întârzieri/ estimări sosire.

Modelare conceptuală

- **Thread Management**: Fiecare client este asociat unui thread care primește cererile și le transformă în comenzi.
- **Command Queue**: O coadă centralizată stochează comenzile care sunt procesate secvențial de cate un thread .



4.Aspecte de Implementare

4.1 Server

4.1.1 Implementare coada comenzi

```
typedef enum {
        GET_SOSIRI,
        GET_PLECARI,
        GET_MERS,
        UPDATE_DELAY,
        UPDATE_EARLY
} CommandType;
```

Exista 3 tipuri de cerere de informații din partea clientului: plecari din următoarea ora, sosiri din următoarea oră, mersul trenurilor (plecari si sosiri) din ziua respectivă. De asemenea, exista 2 tipuri de cerere de actualizare a datelor din partea clientului : actualizare intarziere, estimare sosire

```
typedef struct {
        CommandType type;
        int client_id;
        char args[256];
} Command;
```

Structura elementului Command

```
typedef struct {
        Command queue[QUEUE_SIZE]; //
        int front, rear;
        pthread_mutex_t mutex;
        pthread_cond_t cond;
} CommandQueue;
```

`Command queue[QUEUE_SIZE]-` Reprezintă coada propriu-zisă, un tablou de elemente de tip `Command`.

`front-` Indică poziția primului element din coadă (capul cozii).

`rear -` Indică poziția unde următorul element va fi adăugat (coada efectivă).

`pthread_mutex_t mutex -` Un mutex utilizat pentru a proteja accesul concurent la coadă.Previne condițiile de competiție atunci când mai multe thread-uri accesează sau modifică simultan coada.

`pthread_cond_t cond-` O variabilă de condiție care permite sincronizarea între thread-uri.Este folosită pentru a aștepta până când un element devine disponibil în coadă sau până când coada este pregătită pentru o operație.

```
void initQueue(CommandQueue *q) {
        q->front = q->rear = 0;
        pthread_mutex_init(&q->mutex, NULL);
        pthread_cond_init(&q->cond, NULL);
}
```

`pthread_mutex_init(&q->mutex, NULL)-`Inițializează mutex-ul asociat cozii.

`pthread_cond_init(&q->cond, NULL)-` Inițializează variabila de condiție pentru coadă.

```
void enqueue(CommandQueue *q, Command cmd) {
        pthread_mutex_lock(&q->mutex);
        q->queue[q->rear] = cmd;
        q->rear = (q->rear + 1) % QUEUE_SIZE;
        pthread_cond_signal(&q->cond);
        pthread_mutex_unlock(&q->mutex);
}
```

`pthread_mutex_lock(&q->mutex)-`Blochează mutex-ul asociat cozii pentru a preveni accesul concurent.
`q->queue[q->rear] = cmd-` Adaugă comanda (`cmd`) la poziția curentă indicată de `rear`.
`q->rear = (q->rear + 1) % QUEUE_SIZE-` Actualizează indicele `rear` pentru a indica următoarea poziție liberă din coadă. Folosește operatorul modulo (`%`) pentru a implementa coada circulară, prevenind depășirea limitelor tabloului.
`pthread_cond_signal(&q->cond)-` Semnalizează altor thread-uri care așteaptă pe condiția `q->cond` că un element a fost adăugat în coadă și este disponibil pentru procesare.
`pthread_mutex_unlock(&q->mutex)-` Eliberează mutex-ul, permițând altor thread-uri să acceseze coada.

```
Command dequeue(CommandQueue *q) {
        pthread_mutex_lock(&q->mutex);
        while (q->front == q->rear) { // Coadă goală
        pthread_cond_wait(&q->cond, &q->mutex);
        }
        Command cmd = q->queue[q->front];
        q->front = (q->front + 1) % QUEUE_SIZE;
        pthread_mutex_unlock(&q->mutex);
        return cmd;
}
```

`pthread_mutex_lock(&q->mutex)-`Blochează mutex-ul pentru a preveni accesul concurent la coadă.
`while (q->front == q->rear)-` Verifică dacă coada este goală.
`pthread_cond_wait(&q->cond, &q->mutex)-` Așteaptă pe variabila de condiție `q->cond` până când este semnalizată de alt thread că un element a fost adăugat în coadă. Mutex-ul este temporar eliberat în timpul așteptării, permițând altor thread-uri să opereze asupra cozii.
`Command cmd = q->queue[q->front]-` Preia elementul de la poziția curentă indicată de `front`.
`q->front = (q->front + 1) % QUEUE_SIZE-` Actualizează indicele `front` pentru a indica următorul element din coadă. Folosește operatorul modulo pentru a implementa coada circulară.
`pthread_mutex_unlock(&q->mutex)-` Eliberează mutex-ul, permițând altor thread-uri să acceseze coada.
`return cmd-` Returnează comanda preluată din coadă pentru procesare.

Comenzile sunt adăugate în coadă în funcția `clientHandler`, care este rulată într-un thread separat pentru fiecare client.Comenzile sunt plasate în coadă imediat ce serverul le primește de la client și le procesează pentru a extrage tipul comenzii și parametrii acesteia.

```
if (strcmp(command_name, "GET_SOSIRI") == 0) {
        char *time = strtok(NULL, " ");
        if (time) {
                cmd = (Command){GET_SOSIRI, client_id, ""};
                strcpy(cmd.args, time);
                enqueue(&commandQueue, cmd);
        }
```

Main

Pornirea thread-ului de procesare a comenzilor
pthread_t processor_thread; pthread_create(&processor_thread, NULL, commandProcessor, NULL);
Funcția `commandProcessor` rulează într-un buclă infinită, apelând `dequeue` pentru a scoate comenzile din coadă și procesându-le în funcție de tipul lor

Serverul este configurat să accepte conexiuni printr-un socket TCP. Este creat socket-ul serverului (socket(AF_INET, SOCK_STREAM, 0)).Adresa și portul serverului sunt configurate (bind).Serverul este setat să accepte conexiuni cu listen.

Serverul intră într-un buclă infinită în care acceptă o conexiune nouă folosind accept si creează un thread separat pentru fiecare client care se conectează.

pthread_t client_thread;
pthread_create(&client_thread, NULL, clientHandler, data);

pthread_detach(client_thread);
Se creează un thread separat care rulează funcția `clientHandler`.Funcția `clientHandler` primește comenzile clientului, le parcurge, le identifică și le pune în coada globală folosind enqueue.

4.1.2 XML

-din functia sendTrainSchedule:

xmlDoc *doc = xmlReadFile("train_schedule.xml", NULL, 0);
`xmlReadFile`: Această funcție din biblioteca libxml2 încarcă și parsează fișierul XML specificat și creează un obiect de tip xmlDoc.

xmlNode *root = xmlDocGetRootElement(doc);
`xmlDocGetRootElement`: Returnează nodul rădăcină al documentului XML (`<trains>`). Toate nodurile copil ale rădăcinii sunt accesibile în structura ierarhică a documentului XML.

```
for (cur_node = root->children; cur_node; cur_node = cur_node->next) {
        if (cur_node->type == XML_ELEMENT_NODE &&
        xmlStrcmp(cur_node->name, (const xmlChar *)"train") == 0) {
```

xmlChar *id = xmlGetProp(cur_node, (const xmlChar *)"id");
`xmlGetProp`: Obține valoarea atributului `id` din nodul `<train>` curent.

```
        xmlChar *departure = NULL;
        xmlChar *arrival = NULL;
        xmlChar *route = NULL;
```

Variabilele `departure`, `arrival` și `route` sunt inițializate ca fiind NULL pentru a stoca ulterior conținutul elementelor `<departure>`, `<arrival>` și `<ruta>` ale nodului curent.

```
        for (child = cur_node->children; child; child = child->next) {
                if (child->type == XML_ELEMENT_NODE) {
                if (xmlStrcmp(child->name, (const xmlChar *)"departure") == 0) {
                departure = xmlNodeGetContent(child);
                } else if (xmlStrcmp(child->name, (const xmlChar *)"arrival") == 0) {
                arrival = xmlNodeGetContent(child);
                } else if (xmlStrcmp(child->name, (const xmlChar *)"ruta") == 0) {
                route = xmlNodeGetContent(child);
                }
                }
        }
```

Iterarea prin copiii nodului `<train>`: Se accesează fiecare copil al nodului `<train>`.
Verificare tip nod: Se verifică dacă nodul copil este un element XML valid (`XML_ELEMENT_NODE`).Dacă nodul copil este `<departure>`, `<arrival>` sau `<ruta>`, conținutul acestuia este extras folosind `xmlNodeGetContent` și stocat în variabilele respective (`departure`, `arrival`, `route`

xmlFreeDoc(doc);
Eliberează memoria utilizată de obiectul `doc`, care reprezintă documentul XML încărcat.

-din funcția updateTrainTime:

```
for (cur_node = root->children; cur_node; cur_node = cur_node->next) {
        if (cur_node->type == XML_ELEMENT_NODE &&
                xmlStrcmp(cur_node->name, (const xmlChar *)"train") == 0) {
                xmlChar *id = xmlGetProp(cur_node, (const xmlChar *)"id");
```

Obține valoarea atributului `id` din nodul `<train>` curent.

if (id && strcmp((char *)id, train_id) == 0)
`strcmp((char *)id, train_id)` compară ID-ul obținut din XML cu cel specificat în cerere (`train_id`).Dacă se găsește trenul cu ID-ul potrivit, intrăm în blocul de cod pentru actualizare.

```
                { xmlNode *child = NULL;
                 for (child = cur_node->children; child; child = child->next) {
  if (child->type == XML_ELEMENT_NODE && strcmp((char *)child->name, type) == 0) {
                        char *current_time = (char *)xmlNodeGetContent(child);
```

Verifică dacă nodul curent (`child`) este un element XML valid (`XML_ELEMENT_NODE`) și dacă numele nodului este egal cu valoarea specificată de `type` (fie `departure`, fie `arrival`).

xmlNodeSetContent(child, (const xmlChar *)updated_time);
Actualizează conținutul nodului XML curent cu noua valoare a orei (`updated_time`), care a fost calculată în funcție de ajustarea specificată.

xmlSaveFormatFileEnc("train_schedule.xml", doc, "UTF-8", 1);
Salvează documentul XML modificat în fișierul `train_schedule.xml`, păstrând formatul original și utilizând codificarea UTF-8 (o codificare standard care suportă caractere Unicode) Indică faptul că fișierul trebuie salvat într-un format "frumos" (eng. "pretty-printed").Formatul "frumos" înseamnă că nodurile XML sunt aliniate corespunzător, cu indentare și spații adăugate, ceea ce face fișierul mai ușor de citit de către oameni.

4.2 Client

```
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Eroare la socket().\n");
        return errno;
}
```

inițializare socket

server.sin_family = AF_INET;

```
server.sin_addr.s_addr = inet_addr(argv[1]);
server.sin_port = htons(port);
```
Configurare adresă server

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

Conectare server

Clientul intră într-o buclă în care solicită o comandă de la utilizator, verifică formatul comenzii și o validează, apoi trimite comanda către server, iar la final primește răspunsul de la server și îl afișează.

```
if (write(sd, buffer, strlen(buffer)) <= 0) {
   perror("[client]Eroare la write() spre server.\n");
   free(buffer);
   return errno;
}
```
Comanda validată este trimisă la server

```
 while (!response_complete && (bytes_read = read(sd, response + total_read,
response_size - total_read - 1)) > 0) {
        total_read += bytes_read;
        if (total_read >= response_size - 1) {
           response_size *= 2;
           char *new_response = realloc(response, response_size);
           if (!new_response) {
              perror("[client]Eroare la realocarea memoriei.\n");
              free(response);
              return errno;
           }
           response = new_response;
        }
        if (response[total_read - 1] == '\n') {
           response_complete = 1;
        }
     }
     if (bytes_read < 0) {
        perror("[client]Eroare la read() de la server.\n");
        free(response);
        return errno;
     }
     response[total_read] = '\0';
     printf("[client]Raspunsul serverului: %s\n", response);
```

```
    free(response);
}
```

Primirea răspunsului de la server. Citire prin alocare dinamica de memorie. Când răspunsul include `\n` la final, citirea se oprește.

4.3 Scenarii reale de utilizare

- Un pasager dorește să afle informații despre trenurile care sosesc sau pleacă din gară din ziua respectivă. Clientul trimite comanda `GET_MERS.` Serverul caută în baza de date informațiile cerute și răspunde cu lista trenurilor și orele aferente.
- Un pasionat de trenuri dorește sa vadă ce trenuri ajung în următoarea oră, sperand ca unul din trenuri să ajungă în stația în care se afla el. Acesta introduce comanda `GET_SOSIRI` . Dacă nu introduce nicio ora (ex: 16:00) va fi folosită ora din sistem.
- Operatorul gării actualizează întârzierile unui tren cauzate de condițiile meteorologice. Clientul trimite comanda `UPDATE_DELAY 3 15 departure.` Unde 3 este id-ul trenului, 15 este numărul de minute întârziere, iar “departure” este una dintre cele 2 opțiuni de actualizare (departure/arrival)`.` Serverul actualizează informația în baza de date și răspunde cu o confirmare.
- Un utilizator închide aplicația. Clientul trimite comanda `EXIT` . Serverul confirmă.

5.Concluzii

O posibila îmbunătățire ar fi ca în loc să se creeze câte un thread nou pentru fiecare conexiune client, serverul utilizează un grup (pool) prealocat de thread-uri. Aceste thread-uri sunt reutilizate pentru a deservi noi conexiuni, ceea ce reduce costurile asociate creării și distrugerii thread-urilor.Pool-ul limitează numărul maxim de thread-uri, prevenind utilizarea excesivă a memoriei și a procesorului. Timpul de răspuns este redus, deoarece thread-urile sunt deja disponibile pentru a gestiona conexiunile.

6.Bibliografie

https://edu.info.uaic.ro/computer-networks/cursullaboratorul.php
Andrew S. Tanenbaum, Nick Feamster, David Wetherall, Computer Networks, 6th Edition, ISBN-13: 9780137523214, 2021
https://en.wikipedia.org/wiki/Network_socket
https://en.wikipedia.org/wiki/Thread_(computing)
https://app.diagrams.net/
https://stackoverflow.com/questions/399704/xml-parser-for-c
https://www.w3schools.com/xml/xml_whatis.asp