Mersul trenurilor

Raport tehnic realizat de Petrasuc Ana

Facultatea de informatica

1 Introducere

1.1 Viziunea principala

- Proiectul ales vizeaza dezvoltarea unui sistem **server-client** care ofera si actualizeaza informatii in timp real despre mersul trenurilor.
- **Serverul** va extrage datele din fisiere sursa, de tip XML, si va oferi clientilor informatii despre mersul trenurilor in ziua respectiva si, la cerere, statusul plecarilor sau a sosirilor in urmatoarea ora.
- **Clientul** va face sesizare referitoare la intarzierile trenurilor, trimitand spre server numarul trenului, intervalul orar in care intarzie si estimarea sosirii.

1.2 Obiectivele proiectului

- Gestionarea simultana a mai multor clienti prin intermediul firelor de executie**(threads)** si asigurarea unei comunicari eficiente intre server si clienti.
- Utilizarea **design pattern-ului Command** pentru gestionarea comenzilor primite de la clienti si implementarea unuei **cozi** care sa le stocheze, astfel incat acestea sa poata fi procesate intr-o maniera sincronizata si organizata.
- **Actualizarea in timp real** a informatiilor referitoare la intarzieri pe baza datelor primite de la clienti.

2 Tehnologii aplicate

2.1 TCP/IP concurent cu *threads* (un server concurent care creeaza un fir de executie pentru fiecare client conectat)

: ales datorita fiabilitatii sale si asigurarii livrarii datelor in ordine la destinatie, aspect important pentru un sistem feroviar in timp real, unde integritatea si secventa datelor sunt prioritare.

2.2 Sockets

: acestea permit comunicarea intre server si clienti intr-un mod flexibil si faciliteaza realizarea conexiunilor si transferul de date bidirectional intre entitatile implicate, esential pentru un sistem de transmitere a informatiilor referitoare la mersul trenurilor intre server si clienti.

2.3 XML Parsing

: datele despre mersul trenurilor sunt stocate in fisiere XML. Tehnologia de analiza XML este cruciala pentru extragerea si manipularea eficienta a datelor din aceste fisiere. Pentru aceasta operatiune poate fi folosita biblioteca ***libxml2***.

2.4 Threads-*fire de executie*

: acestea permit gestionarea simultana a mai multor clienti. Acest lucru este important pentru server, acesta trebuind sa proceseze si sa raspunda la comenzile multiple din partea clientilor in timp real, asigurand o experienta fluida.

2.5 Command Design Pattern

: cu ajutorul acestuia fiecare comanda specifica (mersul trenurilor, plecari, sosiri) poate fi usor gestionata, adaugata sau modificata, fara a afecta structura generala a sistemului.

3 Structura Aplicatiei

3.1 Concepte de modelare

Command Design Pattern Se utilizeaza acest pattern pentru a separa obiectele care solicita serviciile(comenzi) de obiectele care le furnizeaza(executoare). In context, fiecare cerere de la client este destinata unui anumit tip de comanda generala, facilitand astfel gestionarea si extinderea functionalitatilor.

Command Queue Permite stocarea si gestionarea comenzilor primite de la clienti intr-un mod ordonat si sincronizat. Aceasta este esentiala pentru a preveni aglomerarea si a gestiona eficient sarcinile primite.

Threads - *fire de executie* Utilizarea acestora permite gestionarea concurenta a mai multor clienti. Fiecarui client ii este deservit un fir de executie, asigurandu-se astfel o comunicare eficienta si sincronizata.

XML Parsing Aceasta permite analiza eficienta a documentelor XML si manipularea datelor pentru a le integra in aplicatie.

Sockets Comunicarea intre server si clienti se realizeaza prin intermediul socket-urilor. Acestea faciliteaza transmiterea bidirectionala a datelor intre entitati si sunt esentiale pentru functionarea unui server-client.

3.2 Diagrama Detaliata a Aplicatiei (vedeti fig.1 la finalul documentului)

4 Aspecte de implementare

4.1 Sectiuni de cod specifice:

- Cred ca principalele sectiuni de cod ale acestui proiect sunt **structurile** care definesc toate elementele necesare pentru transportul de date feroviare intre server si client (vezi **fig.2** la finalul documentului)
- De asemenea, **citirea din fisierele XML** este una inovativa, care foloseste biblioteca ***libxml2*** (vezi **fig.3** la finalul documentului)
- La fel, plasarea comenzilor intr-o coada (**Commands queue**) necesita mai multe functii care ajuta atat la adaugarea elementelor in coada, cat si la scoatere lor in ordinea intrarii lor (vezi **fig.4** la finalul documentului)

4.2 Documentatie pentru sectiunile de cod:

Structurile necesare pentru transportul de date feroviare Aceasta sectiune de cod nu a necesitat o documentatie avansata, ci doar gandirea propriu-zisa a structurii proiectului. Am realizat o structura specifica unei gari(**tip TrainStations**) care retine numele garii, numarul de trenuri care trec prin acea gara si baza de date care retine orarul trenurilor, citit din fisiere XML. Cea din urma este de **tip TrainsDataBase**, o alta structura care e specifica unui tren *(numarul trenului, orele la care ajunge in gara, orele la care pleaca din gara, intarzierea pe care o are, in anumite situatii)*

Citirea datelor din fisiere XML: Aceasta sectiune foloseste biblioteca libxml2, aceasta are la baza o structura care ajuta la citirea datelor dintr-un fisier XML pornind de la radacina (**root**). Cu ajutorul acesteia se recunosc simbolurile specifice pentru XML , cum ar fi $<, />$.

Plasarea comenzilor intr-o coada: CommandsQueue este o structura care va retine toate comenzile primite de la fiecare client in parte. Aceasta in vectorul *items* va retine comenzile, in variabila *front* se va afla pozitia primului element adaugat in coada, iar variabila *rear* va memora pozitia ultimului element adaugat in coada. Pentru adaugarea elementelor din coada si a scoaterii lor am realizat cele doua functii **enqueue** si **dequeue**.

4.3 Protocolul la nivelul aplicatiei

Pentru realizarea conexiunii dintre server si client am apelat la modelul TCP/IP cu thread-ri.Am folosit un socket pentru conexiune, am citit in server datele din fisierele XML si le-am memorat intr-o structura specifica, iar pentru fiecare client in parte am creat un fir de executie care functioneaza dupa urmatoarele reguli:

- datele ce trebuie folosite in thread se transmit printr-o structura specifica lui, data ca argument in functia acestuia.
- se asteapta trimiterea locatiei de catre client
- se cauta mersul trenurilor pentru locatia clientului si se trimit la acesta
- dupa ce citeste datele primite, clientul memoreaza intr-o coada toate cererile lui si le transmite spre server
- server-ul proceseaza comenzile, luand la rand comenzile din coada si transmite informatiile cerute(sosiri, plecari, intarzieri)

4.4 Descrieti scenarii reale de utilizare

Avand in vedere ca proiectul vizeaza transmiterea datelor feroviare, aplicarea acestuia in realitate nu e dificila, existand deja astfel de aplicatii. Oamenii s-ar conecta la aplicatie, trimitand locatia lor(gara in care se afla), iar aplicatia le-ar trimite mersul trenurilor din ziua respectiva. Acestia ar putea trimite si cereri in legatura cu plecari/ soiri in urmatoarea ora si ar avea posibilitatea sa trimita aplicatiei informatii referitoare la intarzierile trenurilor, pe care sa le poata vedea si alti oameni conectati la aplicatie.

5 Concluzii

Consider ca proiectul este unul complex, insa ar putea avea si anumite imbunatatiri cum ar fi: **preluarea fisierelor XML din surse oficiale** si nu prin crearea lor, astfel datele ar fi exacte si aplicatia ar putea fi folosita in realitate. De asemenea, s-ar putea adauga o **interfata grafica**, atragatoare(*care ar incuraja oamenii sa aleaga aplicatia aceasta*), o **sectiune noua**, in care un client poate sa-si cumpere sau sa-si anuleze un bilet pentru tren si un set de reguli de conectare: **login, logout** etc.

6 Bibliografie

1. *UNIX Network Programming: The sockets networking API*, W. Richard Stevens, Bill Fenner, Andrew M. Rudoff
2. *The illustrated Network: How TCP/IP Works in Modern Network*, Waalter Goralski
3. *Computer Networks*, A. Tanebaum
4. *IBM-TCP/IP Tutorial and Technical Overview*, Lydia Parziale, David T. Britt

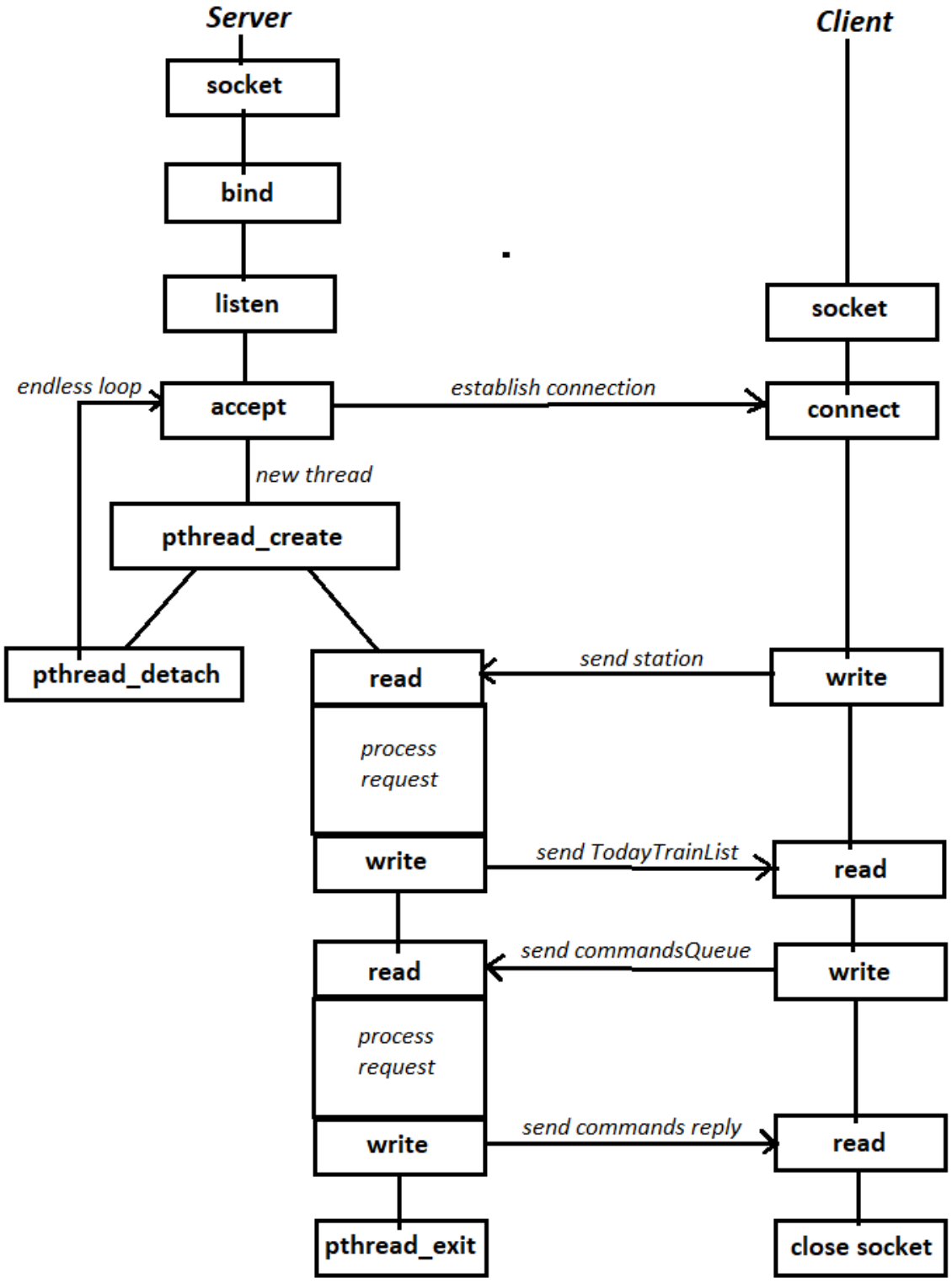

Fig. 1. Diagrama Aplicatiei

```c
struct Trains_DataBase{
    char trainNumber[30];
    char arrivalTime[30];
    char departureTime[30];
    char delay[30];
    char ArrivalUpdate[30];
    char DepartureUpdate[30];
};

struct Train_Stations {
    char stationName[30];
    struct Trains_DataBase DataBase[MAX_TRAINS];
    int trainsNumber;
};
```

Fig. 2. structuri

```c
struct Train_Stations readXMLFile(const char* xmlFile){
    xmlDocPtr doc;
    xmlNodePtr root;

    doc=xmlReadFile(xmlFile,NULL,0);

    if(doc==NULL){
        perror("[ServerError] Eroare la citirea fisierului XML.\n");
    }

    root=xmlDocGetRootElement(doc);

    if(xmlStrcmp(root->name, (const xmlChar*)"trains")!=0){
        perror("[ServerError] Fisierul XML nu contine date referitoare la trenuri.\n");
        xmlFreeDoc(doc);
    }

    struct Train_Stations station=dataRegistration(root);

    xmlFreeDoc(doc);
    xmlCleanupParser();

    return station;
}
```

Fig. 3. citireXML

```
typedef struct CommandsQueue{
    int items[30];
    int front;
    int rear;
};
```

```
void initializeQueue(struct CommandsQueue *queue){
    queue->front = -1;
    queue->rear = -1;
}

int *dequeue(struct CommandsQueue *queue){
    int command;
    command=queue->items[queue->front];

    if (queue->front == queue->rear) {
        initializeQueue(queue);
    } else {
        queue->front = (queue->front + 1) % 30;
    }
    return command;
}
```

```
oid enqueue(struct CommandsQueue *queue, int command){
if(queue->front == -1 && queue->rear == -1){
    queue->front = queue->rear = 0;
}
else{
    queue->rear = (queue->rear + 1) % 30;
}

queue->items[queue->rear]=command;
```

Fig. 4. CommandsQueue