

Proxy WebServer

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

Banică Mihai Ovidiu

Condurache Daniel Florin

2025-2026

Cuprins

ProxyWebServer.....	1
Ce e un proxy.....	2
Probleme	3
Soluții.....	3
Design	4
Pagina html (webProxy.html)	4
Elemente de javascript (server.js)	4
Scripturile shell.....	4
setup_page.sh	4
fetch_page.sh.....	4
De ce wget?	5
Implementare.....	5
server.js.....	5
fetch_page.sh.....	7
Probleme întâmpinate	9
Concluzie	10

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 același server ceea ce consumă bandwidth ce duce la răspunsuri mai lente sau la supraîncărcarea serverelor.
5. Pentru a accesa același 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, adăugând 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, html, shell script, WSL/ Mașină virtuală, serverul si funcționalitatea sa rulând pe Linux.

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

Design

Există mai multe componente ce fac serverul să funcționeze, acestea fiind:

Pagina html (**webProxy.html**)

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

Elemente de javascript (**server.js**)

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

Scripturile shell

setup_page.sh

Creează fișierul 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 dacă resursa cerută se află în cache, dacă există o deschide de acolo, iar dacă nu o descarcă folosind wget și o salvează în cache.

Interacțiunea între aceste componente poate fi cel mai ușor interpretată printr-o analogie cu o casă. Html-ul este exteriorul, javascript reprezintă instalațiile, iar scripturile shell sunt fundația casei.

De ce wget?

Am ales să folosim wget în loc de alte metode cum ar fi curl, librării native node.js sau baze de date, deoarece astfel putem observa cum o pagină web devine un fișier concret în memorie.

Un exemplu ar fi comparația cu instrucțiunea curl, unde aceasta este mai orientată pe transferul brut de date decât pe descărcarea de fișiere ceea ce nu reprezintă direcția proiectului și anume stocarea locală a unei pagini web.

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

Deși din metodele menționate anterior, în funcție de scopul proiectului, poate fi mai optimă folosirea unei baze de date sau a unei librării deja existente, dar acestea necesită mai multe resurse ce ar face funcționarea serverului nostru (care este relativ la scară mică și experimentală) mult mai costisitoare ca memorie și mai dificil de implementat.

Implementare

server.js

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

De asemenea importăm și modulele :

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

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

Următoarea comandă este cea care ne creează instanța aplicației noastre. Practic este obiectul unde vom defini regulile 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");
}
```

Trimiterea 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 pornește descărcarea paginii cu anumite condiții și anume:

- -q => Nu afișează progresul în terminal (quiet)
- -O => Salvează conținutul direct în cache (tmpfs)
- & => Acesta trimite procesul în fundal astfel încât scriptul nu se oprește la acest pas până e gata și continuă executarea

WGET_PID=\$! Salvează ID-ul procesului și va fi folosit să verifice dacă wget s-a terminat cu succes sau a dat de o eroare.

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

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

Comanda wait \$WGET_PID se asigură că procesul wget s-a terminat. Deși inotifywait ne-e spus că fișierul e gata, având și wait, putem să colectăm codul de ieșire al wget astfel încât 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 și WSL, probleme de compatibilitate între fișiere, probleme de sincronizare cu GitHub și 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 mașină virtuală, au apărut probleme de compatibilitate mai ales când trimiteam fișierele între noi. Uneori, acestea erau înregistrate ca fișiere pentru Windows și nu funcționau.

Totuși după un research amănunțit ne-am am folosit un terminal Linux rulat din VS code, pentru a convertii fișierele în format unix, scăpând de probleme.

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

După mai multe încercări, am reușit să ne sincronizăm fișierele cu tot cu modificările aferente.

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

În continuu serverul nu putea să se pornească din cauza unei erori ce nu identifica node.js. Prin mult research am ajuns la ceea ceva straniu și anume ca

Windows-ul folosea o variantă de node.js pentru procesorul de tip x86x64, dar WSL deși recunoștea 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 și reinstalarea a node.js. Nu aș putea explica cum de acest conflict s-a produs, cu toate că din fericire am putut să-l rezolv.

Limitările proiectului nostru sunt prin faptul ca noi nu avem toata funcționalitatea 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 și vizualizarea acesteia.

Concluzie

Prin realizarea acestui proiect am învățat ce este un server proxy, funcționalități GitHub, cum lucrează WSL-ul în interiorul Windows-ului.

De asemenea am că munca în echipă makes everything awesome!