Monitorizarea traficului

Bogdan Zavadovschi

Facultatea de Informatică,
Universitatea "Alexandru Ioan Cuza" Iași,
bogdan.zavadovschi@info.uaic.ro

Abstract

Monitorizarea traficului este un proiect care constă în implementarea unei aplicații prin care un participant la trafic poate transmite si primi informații despre diverse evenimente rutiere, trimițând în mod automat viteza și locația la un interval regulat de timp. Acesta poate sa primească și diverse informații despre vreme, prețul combustibilului și ultimele evenimente sportive. Aplicația este alcătuită din două componente: un client și un server, care folosesc tehnici de concurență, din motive diferite: serverul trebuie să accepte mai multe conexiuni simultan, iar clientul poate transmite și primi date în același timp.

1 Introducere

Aplicația de monitorizare a traficului constă în două componete cheie: un client și un server, ambele implementate concurent. Clientul are capacitatea de a procesa mesajele venite de la server și de a transmite informații in mod simultan, fără a fi nevoie să se aștepte terminarea unei cereri. Totodată acesta **simulează** doi sensori: unul de viteză și unul de locație. Aceștia vor fi interogați la un interval regulat de timp pe care fiecare client îl va putea seta, în funcție de preferință, iar informațiile vor fi trimise către server. Pe baza datelor primite, serverul va returna incidentele din trafic (dacă există in regiunea sa), limita de viteză pe secțiunea de drum și va modifica ultima locație a utilizatorului în baza de date.

Serverul are posibilitatea de a suporta mai mulți clienți simultan, datorită implementării concurente. La un interval regulat de timp, acesta va trimite informații către toți clienții (care au selectată opțiunea) despre vreme, evenimente sportive și ultimele prețuri la combustibil. În cazul în care este raportat un incident, serverul va notifica toți utilizatorilor aflați în proximitate producerii evenimentului.

Având imaginea de ansamblu, detaliile privind implementarea și arhitectura aplicației vor fi prezentate în urmatoarele secțiuni, iar la final voi propune și anumite imbunătățiri pe partea de securitate și performanță a aplicației.

2 Tehnologii utilizate

2.1 Protocolul de comunicare

Comunicarea dintre client/clienți și server este realizată prin **Transmission Control Protocol (TCP)**. Decizia de utilizare a acestui protocol provine din faptul că toate pachetele de informație trebuie să ajungă și să fie în aceeași ordine, fiind un aspect important întrucât pe baza datelor primite de server este implementată logica de transmitere a evenimentelor rutiere, iar mesajele de tip text primite de client trebuie să fie inteligibile.

2.2 Stocare datelor

Pentru a putea ține o anumită evidență a datelor, atât a utilizatorilor, a evenimentelor media, incidente rutiere, limite de viteză șamd. aplicația utilizează o bază de date, **sqlite3** ce asigură persistența și stocarea datelor. Folosindu-se de limbajul SQL, abstractizările la nivelul implementării sunt relativ ușor de făcut, scurtând timpul de implementare.

2.3 Securitate

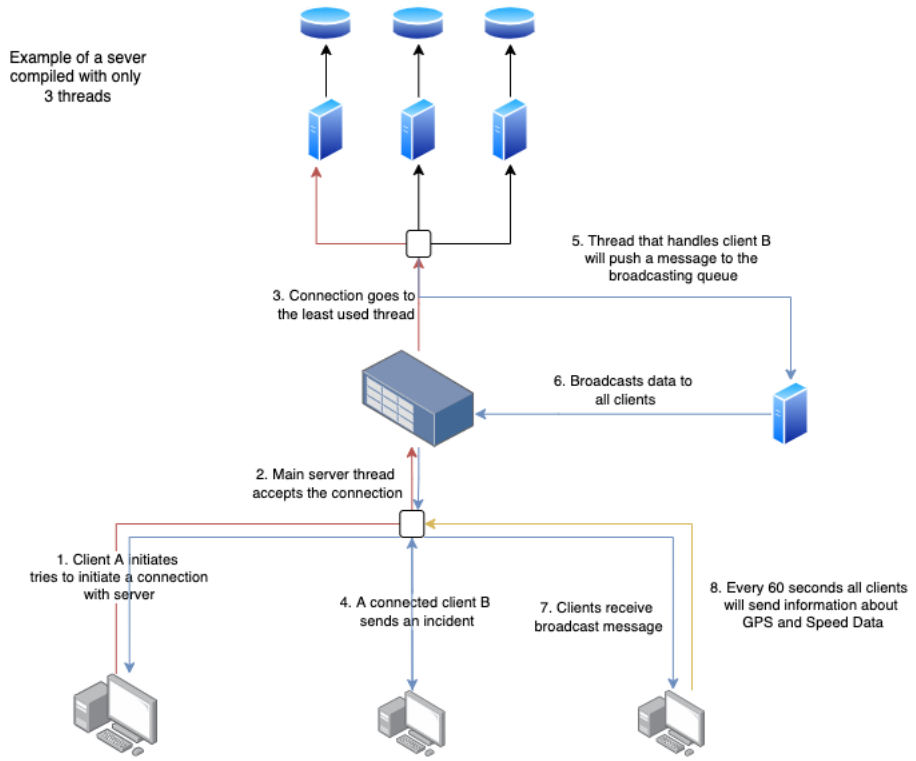
Din motive de securitate, în momentul în care conexiunea între client și server s-a stabilit, vom aveam un set limitat de comenzi pe care un utilizator neautentificat le poate utiliza. Totodată stocarea parolele în plain-text nu este o soluție bună pentru zilele noastre, astfel că din aceeași vom folosi agoritmul de hashing **SHA-256**.

2.4 Simularea Senzorilor

În încercarea de a simula un mediu cât mai realist, am adăugat un feature prin care se pot simula doi senzori, unul de viteză și unul de locație, strâns legați între ei. Aceștia pot fi citiți, returnând date cu un anumit nivel de corență între ele.

3 Arhitectura aplicatiei

3.1 Diagrama aplicației



Pentru a putea explica diagrama de mai jos vom împărții în două secțiuni: clientul și serverul

3.2 Arhitectura Clientului

Deși clientul nu are o arhitectură concurentă în adevăratul sens al cuvântului, acesta suportă mai multe operații aproximativ simultan, prin folosirea funcției select și a alertelor. Următorii pași explică cum funcționează clientul:

1. Inițial părintele încearcă să stabilească o conexiune cu serverul, folosind socket-uri. (pasul 1)

2. Dacă serverul acceptă conexiunea, clientul porneste un loop care va rula, cel mult atata timp cat serverul este deschis, asteptand atat date de la server cât și date de tip I/O. Implementarea acesta este posibilă întrucât într-un sistem UNIX citirea I/O este realizată tot printr-un fișier.

3. La un interval de timp prestabilit se vor trimite informațiile despre viteză și locație (pasul 8)

4. La un alt interval de timp, clientul va primi informații media (dacă a selectat această opțiune)

5. Daca un client raportează un incident serverul îl va procesa și va încerca să facă boardcast la toți clienții. Modul în care evenimentul aceasta se întamplă poate să fie urmărit pe diagrama de mai sus, urmărind pașii de la 4 la 7.

3.3 Arhitectura Serverului

Severul folosește tot o arhitectură concurentă, prin care reușește să suportate atât mai multe conexiuni simultan, cât și să răspundă la mai mulți clienți în același timp.

1. La deschiderea serverului se formează un thread pool de N threaduri (numarul fiind stabilit încă din timpul compilării) și un thread care va fi folosit pe post de broadcaster.

2. Când un client încearcă să se conecteze la server, threadul principal va accepta conexiunea și va alege thread-ul cel mai puțin utilizat (cu cele mai puține conexiuni) după care îl va adăuga în lista lui.

3. Threadurile care aparțin thread poll-ului folosesc **Multiplexarea I/O** realizată cu ajutorul funcției select. Totodată pentru a ne asigura că orice conexiune nouă este acceptată cât mai repede, selectul are un timeout de o secundă.

4. Threadul care se ocupă de funcția de broadcast folosește atât alarme, cât și conditional wait. Pentru a nu solicita inutil sistemul, thread-ul de broadcast rulează doar atunci când: un element a fost adăugat în coada de mesaje, sau o dată la 60 de secunde, când transmite informații media tuturor clienților.

4 Detalii de implementare

Întrucât am menționat destul de explicit modul în care se tratează cererile și cum este implementat clientul și serverul, în următoarele secțiuni voi detalia particularitățile aplicației.

4.1 Trimiterea Mesajelor

Pentru a putea comunica între client și server am implementat următorul format prin care mesajele sunt parsate (pe baza aceasta se pot recreea și pașii de construire):

1. Orice mesaj primit sau trimis este prefixat pe 4 biți cu lungimea acestuia (astfel că lungimea maximă a unui mesaj este de 9999 caractere). Prin

simulările făcute, am observat că nu ar exista un caz în care are trebui să trimitem un mesaj mai mare de 10KB. Un exemplu pentru format ar fi: **0010helloworld**

2. Fiecare element din mesaj este alcătuit din următoarele regului:
   (a) Începe cu |
   (b) (opțional) Urmează un sir de lungime de lungime 33 definit sub forma $< Identifier >:< Validator >$ (doar pentru comenzile care necesită autentificare). Atât validatorul, cât și identificatorul au o lungime fixă de 16 caractere fiecare. La finalul celor 33 de caractere se va afla un |, urmând mesajul propriu-zis.
   (c) În continuare avem o operație care va fi mereu tratată ca **string** ce se termină în '**:**' (aceasta poate lua un număr fix de valori)
   (d) Se citește un char care poate sa fie: **i** (int), **s** (string), **c** (coordonates),**d** (dict).
   (e) Și în final avem valoarea (ce se va extinde până la final, dat de lungime) - în cazul în care primul caracter citit nu a fost **n**.

   Un exemplu pentru o comandă ce nu necesită ca utilizatorul să fie autentificat ar fi:

   0051|Authenticate:d{suser:zabogdan,spassword:P@ssw0rd1}

   Iar pentru o comandă protejată:

   0084|6dfb834db2b08404:93cb6971e7084ec4|Incident:

   d{itype:1,cplace:[47.1731307,27.574399]}

3. Din schema enumerată mai sus, putem afirma că orice cerere am aveam, aceasta execută o singură comandă pe server.

Folosind această metodă de transmitere a datelor, putem să asigurăm atât validarea userului cât și suportarea mai multor tipuri de date, într-un format standardizat și ușor de modificat.

4.2 Validarea comenzilor

Întrucât ne-am asigurat că mesajele vin într-un format deja standardizat, înțeles atât de client cât și server, următorul pas ar fi să validăm datele primite. Astfel, pentru fiecare comandă pe care o putem executa pe server există un anumit set de reguli, care vor fi analizate imediat după parsarea mesajelor. Astfel că după parsare avem următoarele elemente:

```
parsedMessage: {
    token: {
        identifier: "6dfb834db2b08404",
        validator: "93cb6971e7084ec4"
    },
```

```
    command: "Incident",
    payload: "d{itype:1,cplace:[47.1731307,27.574399]}"
}
```

Acum, cel mai mult ne interesează fieldul **command**. Pe baza lui,vom calcula valoarea CRC, după care vom putea afla dacă comanda există și dacă necesită autentificare sau nu. În final, dacă trece de cele două verificări, putem începe să parsăm **payload-ul** și să facem validările de tip, specifice fiecărei comenzi.

4.3 Simularea senzorilor

În încercarea de a face o migrare a aplicației cât mai ușoară către un scenariu realist, am implementat doi senzori care generează date random, în următorul mod:

```
//SpeedSensor.h
class Speed
{
    private:
        int speedData;

        void Update();
    public:
        Speed();
        int Read();
};

//SpeedSensor.cpp
BTruckers::Client::Sensors::Speed::Speed()
{
    LOG_INFO("Initializing hardware SPEED sensor.");
    std::srand(std::time(nullptr));
    this->speedData = 50 + std::rand() % 30;
}

int BTruckers::Client::Sensors::Speed::Read()
{
    this->Update();
    return this->speedData;
}

void BTruckers::Client::Sensors::Speed::Update()
{
    std::srand(std::time(nullptr));
    int maxSpeedDiff = 1 + std::rand()%15;
    //here we can have a maxium of +-16 kmh.
```

```
        //we decide this value randomly
        this->speedData = std::max(0,
            this->speedData -
            std::rand() % maxSpeedDiff +
            std::rand() % maxSpeedDiff
        );
    }
```

Iar senzorul de viteză va avea aproximativ aceeași implementare, singura modificare fiind că vom calcula distanța parcursă bazată pe viteza de la momemntul în care citim din senzor.

4.4 Baza de date

Întrucât avem nevoie de persistența datelor, acestea trebuiesc salvate într-o bază de date. În schimb, pentru a putea să avem access la date și să realizăm operațiile standard (**CRUD**) trebuie să facem o clasă care acționează pe post de Wrapper la API-ul de *sqlite3*. Folosind clasa descrisă mai jos, reușim ca implementarea unui nou tabel sa fie facută într-un timp foarte scurt, focusându-ne mai mult pe logica de implementat pentru fiecare tabel, și nu pe aspectele de bază. Astfel că funcțiile pe care le vor moșteni clasele sunt definite astfel:

```
    //models/BaseModel.h
    class BaseModel
{
private:

    static bool Populate(BaseModel& current, SQLiteResponse& data);
    bool Execute(std::string sql);

public:

    //virtual functions to override
    virtual inline bool& IsLocked() = 0;
    virtual inline bool& IsInitialized() = 0;
    virtual inline bool& AllowUUIDChanges() = 0;
    virtual inline const std::string& GetTableName() = 0;
    virtual inline DB_FIELDS& GetDBFields() = 0;
    virtual inline BTruckers::Server::Core::DBHandler* GetConnection() = 0;

    //CRUD
    bool FindBy(std::string identifier, std::string value);
    bool Create();
    bool Update();
    bool Delete();
    bool GetRowsCount();
```

```
    //Update field
    bool UpdateField(std::string key, std::string value);
    std::string GetField(std::string key);

    //print the variables
    void Print();

};
```

4.5 Clientul

Deși flow-ul de pe client a fost explicat destul de bine, partea de client mai are nevoie de câteva explicații. Prin urmare, pentru a putea trimite și primi informații simultan, folosim instrucțiunea select, care citește I/O doar când apăsăm tasta ENTER (ceea ce ne scapă de problemele de tipul un utilizator scrie mai încet, a început să scrie și a uitat altceva). Totodată la lista de descriptori pe care selectul o urmărește, adăugam si socket-ul serverului, pentru a putea citi imediat informația. Prin această metodă garantăm că putem scrie și citi fără a avea blocaje.

Totuși a mai rămas o problemă. Clientul trebuie să transmită o dată la 60 de secunde date despre locația și viteza sa. Aceasta paote să fie ușor remediată folosind o alarmă. Dacă setăm o dată la 60 de secunde o alarmă, selectul va returna EINTR și noi vom putea implementa logica aferentă alarmei.

5 Concluzii

Aplicația **Monitorizarea Traficului** implementează un set de funcționalități destul de minimalist, existând loc de îmbunătățiri atât pe partea de securitate cât și pe cea de performanță a aplicației.

În primul rând, pentru a putea să scalăm aplicația ar trebui modificată baza de date din **sqlite3** într-o bază de date de tipul NoSQL (precum MongoDB), iar pentru stocarea datelor de autentificare (token-urile) ar trebui folosit un in-memory database precum Redis (întrucât necesită un număr relativ mare de interogări, mai precis este strâns legat de numărul de cereri pe care le face utilizatorul).

În al doilea rând, modul în care transmitem informațiile între client și server este prea particular pentru a se putea scala bine. Trecerea către un format de tipul JSON/XML este o necesitate. Un al aspect, destul de important, este și că folosim o conexiune necriptată. Aceasta, în schimb, se poate rezolva destul de ușor folosind OpenSSL.

În final, pe partea de securitate a aplicației, am putea implementa JSON Web Tokens folosind diferite Key Set-uri, ceea ce ar ajuta foarte mult la scalarea aplicației, în cazul în care ne-am dori mai multe servere să fie pornite simultan.

6 Bibliografie

1. SSL Server-Client using OpenSSL
2. Algorithm for offsetting coordonates by some amount of meters
3. Using ThreadPools in C