Universitatea Tehnică "Gheorghe Asachi", Iași

Facultatea de Automatică și Calculatoare

Sistem distribuit de stocare și versionare a fișierelor

- Raport intermediar II pentru lucrarea de diplomă -

Student : Stratulat Ștefănel Constantin (1409A)

Îndrumător : Buțincu Cristian

Anul universitar 2020 – 2021

Cuprins

Descriere

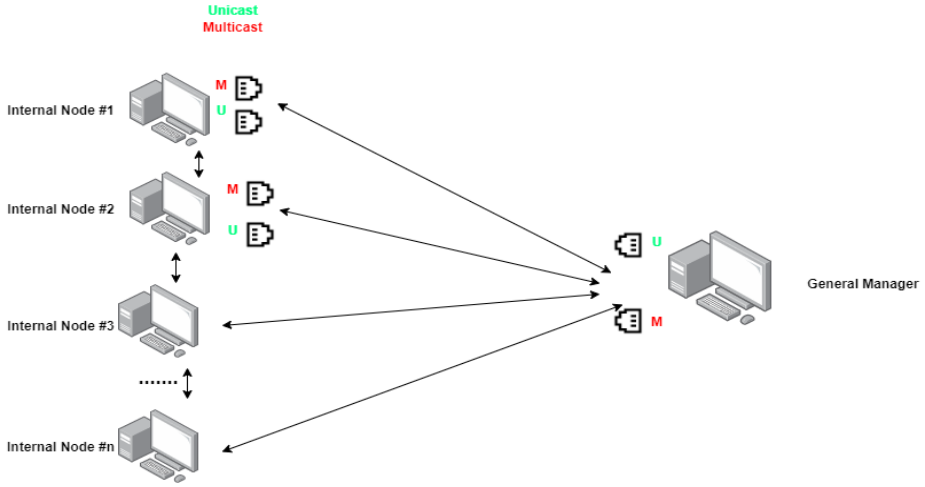
Lucrarea are ca și obiectiv dezvoltarea unui sistem distribuit de stocare și versionare a fișierelor. Mai exact, se are în vedere un sistem în care utilizatorul își poate încărca fișierele astfel încât să le poată accesa ulterior din orice loc în care are acces la aplicație și la o conexiune la internet. Totodată, se asigură versionarea fișierelor, astfel încât utilizatorul să poată analiza toate modificările suferite de fișiere.

Proiectul adresează, în primul rând, disponibilitatea datelor. În acest sens, în funcție de tipul utilizatorului și de importanța oferită unui fișier încărcat, se va asigura replicarea automată a fișierului pe mai multe noduri ale sistemului și menținerea numărului de replici disponibile pe toată durata de viață a fișierului (dacă un nod al sistemului va pica, se va asigura replicarea automată a fișierului pe alt nod; dacă un nod al sistemului repornește, se va asigura ștergerea fișierului de la unul dintre noduri, cel mai încărcat).

Proiectarea hardware/software a aplicației

Aplicația își propune să ofere o soluție de stocare a fișierelor, soluție a cărei principală caracteristică este disponibilitatea, care se poate asigura printr-un mecanism de persistență a fișierelor pe mai multe sisteme de calcul. Astfel, nu mai depindem de o singură entitate de stocare, a cărei coruperi poate duce la pierderea irecuperabilă a datelor, ci avem fișierele pe mai multe noduri astfel încât, la coruperea unuia dintre noduri, să putem extrage datele de pe unul dintre celelalte noduri.

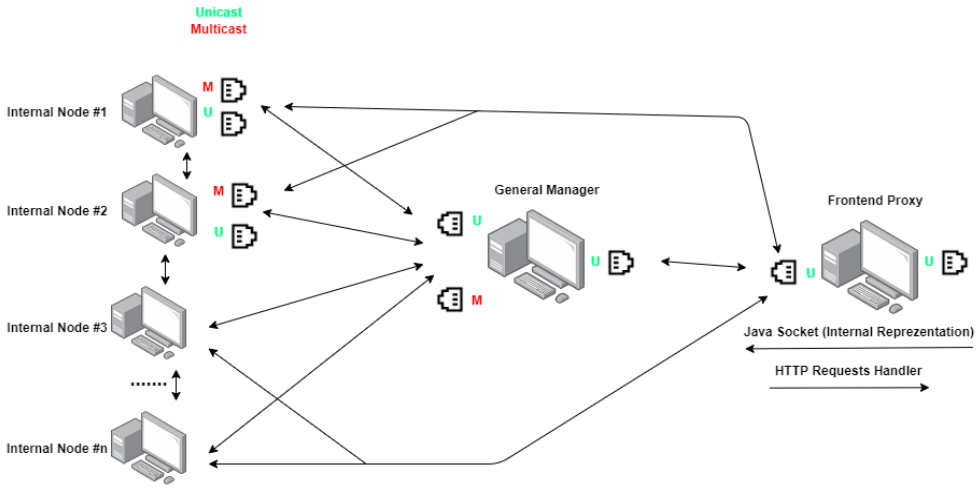
Așadar, sistemul distribuit care cuprinde logica internă de stocare și prelucrare a fișierelor are o arhitectură de tip Master-Slave . Componenta master este reprezentată de nodul general, care se ocupă de toate prelucrările fișierelor și care ține evidența nodurilor interne (slave) care vor stocare datele. Nodul general va cunoaște în permanență statusul nodurilor interne (dacă este activ și ce fișiere sunt stocate). Totodată, nodul general va cunoaște toate fișierele care există în sistem, împreună cu nodurile pe care sunt stocate. Astfel, cunoscând totalitatea fișierelor și statusul nodurilor, nodul general va putea lua decizii de replicare sau eliminare a fișierelor (dacă un nod devine inactiv), va putea decide pe ce noduri se poate stoca un nou fișier care apare în sistem. În aceeași măsură, nodul general va realiza și versionarea fișierelor, înregistrând fiecare prelucrare a fișierelor. Fiind un sistem distribuit în care comunicarea se realizează prin rețea, nodurile interne pot exista pe stații diferite în locații diferite.



În sistem există două modalități principale de comunicare în rețea, motiv pentru care avem și două categorii de socket-uri la nivel de implementare. În primul rând, avem o comunicare de tip unicast, realizată doar între nodul general și nodul intern sau între două noduri interne (transfer de fișiere, feedback, cereri de prelucrare : eliminare, redenumire, replicare, etc) și o altă comunicare de tip multicast realizată între toate nodurile sistemului. În cazul master-slave, folosim această comunicație în două moduri. De la nodurile interne către nodul general se trimit periodic heartbeat-uri astfel încât să se confirme disponibilitatea nodului intern și să se transmită statusul stocării fișierelor. În sens invers, nodul general trimite mesaje periodice tuturor nodurilor interne, prin care solicită statusul integrității fișierelor (calcularea crc-ului fiecărui fișier), astfel încât să se identifice eventualele coruperi are unității de stocare. Nodurile interne și nodul general vor fi dezvoltate folosind limbajul Java, împreună cu toate funcționalitățile de bază oferite pentru comunicarea în rețea (socket-uri, serializare/deserializare, prelucrarea sistemului local de fișiere, etc.)

Ținând cont de faptul că sistemul va stoca fișierele unor utilizatori, avem nevoie de o componentă de tip client, care va face disponibile către acesta, totalitatea operațiilor permise. Componenta de tip client va fi o aplicație web, dezvoltată folosind framework-ul React, în limbajul JavaScript. Având în vedere faptul că toate componentele de logică internă (nodul general și nodurile interne) sunt dezvoltate cu componente de bază Java (socket-uri) , iar comunicarea se realizează prin pachete de date ce conțin obiecte în reprezentarea internă Java (beans), o comunicare directă dintre

componentele de logică internă și componenta client este mai greu de realizat. Din acest motiv, a fost dezvoltat o componentă proxy, care să faciliteze această comunicare.

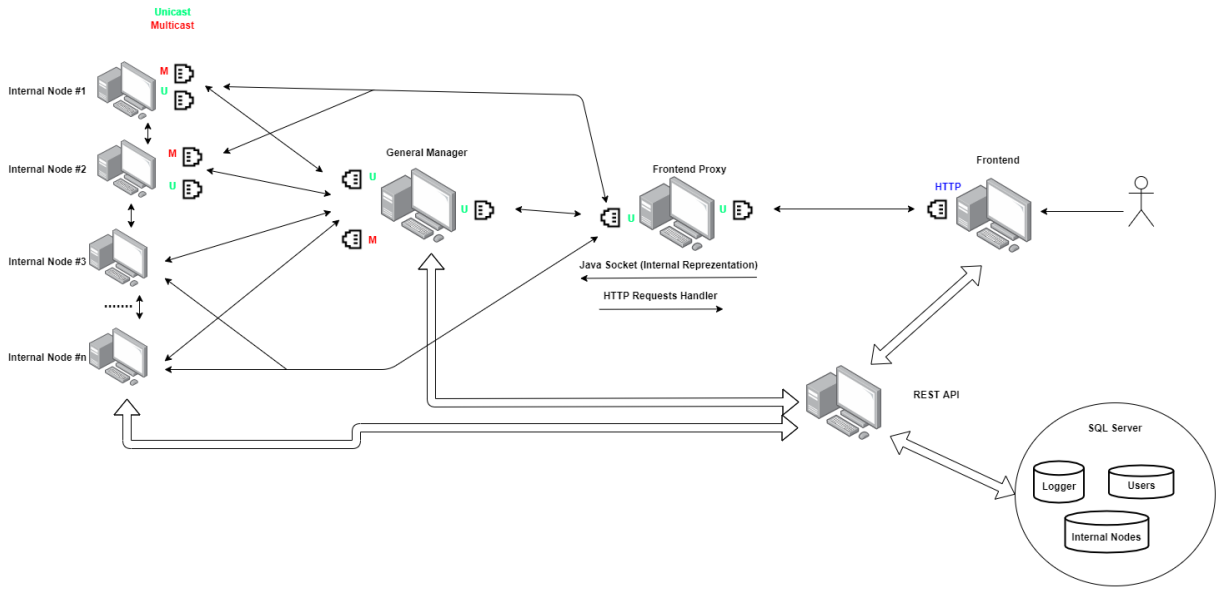


Componenta de tip proxy va conține un modul de interpretare de cereri HTTP, astfel încât să poată comunica cu componenta de tip client dezvoltată în React. În urma interpretării cererilor HTTP, aplicația proxy va instanția obiectele Java Beans corespunzătoare cererii și va trimite mai departe către nodul general și (unde este cazul) către nodurile interne. Comunicarea dintre proxy și nodul general are un scop destul de evident, având în vedere că nodul general se ocupă de toate prelucrările fișierelor, însă comunicarea dintre proxy și nodurile interne este mai puțin evidentă, aceasta având ca scop trimiterea fișierelor. Nodul general are doar rolul de delegare, nu și de stocare, motiv pentru care nu mai trecem conținutul fișierelor prin nodul general, ci le trimitem direct către nodurile interne pe baza deciziilor luate de nodul general.

Principalul argument pentru folosirea unui proxy care să transforme cererile HTTP primite de la aplicația client, în obiecte cu reprezentare internă Java (beans) este principiul de separare a responsabilităților. Nodul general are doar rolul de a realiza logica de administrare a fișierelor și a nodurilor interne, fără a supraîncărca și cu o componentă de tip server HTTP, pentru prelucrarea cererilor de acest tip. Având nevoie de comunicare și cu nodurile interne, am avea nevoie de această componentă HTTP și pe nodurile interne. Un alt argument ar fi faptul că este mai eficient să transferăm conținutul binar al fișierelor folosind aceeași tehnologie (Java), în loc să avem tehnologii diferite (Java – React Js & HTTP).

Totodată, avem nevoie de un nivel de persistență a datelor de identificare ale utilizatorilo aplicației (date de autentificare, autorizare, contact, etc.). Aplicația are două tipuri principale de utilizatori : utilizator obișnuit și admin al sistemului. Pentru fiecare tip de utilizator, vor fi disponibile funcționalități diferite în interfață astfel că, utilizatorul obișnuit va avea acces la propriile fișiere, iar adminul sistemului va avea disponibilă o fereastră de depanare, unde va vizualiza statusul nodurilor și o evidență (log) a tuturor evenimentelor apărute în sistemul intern. Astfel, există disponibilă o componentă de tip REST API, dezvoltată folosind Spring Framework, cu o conexiune la un sistem de baze de date relationale SQL. Această componentă va suporta toate operațiile de tip CRUD asupra datelor utilizatorilor, asupra nodurilor interne (datele de identificare ale acestora: adresă, locație,

memorie) și asupra istoricului evenimentelor. În ceea ce privește utilizatorii, pentru a securiza datele acestora în transferul de date, se vor folosi JWT (JSON Web Token), identitatea utilizatorului pentru care se face cererea plasându-se în header-ul http authorization, în format criptat, conform standardului JWT.



Rezultate intermediare obținute și soluții de rezolvare

Rezultatele intermediare ce vor fi prezentate vor analiza componenta de logică internă, de stocare si prelucrare a fișierelor.

În primul rând, atunci când un utilizator dorește să stocheze un fișier, se trimite o cerere de la aplicația client cu datele de bază despre fișier (nume, dimensiune, factor de replicare) către nodul general. Acesta, după ce verifică statusul stocării alcătuit pe baza heartbeat-urilor de la nodurile interne, întoarce un token ce va reprezenta nodurile ce vor putea stoca fișierul. După ce va primi acest token, frontend-ul va trimite fișierul către primul nod din acest lanț, care, la rândul lui, va trimite următorului, și așa mai departe. În acest timp, nodul general înregistrează noul fișier cu starea pending urmând ca, după ce primește confirmare că a fost stocat cu succes la cel puțin unul dintre noduri, să îl înregistreze ca valid.

```
Client nou conectat : [/127.0.0.1 : 8081]

User uploaded a new file with size : 219366996 and replication factor : 3
Generam token-ul..
====================================
127.0.0.5-127.0.0.2-127.0.0.3
====================================
Token-ul a fost trimis catre client : 127.0.0.5-127.0.0.2-127.0.0.3
Inregistram noul fisier.
Cerinta clientului a fost realizata..
Am primit un hearthbeat de la 127.0.0.3 : 8246 ...
```

În continuare, putem observa starea nodului general la un moment dat.

Storage Status Table va fi actualizat la fiecare heartbeat primit de la un nod, și va conține fișierele existente în sistem, nodurile pe care sunt stocate, împreună cu versiunea și hash-ul.

Content Table va fi populată la apariția unui nou fișier în sistem și conține starea fișierului de la ultima prelucrare. Pentru ca un fișier de la un nod să fie valid, trebuie ca datele din aceste două tabele (crc și versiunea) să corespundă.

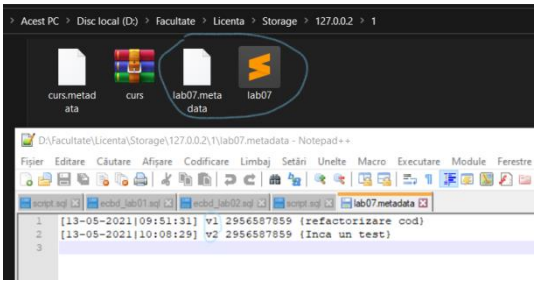
Node Storage Quantity Status va conține statusul stocării tuturor nodurilor interne, mai precis cantitatea de memorie ocupată/disponibilă.

În final, Replication Status este rezultatul buclei de verificare a consistenței celor două tabele. În cadrul acestei etape se iau decizii de replicare sau eliminare a unui fișier. (dacă un nod pică se face replicare, dacă sunt prea multe noduri se face eliminare).

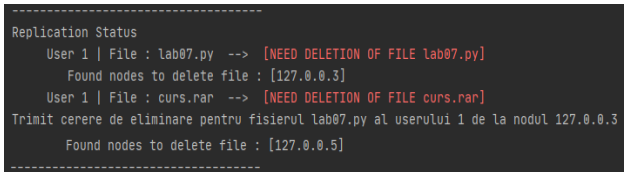
```
-----------------------------------
Storage Status Table
User id  : 1
    Filename : lab06.py
        127.0.0.4  [CRC : b039f353 | VersionNo : v1]
        127.0.0.2  [CRC : b039f353 | VersionNo : v1]
        127.0.0.3  [CRC : b039f353 | VersionNo : v1]
    Filename : curs.rar
        127.0.0.5  [CRC : 5b13a725 | VersionNo : v1]
        127.0.0.3  [CRC : 5b13a725 | VersionNo : v1]
        127.0.0.2  [CRC : 5b13a725 | VersionNo : v1]

-----------------------------------

-----------------------------------
Content Table
lab06.py [repl. : 3] [CRC : b039f353 | VersionNo : v1] [Status : [VALID]]
curs.rar [repl. : 3] [CRC : 5b13a725 | VersionNo : v1] [Status : [VALID]]
-----------------------------------

-----------------------------------
Node Storage Quantity Status
    127.0.0.5 --> 209.2 MB/100.0 GB
    127.0.0.4 --> 3.56 KB/100.0 GB
    127.0.0.3 --> 209.21 MB/100.0 GB
    127.0.0.2 --> 209.21 MB/100.0 GB
-----------------------------------



-----------------------------------
User Storage Quantity Status
-----------------------------------



-----------------------------------
Replication Status
     User 1 | File : lab06.py  -->  [OK].
     User 1 | File : curs.rar  -->  [OK].
-----------------------------------
```

În urma procesului de prelucrare a unui fișier (redenumire) ca urmare a cererii venite de la frontend, observăm, atât local în fișierul .metadata, cât și în aplicație, că a fost înregistrată o nouă versiune a fișierului. În cazul redenumirii, se salvează o înregistrare a vechii versiuni (cu vechiul nume) astfel încât, dacă între timp un nod care avea fișierul a fost oprit fără a înregistra redenumirea, când va fi repornit, va declara vechea versiune a fișierului, de care nu mai avem nevoie, urmând astfel să fie eliminată. Am ales această soluție (a șterge o variantă veche în loc de a redenumi la noua versiune) întrucât, în urma redenumirii, fișierul ar fi adus la noua versiune, rezultând astfel un număr prea mare de replici în sistem, ceea ce va rezulta în eliminarea uneia dintre acestea, ceea ce este echivalent.

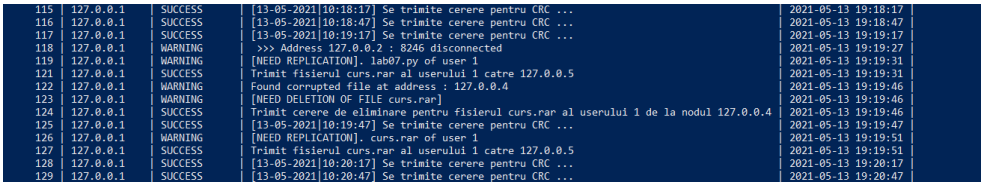
```
-----------------------------------
Storage Status Table
User id  : 1
    Filename : curs.rar
        127.0.0.5  [CRC : 5b13a725 | VersionNo : v1]
        127.0.0.3  [CRC : 5b13a725 | VersionNo : v1]
        127.0.0.2  [CRC : 5b13a725 | VersionNo : v1]
    Filename : lab07.py
        127.0.0.3  [CRC : b039f353 | VersionNo : v2]
        127.0.0.4  [CRC : b039f353 | VersionNo : v2]
        127.0.0.2  [CRC : b039f353 | VersionNo : v2]

-----------------------------------

-----------------------------------
Content Table
lab07.py [repl. : 3] [CRC : b039f353 | VersionNo : v2] [Status : [VALID]]
curs.rar [repl. : 3] [CRC : 5b13a725 | VersionNo : v1] [Status : [VALID]]
lab06.py [repl. : 0] [CRC : b039f353 | VersionNo : v1] [Status : [DELETED]]
-----------------------------------
```



Atunci când un nod intern pică, nodul general va depista acest lucru (nu mai primește heartbeat-uri de la acesta) și ia decizia de a replica pe alt terminal toate fișierele care erau disponibil pe acest nod. Observăm că se găsește o sursă de replicare (aleatoriu) și un nod care va putea stoca acel fișier (nodul cu cea mai multă memorie liberă). Se va trimite o cerere către nodul sursă, care va iniția comunicarea cu nodul destinație, trimițând fișierul. Se va trimite și fișierul de metadate, astfel încât să fie păstrate toate versiunile. În aceeași măsură, dacă nodul revine în sistem avem suprareplicare și se generează comandă de eliminare a fișierelor de la anumite noduri.



Observăm că toate evenimentele relevante la nivelul nodului general sau al nodurilor interne sunt înregistrate în baza de date de logging. Se înregistreaza adresa nodului care a înregistrat evenimentul, tipul și descrierea evenimentului și data la care a fost înregistrat.



115	127.0.0.1	SUCCESS	[13-05-2021|10:18:17] Se trimite cerere pentru CRC ...	2021-05-13 19:18:17
116	127.0.0.1	SUCCESS	[13-05-2021|10:18:47] Se trimite cerere pentru CRC ...	2021-05-13 19:18:47
117	127.0.0.1	SUCCESS	[13-05-2021|10:19:17] Se trimite cerere pentru CRC ...	2021-05-13 19:19:17
118	127.0.0.1	WARNING	>>> Address 127.0.0.2 : 8246 disconnected	2021-05-13 19:19:27
119	127.0.0.1	WARNING	[NEED REPLICATION]. lab07.py of user 1	2021-05-13 19:19:31
121	127.0.0.1	SUCCESS	Trimit fisierul curs.rar al userului 1 catre 127.0.0.5	2021-05-13 19:19:31
122	127.0.0.1	WARNING	Found corrupted file at address : 127.0.0.4	2021-05-13 19:19:46
123	127.0.0.1	WARNING	[NEED DELETION OF FILE curs.rar]	2021-05-13 19:19:46
124	127.0.0.1	SUCCESS	Trimit cerere de eliminare pentru fisierul curs.rar al userului 1 de la nodul 127.0.0.4	2021-05-13 19:19:46
125	127.0.0.1	SUCCESS	[13-05-2021|10:19:47] Se trimite cerere pentru CRC ...	2021-05-13 19:19:47
126	127.0.0.1	WARNING	[NEED REPLICATION]. curs.rar of user 1	2021-05-13 19:19:51
127	127.0.0.1	SUCCESS	Trimit fisierul curs.rar al userului 1 catre 127.0.0.5	2021-05-13 19:19:51
128	127.0.0.1	SUCCESS	[13-05-2021|10:20:17] Se trimite cerere pentru CRC ...	2021-05-13 19:20:17
129	127.0.0.1	SUCCESS	[13-05-2021|10:20:47] Se trimite cerere pentru CRC ...	2021-05-13 19:20:47

Dificultăți/provocări întâmpinate

Principalele dificultăți întâmpinate au fost cauzate de lipsa de experiență a lucratului cu proiecte de o asemenea dimensiune, unde o mică modificare poate impacta foarte mult. Problema vine, de fapt, din proiectarea artificială a componentelor, neținând cont de toate aspectele ce urmează a fi implementate. Concluzia ar fi că ar trebui să petrec mai mult timp proiectând și analizând sistemul în ansamblu înainte de a începe implementarea efectivă.