ProxyWebServer

Universitatea din București
Departamentul de Informatică
Instrumente și tehnici de bază în informatică

Banică Mihai Ovidiu

Condurache Daniel Flor

2025-2026

Cuprins

Ce e un proxy

Un server proxy este un sistem ce se plasează între client și un server pe care îl are ca destinație. Proxy-ul are rolul de a gestiona, redirecționa sau filtra cererile de rețea înainte să ajungă la serverul final.

Probleme

Utilizarea comunicării directe cu un server poate genera mai multe probleme:

1. Fără un proxy, clientul este expus la atacuri spre exemplu: malware, phishing, atacuri DDoS.
2. Nu există filtre pentru conținut periculos.
3. Administrarea și monitorizarea traficului devin dificile: nu se poate vedea cine accesează ce resurse sau când.
4. Se pot face cereri repetate la acelaşi server ceea ce consumă bandwidth ce duce la răspunsuri mai lente sau la supraîncărcarea serverelor.
5. Pentru a accesa acelaşi lucru se face de fiecare dată câte o cerere în loc să se salveze informația.

Soluții

Accesând un server printr-un proxy IP-ul clientului nu mai e vizibil, adaugând un pic de securitate.

În scopul proiectului nostru, folosim o interfață minimală ca să introducem un URL, apoi salvăm, local in cache (folosind politica LRU), pagina web ce dorim să o accesăm, astfel reducând traficul către server.

Pentru creerea serverul nostru, am utilizat elemente de limbaj javascript, hmtl, shell script, WSL/ Maşină virtuală, serverul si funcționalitatea sa rulând pe Linux.

Proiectul poate fii utilizat pe orice calculator, singura limitare fiind a sistemului de operare (Linux). Aceasta nu este un impediment deoarece se poate folosii o maşină virtuală sau un terminal de tip WSL (pentru Windows).

Design

Există mai multe componente ce fac serverul să funcţioneze, aceestea fiind:

Pagina html (webProxy.html)

Este interfaţa principală cu care utilizatorul interacţioneaza. Are un design minimalist cu un input tip text şi un buton de fetch care iniţializează cererea către serverul proxy. Mai apoi pagina la care s-a dat fetch se deschide tot aici.

Elemente de javascript (server.js)

Ruleaza un server local pe portul 3000, preia URL-ul introdus şi face legătura către backend prin apelarea scriptului fetch_page.sh. Gestionează răspunsurile de la server, afişând în browser resursă descărcată sau notificări privind statusul cererii.

Scripturile shell

setup_page.sh

Creează fişierul proxy_cache cu o dimensiune maximă de 100MB în directorul /mnt adică în memoria RAM. Este nevoie să fie rulat înainte de a face fetch la o pagină.

fetch_page.sh

Aici se verifică daca resursa cerută se află în cache, dacă există o deschide de acolo, iar dacă nu o descarcă folosind wget şi o salvează în cache.

Interacţiunea între aceste componente poate fi cel mai uşor interpretată printr-o analogie cu o casă. Html-ul este exteriorul, javascript reprezintă instalaţiile, iar scripturiile shell sunt fundaţia casei.

De ce wget?

Am ales să folosim wget in loc de alte metode cum ar ficu curl, librării native node.js sau baze de date, deoarece astfel putem observa cum o pagina web devine un fişier concret îm memorie.

Un exemplu ar fii comparaţia cu instrucţiunea curl, unde aceasta este mai orientată pe transferul brut de date decât pe descărcarea de fişiere ceea ce nu reprezintă direcţia proiectului şi anume stocarea locală a unei pagini web.

Astfel este mai uşor de gestionat un cache prin comenzi de fişiere decât să scriu o bază de date complexă.

Deşi din metodele menţionate anterior, în funcţie de scopul proiectului, poate fi mai optimă folosirea unei baze de date sau a unei librarii deja existente, dar acestea necesită mai multe resurse ce ar face funcţionarea serverului nostru (care este relativ la scară mică şi experimentală) mult mai costisitoare ca memorie şi mai dificil de implementat.

Implementare

server.js

Importarea librăriei Express, care ne ajută pentru a gestiona rute, cereri HTTP şi fişiere într-un mod simplu.

De asemena importăm şi modulele :

- “child_process”, care permite serverului să ruleze şi să execute script-uri bash
- "path", care este o utilitate ce ne ajută să lucrăm cu path-urile de fişiere

```
const express = require("express");
const { execFile } = require("child_process");
const path = require("path");
```

Următoarea comandă este cea care ne creează instanța aplicațtiei noastre. Practic este obiectul unde vom definii regulire serverului.

```
const app = express();
```

```
const url = req.query.url;
    if (!url) {
        return res.status(400).send("URL parameter is required");
    }
```

Apelarea scriptului fetch_page.sh

```
execFile(path.join(__dirname, "fetch_page.sh"), [url], (error, stdout, stderr) =>
```

Tratarea erorilor

```
if (error) {
            console.error(stderr);
            return res.send("Wget error");
        }
```

Trimitera fișierului către client

```
const filepath = path.resolve(stdout.trim());
res.sendFile(filepath);
```

Pornirea serverului

```
app.listen(3000, () => console.log("Server running on port 3000"));
```

fetch_page.sh

Preluarea URL

```
URL="$1"
if [ -z "$URL" ]; then
    echo "Eroare: Nu a fost furnizat niciun URL." >&2
    exit 1
fi
```

Funcția LRU care șterge primul fișier în ordinea cronologică și face loc pentru cel nou când ajunge la o limită de 85%.

```
clean_lru() {
    local THRESHOLD=85

    local usage=$(df "$CACHE_DIR" | awk 'NR==2 {print $5}' | sed 's/%//')

    while [ "$usage" -gt "$THRESHOLD" ]; do
        local oldest=$(ls -tu "$CACHE_DIR"/*.html 2>/dev/null | tail -1)

        if [ -n "$oldest" ]; then
            echo "[LRU] Cache aproape plin ($usage%). Se elimină: $(basename
"$oldest")" >&2
            rm -f "$oldest"
            usage=$(df "$CACHE_DIR" | awk 'NR==2 {print $5}' | sed 's/%//')
        else
            break
        fi
    done
}
```

Validarea existenței paginii în cache

```
if [ -f "$CACHE_FILE" ]; then
    echo "[Proxy] CACHE HIT pentru $URL" >&2

    touch -a "$CACHE_FILE"

    ln -sf "$CACHE_FILE" "$TMP_LINK"

    echo "$CACHE_FILE"
    exit 0
fi
```

Comanda wget porneşte descărcarea paginii cu anumite condiţii şi anume:

- -q => Nu afişează progresul în terminal (quiet)
- -0 => Salvează conţinutul direct in cache (tmpfs)
- & => Acesta trimite procesul in fundal astfel încât scriptul nu se opreşte la acest pas până e gata şi continuă executarea

WGET_PID=$! Salveaza ID-ul procesului si va fii folosit să verifice dacă wget s-a terminat cu success sau a dat de o eroare.

În instrucţiunea inotifywait -e close_write "$CACHE_FILE" >/dev/null 2>&1 aici se întâmplă mai multe lucruri deodată aşa că să le luăm pas cu pas

- Inotifywait => este un utilitar al kernelului ce urmăreşte ce se întâmplă cu fişierul nostru (pagina web)
- -e close_write => Acesta este o barieră ce opreşte scriptul până ce wget deschide fişierul pentru scriere

Comanda wait $WGET_PID se asigură ca procesul wget s-a terminat. Deşi inotifywait ne-e spus ca fişierul e gata, având si wait, putem să colectăm codul de ieşire al wget astfel încat putem observa dacă are eroare sau nu.

```
wget -q -O "$CACHE_FILE" "$URL" &
WGET_PID=$!
inotifywait -e close_write "$CACHE_FILE" >/dev/null 2>&
wait $WGET_PID
```

Probleme întâmpinate

De-a lungul procesului am întâmpinat mai multe probleme legate de pornirea serverului, conflicte între Windows şi WSL, probleme de compatibilitate între fişiere, probleme de sincronizare cu GitHub şi desigur problemele de sintaxă ce au fost rezolvate pentru a rula serverul.

Din toate aceste cea mai mare problemă a fost cea de a porni local serverul folosind node.js. Pentru că am lucrat cu WSL, în loc de Linux (ca OS)

sau cu o maşină virtuală, au apărut probleme de compatibilitate mai ales când trimiteam fişierele între noi. Uneori, acestea erau înregistrate ca fişiere pentru Windows şi nu funcţionau.

Totuşi după un research amănunţit ne-am am folosit un terminal Linux rulat din VS code, pentru a convertii fişierele în formal unix, scăpând de probleme.

O altă problemă notabilă a fost cea de sincronizare a fisierelor cu GitHub. Am folosit aplicaţia de Desktop pentru a putea lucra în paralel la proiect, iar când unul din noit termina o parte, foloseam push pentru a updata proiectul direct pe Git. Dintr-un motiv anume aceste schimbari nu aveau loc în urma sincronizării, deşi ar fii trebuit.

După mai multe încercări, am reuşit să ne sincronizăm fişierele cu tot cu modificăriile aferente.

Ultima dificultate notabilă a fost întampinată doar de unul din noi (Mihai), aceasta fiind faptul ca după ce proiectul a fost finalizat, am încercat să redeschid serverul si nu a functionat.

În continu serverul nu putea să se pornească din cauza unei erori ce nu identifica node.js. Prin mult research am ajuns la ceea ceva straniu şi anume ca Windows-ul folosea o variantă de node.js pentru procesorul de tip x86x64, dar WSL deşi recunoştea procesorul ca fiind x86x64 refuza să ruleze node.js.

Rezolvarea a venit prin a reinstala WSL, a redownloada proiectul de pe GitHub, a-l convertii din nou în format unix şi reinstalarea a node.js. Nu aş putea explica cum de acest conflict s-a produs, cu toate că din fericire am putut să-l rezolv.

Limităriile proiectului nostru sunt prin faptul ca noi nu avem toata funcţionalitatea unui server proxy de a accesa mai multe file sau masuriile de securitate ale unuia. Proiectul nostru fiind mai degrabă un prototip al unui server proxy, care se axează pe stocarea locala a unei pagini web si vizualizarea acesteia.

Concluzie

Prin realizarea acestui proiect am învăţat ce este un server proxy, funcţionalităţi GitHub, cum lucreaza WSL-ul în interiorul Windows-ului.

De semenea am că munca în echipă makes everything awesome!