

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 și primi informații despre diverse evenimente rutiere, trimițând în mod automat viteza și locația la un interval regulat de timp. Acesta poate să 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 componente 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 în 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 în 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 selectat opțiunea) despre vreme, evenimente sportive și ultimele prețuri la combustibil. În cazul în care este raportat un incident, serverul va notifica toți utilizatorii din regiunea în care s-a produs despre eveniment.

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

2 Tehnologii utilizate

2.1 Protocolul de comunicare

Comunicarea dintre client/clienti 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

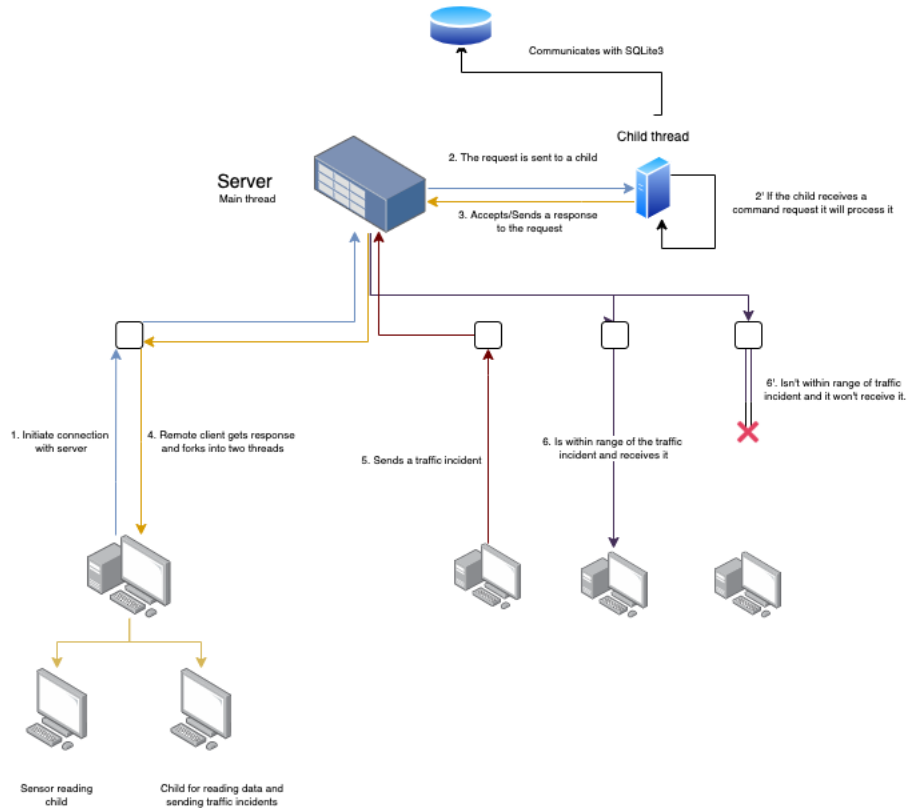
Din motive de securitate, în momentul în care conexiunea între client și server s-a stabilit, vom crea un nou context de tipul SSL, transmitând astfel datele criptate. Pentru acesta vom folosi librăria *OpenSSL* din C++. Totodată stocarea parolele în plain-text nu este o soluție bună pentru zilele noastre, astfel că din aceeași librărie vom folosi algoritmul 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 aplicației

3.1 Diagrama aplicației



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

3.2 Arhitectura Clientului

După cum se poate observa încă din diagramă clientul este bazat pe o arhitectură concurentă, având următoarele caracteristici:

1. Inițial părintele încearcă să stabilească o conexiune cu serverul, folosind socket-uri.
2. Dacă serverul acceptă conexiunea, în client se va forma un nou thread care va folosi socketul părinte să trimită informațiile despre viteză și locație, în timp ce părintele va putea să trimită și să primească informații rutiere sau media.
3. La un interval de timp prestabilit se vor trimite informațiile despre viteză și locație

4. La un alt interval de timp, clientul va primi informații media (dacă a selectat această opțiune)
5. Dacă un client raportează un incident serverul îl va procesa și va încerca să facă boardcast la toți clienții aflați în aria în care s-a raportat incidentul. După cum se poate observa în pașii 6 și respectiv 6' informația nu este transmisă la toți clienții.

3.3 Arhitectura Serverului

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

1. Prin **multiplexarea I/O** se vor monitoriza descriptorii socketurilor asociate clienților remote, creându-se un nou thread la momentul în care o cerere nouă ajunge. Astfel putem suporta mai multe cereri simultan.
2. Primul pas prin care o cerere trece este procesul de autentificare, urmând ca mai apoi să fie validată comanda, iar în final să se încerce procesarea informației.
3. Pentru validarea autentificării folosim tokenuri generate random, care sunt stocate în baza de date și sunt atribuite unui utilizator. Pentru a putea evita orice fel de manipulare a datelor, tokenul este format din două componente: Identificatorul (care se află stocat în plain text în baza de date) și Validatorul (care se află într-o formă **hashed**, utilizatorul având singura copie a acestuia.)
4. Dacă autentificarea a avut succes se va verifica dacă comanda dată de utilizator există, iar în caz afirmativ se va executa logica.
5. În final, indiferent de rezultatul execuției, serverul va trimite un răspuns clientului, urmând ca mai apoi să fie închis thread-ul ce s-a ocupat de răspuns.

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 recrea 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 reguli:
 - (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 trată ca **string** ce se termină în ':' (aceasta poate lua un număr fix de valori)
 - (d) Se citește un char care poate sa fie: **a** (array), **i** (int), **s** (string), **c** (coordonates), **d** (dict) **n** (null - parsarea se operește aici, nu avem payload).
 - (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, putem afla dacă comanda există, 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
#include <cstdlib>
#include <ctime>

class SpeedSensor : public Sensor{
private:
    int currentSpeed;
    Update();
public:
    SpeedSensor();
    Read();
}

//SpeedSensor.cpp
SpeedSensor::SpeedSensor(){
    std::srand(std::time(nullptr));
    //maxium car speed will be 200km/h
    this->currentSpeed = std::rand()%200;
}
SpeedSensor::Read()
{
    //we will first update the speed
    this->Update();
    //then return the new one
    return this->currentSpeed;
}
```

```

SpeedSensor::Update()
{
    //the maximum difference that can happen
    int maximumSpeedDifference = std::rand()%50;
    //decide weather it will be increasing or
    //decreasing speed
    this->currentSpeed = MAX(0, this->currentSpeed +
                           std::rand()%maximumSpeedDifference -
                           std::rand()%maximumSpeedDifference);
}

```

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*. Funcțiile pe care le vor moșteni clasele sunt definite astfel:

```

//SQLWrapper.h
class SQLWrapper{
private:
    //fiecare clasa va avea un UUID
    std::string UUID = "";
    SQLWrapper();
public:
    //va update fieldurile clasei
    virtual bool Find(
        std::string key,
        std::string value) = 0;
    //va modifica adauga o linie baza de date
    //cu variabilele nenule din clasa
    virtual bool Insert() = 0;
    //va updata linia din clasa cu variabilele
    //nenule din clasa
    virtual bool Update() = 0;
    //va sterge linia din tabel cu UUID-ul din
    //clasa
    virtual bool Delete() = 0;
}

```

Astfel, toate tabelele din baza noastră de date vor avea o clasă ce va asigura

funcțiile esențiale. Obiectele vor fi populate cu date la inițializare, sau chiar ulterior, folosind funcția `Find()`;

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, generarea unor thread-uri pentru fiecare cerere poate deveni destul de exhaustivă, astfel că nu am putea scala prea mult aplicația fără a avea probleme majore. O implementare mai bună ar fi prin folosirea unor thread poll-uri, articol ce poate fi regăsit la punctul 3 din Bibliografie.

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 coordinates by some amount of meters
3. Using ThreadPools in C