UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Sistem distribuit de stocare și versionare a fișierelor

LUCRARE DE DIPLOMĂ

Coordonator științific
Ș.l.dr.ing. Cristian Nicolae Buțincu

Absolvent
Ștefănel-Constantin Stratulat

Iași, 2021

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) STRATULAT ȘTEFĂNEL-CONSTANTIN,
legitimat(ă) cu C.I. seria XV nr. 264255, CNP 1981226336373
autorul lucrării SISTEM DISTRIBUIT DE STOCARE ȘI VERSIONARE A FIȘIERELOR

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea IULIE/2021 a anului universitar 2020-2021, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data Semnătura

Cuprins

Sistem distribuit de stocare și versionare a fișierelor

Ștefănel-Constantin Stratulat

Rezumat

În ultimii ani, odată cu evoluția tehnologică, prin creșterea semnificativă a puterii de procesare, stocare și transfer a datelor, cantitatea de informație generată și consumată de utilizatorii sistemelor de calcul a crescut exponențial, la volume anuale exprimate în miliarde de *terabytes*. Din acest motiv, se observă o reorientare a utilizatorilor către modalitățile de stocare în cloud, datorită flexibilității accesării datelor și a siguranței oferite. Astfel, sistemele de stocare trebuie să asigure disponibilitatea, integritatea, manevrabilitatea datelor, prin intermediul unui sistem fiabil care să poată fi foarte eficient scalat pentru a putea satisface nivelele ridicate de încărcare generate de creșterea numărului de utilizatori, a numărului de fișiere și a dimensiunii acestora. Totodată, experiența utilizatorului în accesul la date trebuie să fie foarte fluentă astfel încât fișierele și metadatele acestora să fie foarte ușor accesibile iar prelucrările să se poată efectua rapid și sigur.

Sistemul prezentat este un sistem distribuit de stocare și versionare de fișiere, care îndeplinește aceste cerințe, prin intermediul unei arhitecturi de tip master/slave, alcătuită dintr-un nod general master și mai multe noduri interne. Această arhitectură asigură disponibilitatea datelor prin faptul că fișierele utilizatorului sunt replicate pe mai multe noduri ale sistemului, într-un număr de replici specific tipului utilizatorului, număr ce se va menține constant pe toată durata de viață a fișierului, prin intermediul unor mecanisme de replicare sau eliminare, în cazul în care s-au produs erori la nivelul nodurilor interne. Sistemul asigură integritatea datelor prin verificări periodice a conținutului fiecărei replici a unui fișier, prin intermediul mecanismelor bazate pe coduri redundante ciclice. Toate aceste mecanisme vor fi posibile prin intermediul unei comunicații de tip multicast între nodurile sistemului, prin vehicularea unor mesaje de tip heartbeat.

Manevrabilitatea datelor va fi asigurată prin multitudinea operațiilor disponibile pentru toate fișierele utilizatorului. Fișierele vor putea fi încărcate și descărcate din sistem, vor putea fi modificate, redenumite și eliminate. Toate aceste operații vor presupune noi versiuni ale fișierului. Aceste versiuni vor fi înregistrate la fiecare prelucrare a fișierului și vor conține atât date de identificare și validare relevante pentru sistem, cât și date de descriere a versiunii, relevante pentru client. Fiind informații importante ale fișierului, se vor asigura toate mecanismele necesare menținerii metadatelor despre versiune alături de fișier.

Din punct de vedere al interacțiunii cu utilizatorul, se va expune o interfață web, care va suporta două tipuri de utilizatori. Utilizatorul obișnuit poate încărca, descărca, previzualiza și actualiza fișiere și versiuni ale acestora, iar administratorul sistemului are la dispoziție o consolă de monitorizare a sistemului, în care poate urmări întreaga activitate a nodurilor prin analiza stării sistemului și a evenimentelor generate în timp.

Introducere

Dezvoltarea tehnologică din ultimii ani a determinat creșterea volumului de informație generată și consumată de utilizatorii sistemelor de calcul. Evoluția și disponibilizarea serviciilor de stocare a datelor în cloud a determinat reorientarea utilizatorilor către această modalitate de stocare, mai ales că asigură disponibilitatea și integritatea datelor mult mai eficient decât modalitățile de stocare convenționale, iar accesul la date se poate realiza prin mai multe modalități, care nu sunt dependente de platforma de stocare a datelor.

Proiectul adresează un astfel de sistem de stocare a fișierelor, sub forma unui sistem distribuit cu o arhitectură de tip master/slave, alcătuită dintr-un nod manager general (master) și mai multe noduri interne (slave). O astfel de arhitectură asigură disponibilitatea datelor prin replicarea datelor pe mai multe noduri pentru a elimina dependența directă a datelor de o singură unitate de stocare. În această manieră, se asigură faptul că toate replicile fișierului vor avea același conținut și vor putea fi mereu furnizate către utilizator. Integritatea datelor se asigură prin verificarea periodică a conținutului și metadatelor fișierelor, astfel încât toate replicile să conțină aceeași stare a fișierului. Îndeplinirea acestor principii se va realiza printr-o comunicație de tip multicast dintre nodurile sistemului. Nodurile interne, care vor stoca replicile fișierelor, vor trimite mesaje de tip heartbeat la intervale regulate de timp, prin care vor furniza starea tuturor fișierelor conținute. Nodul general va trimite mesaje la intervale regulate de timp, prin care va solicita calcularea sumei de control prin intermediul codurilor redundante ciclice, pentru a asigura integritatea datelor. În funcție de rezultatele acestor interacțiuni și ținând cont de faptul că numărul de replici ale fișierelor trebuie să fie menținut constant pe întreaga durată de viață a fișierului, se vor lua decizii de replicare sau eliminare a unor replici de la nivelul nodurilor interne.

Sistemul va pune la dispoziția utilizatorului o multitudine de operații asupra fișierelor, de încărcare și descărcare, redenumire, modificare a conținutului și eliminare. Toate aceste operații presupun versiuni noi ale fișierelor. Versiunile vor conține date de identificare, care sunt relevante pentru sistem (suma de control) și date de descriere, care sunt relevante pentru utilizator (numărul versiunii, indici temporali despre momentul înregistrării versiunii și descrierea acesteia). Acestea vor fi persistate în fișierele de metadate, corespunzătoare fiecărui fișier și vor fi considerate de toate mecanismele de asigurare a disponibilității și integrității, alături de fișiere.

Accesul utilizatorilor la date se va putea realiza prin intermediul unei aplicații de tip web, cu o interfață utilizator care va expune atât modalități de încărcare, descărcare și previzualizare a fișierelor, cât și metode pentru prelucrarea acestora. Totodată, utilizatorul va putea modifica tipul contului de utilizator astfel încât să extindă cantitatea de memorie disponibilă și factorul de replicare al fișierelor. Această interfață utilizator va fi disponibilă și pentru administratorul sistemului. Acesta va avea la dispoziție o consolă de monitorizare, prin intermediul căreia va putea vizualiza și prelucra starea sistemului. Pentru a asigura securizarea datelor utilizatorilor, sunt folosite mecanisme de codificare și semnare digitală a datelor sensibile vehiculate între utilizator și sistem în cadrul cererilor de tip HTTP.

Lucrarea de față prezintă o imagine de ansamblu a acestui sistem. Mai întâi, se analizează sistemul din punct de vedere teoretic, prin descrierea domeniului, contextului și modului de abordare a temei. Apoi, se realizează o analiză a aplicațiilor similare din punct de vedere al temei și modului de abordare. În finalul capitolului, se analizează specificațiile referitoare la caracteristicile sistemului. Îndeplinirea acestor specificații se va analiza în capitolele următoare, prin modalitatea de proiectarea și implementare a aplicației. Proiectarea aplicației va cuprinde o

descriere a modulelor generale ale sistemului și a interacțiunilor dintre acestea, atât în mod abstract, cât și în mod concret, prin analiza arhitecturii din punct de vedere al claselor proiectate împreună cu interacțiunile dintre acestea, al entităților relaționale definite la nivelul sistemului de baze de date. Totodată, se analizează tehnologia aleasă pentru implementare, împreună cu avantaje și dezavantaje ale metodei alese și limite ale sistemului. Implementarea aplicației va analiza funcționarea sistemului, modul de implementare a funcționalităților din punct de vedere al specificațiilor tehnologiilor folosite și a interacțiunii dintre componente. Mai mult decât atât, este abordată și interfața cu utilizatorul. După proiectarea și implementarea sistemului, se realizează testarea acestuia. În cadrul capitolului referitor la testarea aplicației sunt prezentate rezultate experimentale și sunt abordați anumiți parametri de performanță, fiabilitate și securitate.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică pentru tema propusă

Încă de la apariția sistemelor de calcul moderne, una dintre cele mai importante caracteristici ale acestora este reprezentată de capacitatea de a prelucra și stoca date. În esență, un sistem de calcul are la bază două componente principale: unitatea de procesare, numită și CPU și memoria, care este de două categorii: de scurtă durată, folosită în cadrul proceselor, pentru stocarea datelor necesare rulării, și memorie de lungă durată, folosită pentru persistarea datelor pe termen lung. Așadar, se poate deduce în mod evident faptul că, principala caracteristică a unui sistem de calcul este de a manevra un volum de date. Odată cu evoluția tehnologică, care a rezultat în creșterea puterii de procesare, mărirea capacității de stocare a memoriilor și dezvoltarea mecanismelor de transport a datelor prin rețea, volumul de date care este prelucrat și stocat de sistemele de calcul a crescut exponențial (vezi Figura 1.1). Conform statisticilor realizate de organizația Statista, volumul de date creat, transportat și consumat, a crescut de la 2 zettabytes[1] în 2010, până la 79 zettabytes în prezent, urmând să ajungă la 181 zettabytes până în anul 2025, conform predicțiilor [1]. Conform aceleaiași surse, cantitatea de memorie de stocare disponibilă în anul 2020 este de 6.7 zettabytes.

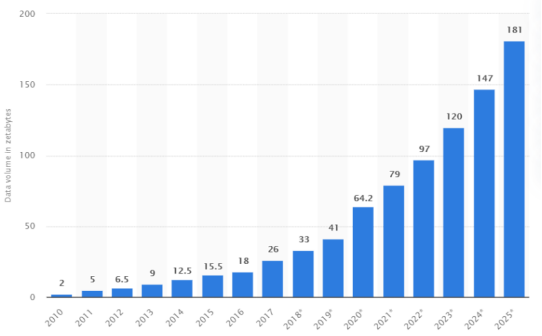

Figura 1.1: Evoluția volumului de date prelucrat de sistemele de calcul între 2010 și 2025

1.1. Domeniul și contextul abordării temei

În cadrul sistemelor de calcul, aceste date pot lua diverse forme, din punct de vedere al perspectivei utilizatorului. Se poate discuta despre conținut text, numere, fotografii, audio, video sau orice alt format care ar putea oferi o prezentare inteligibilă din punct de vedere al percepției umane. Din punct de vedere a formei pe care o poate percepe o memorie digitală, toate datele se reprezintă sub formă binară (valori de 0 și 1), cea mai mică formă de reprezentare cunoscută fiind bitul. Așadar, un fișier de date reprezintă un flux de biți organizați pe baza unei codificări specifice formatului fișierului, stabilit la nivelul sistemului de operare. Tot acest flux de biți este distribuit în mod continuu în zona de memorie, sistemul de operare având rolul de a pune la dispoziție un sistem de fișiere, în care aceste date să fie organizate și disponibile utilizatorilor în formatul specific. Așadar, din perspectiva utilizatorilor, odată amplasate într-o zonă a sistemului de fișiere, datele trebuie să fie disponibile la accesări ulterioare, fără a suferi vreo modificare nedorită. Însă, de tot acest proces al persistenței și al asigurării integrității datelor, trebuie să se ocupe sistemul de operare.

1 1 zettabyte = 1 000 000 000 terabytes

Pentru a asigura persistența datelor în timp în mod eficient, sistemele de stocare trebuie să îndeplinească o serie de principii, care adresează integritatea, disponibilitatea și manevrabilitatea datelor.

- *Mutabilitatea* este principiul conform căruia datele stocate de către utilizator pot fi modificate în orice moment de timp, fără a altera conținutul datelor.
- *Accesibilitatea* este principiul conform căruia datele stocate sunt disponibile utilizatorilor în orice moment de timp, indiferent de frecvența cu care sunt accesate.
- *Integritatea* este principiul conform căruia datele sunt valide și coerente pe întreaga durată de viață a fișierului, fiind invariabile la apariția altor evenimente (adăugarea, modificarea, eliminarea unor date, sau anumite erori) la nivelul altor entități din cadrul sistemului.

Cu toate că aceste principii sunt îndeplinite la nivelul sistemului de operare, aceste principii sunt sensibile la modificări hardware (fizice) are memoriilor, astfel că datele pot fi foarte ușor alterate sau pierdute dacă mediul fizic în care rulează sistemul este sensibil sau memoriile s-au degradat în timp. Conform unui studiu realizat de Google [2], unul dintre factorii fizici care influențează starea fizică a unei memorii este temperatura. S-a constatat faptul că hard disk-urile sunt mult mai sensibile la temperaturi scăzute, decât la temperaturi ridicate, însă și cele foarte ridicate pot afecta semnificativ starea memoriilor. O altă caracteristică ce ar putea influența toleranța la defecte este reprezentată de tipul de memorie și specificațiile hardware. În alt studiu [3], se prezintă faptul că hard disk-urile, care stochează datele pe o suprafață magnetică, au o probabilitate de a se deteriora de 7.01 % pe parcursul întregii durate de viață, mai puțin decât în cazul alternativei SSD[2], care stochează datele în circuite integrate, având o probabilitate de deteriorare de 11.29%.

Un alt considerent semnificativ de luat în calcul pentru construirea ipotezei, considerent care poate îndeplini principiile esențiale ale unui sistem de fișiere, este reprezentat de conceptul de *sistem distribuit,* un sistem care are la bază o rețea de sisteme de calcul independente care comunică și își coordonează activitățile în rețea prin mesaje dar care, din perspectiva utilizatorului, apare ca un singur sistem coerent [4]. Sistemele distribuite au dobândit tot mai multă popularitate în ultimii ani, odată cu creșterea puterii de procesare și a vitezei de transmitere a datelor în rețea, astfel că, volume foarte mari de date se pot procesa foarte rapid folosind calculul distribuit. Fiind alcătuit din mai multe noduri de calcul independente, cu unitate de procesare și memorie proprie, starea globală a sistemului sau a nodurilor individuale nu este afectată de starea unui anumit nod, astfel că, un defect la unul dintre noduri nu implică defectarea întregului sistem. Astfel, un sistem distribuit este un candidat foarte bun pentru a construi un sistem de fișiere, întrucât poate rezolva foarte eficient problema toleranței la defecte fizice a unităților de stocare, prin faptul că, dacă un nod al sistemului eșuează și datele sunt deteriorate sau pierdute, putem prelua datele de pe alt nod al sistemului și refacem starea sistemului.

Pe lângă aceasta, sistemele distribuite sunt independente de platformă. Componentele individuale ale sistemului pot rula pe mașini gazdă cu sisteme de operare diferite (Linux, Windows, MacOs), având sisteme de fișiere diferite, modul de gestiune al datelor fiind transparent pentru sistemul distribuit. Totodată, tehnologiile în care sunt implementate sistemele de gestiune la nivelul nodurilor pot fi diferite întrucât, în cadrul sistemelor distribuite, comunicarea se poate realiza într-un format standardizat, independent de platformă (JSON, XML, etc).

2 Solid State Drive: unități de stocare ce utilizează ansambluri de circuite integrate pentru a stoca datele.

Una dintre caracteristicile datelor la nivelul sistemului de fișiere este mutabilitatea. Utilizatorul are posibilitatea de a modifica fișierele proprii pe parcursul timpului. Astfel, apare nevoia unei modalități de versionare a fișierelor, astfel încât utilizatorul să poată vizualiza toate modificările suferite de fișierele proprii. Aceste versiuni pot fi folosite în cadrul sistemului distribuit pentru a asigura actualitatea datelor.

Pe de altă parte, o altă caracteristică importantă este nevoia utilizatorilor de a accesa aceleași date de pe mai multe dispozitive, din diverse locații. În acest sens, folosirea unui singur sistem pentru stocarea și accesarea datelor reprezintă o soluție ineficientă. Însă, această nevoie poate fi foarte ușor satisfăcută prin intermediul unui sistem distribuit care, pe lângă toată logica necesară stocării datelor, oferă și o interfață pentru accesarea acestora. Această interfață poate fi adaptată pentru mai multe platforme, atât desktop, cât și mobile, pentru mai multe tipuri de aplicații (web, aplicație desktop). Din perspectiva utilizatorului, este pusă la dispoziție o interfață, de unde acesta poate accesa și prelucra fișierele proprii, având garanția integrității, disponibilității și mutabilității datelor, fără a ține cont de modul în care sunt stocate, aranjate, distribuite și prelucrate fișierele.

1.2. Tema propusă și justificarea abordării

Tema propusă este reprezentată de un sistem distribuit de stocare și versionare de fișiere, cu o arhitectură de tip Master-Slave. Sistemul este coordonat de un nod manager general, având în subordine o serie de noduri interne, care vor avea rolul de a stoca datele utilizatorilor. Pentru a asigura disponibilitatea datelor în orice moment, fiecare fișier va fi replicat pe mai multe noduri interne, într-un număr de replici care depinde de tipul utilizatorului. Având în vedere tipul arhitecturii, toată activitatea nodurilor interne va fi coordonată de către nodul general. Astfel, la apariția unui nou fișier în sistem, nodul general va decide care sunt nodurile interne care vor putea stoca acel fișier, pe baza statusului fiecărui nod din sistem, alegând nodurile cu cea mai multă cantitate de stocare disponibilă. Nu va fi sarcina acestuia să trimită fișierul către fiecare nod, ci va crea un token care va conține adresele nodurilor interne care vor stoca fișierul. Acest token va avea forma: 127.0.0.2 – 127.0.0.3 – 127.0.0.200.

Nodul general va trimite acest token către clientul care a solicitat stocarea unui nou fișier, iar clientul va trimite fișierul către primul nod din lanțul conținut în token. La primirea solicitării de stocare a unui nou fișier, nodul intern va analiza token-ul și se va pregăti de stocarea fișierului. Pentru a eficientiza procesul și a nu aglomera clientul, în loc să trimită clientul fișierul către fiecare nod intern, profităm de avantajul sistemelor distribuite, de a asigura comunicare între toate nodurile sistemului, și trimitem fișierul în lanț între nodurile interne. Astfel, la primirea unui fișier, nodul intern va extrage din token propria adresă și, dacă nu reprezintă ultima adresă din lanț, va trimite mai departe fișierul către următorul nod (vezi Figura 1.2).

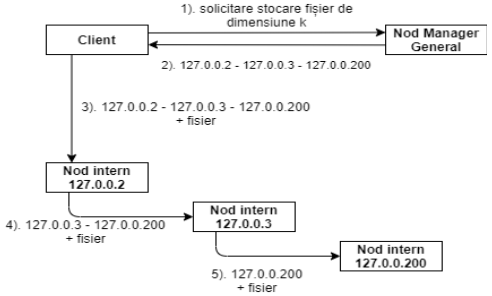

Figura 1.2: Procesul pentru încărcarea unui nou fișier

Una dintre proprietățile unui sistem de fișiere este mutabilitatea. Utilizatorul are posibilitatea să modifice sau să elimine fișierele proprii. Pe lângă adăugarea unui fișier nou, sunt disponibile următoarele operații:

- *Eliminarea* unui fișier. Utilizatorul are posibilitatea de a solicita eliminarea unui fișier. Nodul general va identifica nodurile care stochează fișierul respectiv și le va trimite o Cerere de eliminare a fișierului. (Figura 1.3)
- *Redenumirea* unui fișier. Utilizatorul are posibilitatea de a solicita redenumirea unui fișier. Nodul general va identifica nodurile care stochează fișierul și le va trimite o cerere de redenumire a fișierului. (Figura 1.3)
- *Modificarea conținutului unui fișier.* Din perspectiva nodului general, această operație este echivalentă cu adăugarea unui nou fișier. Procedeul diferă prin faptul că, în urma solicitării făcute de către client, nodul general nu va căuta noduri noi candidat ce ar putea stoca fișierul, ci va identifica nodurile care stochează versiunea anterioară a acestui fișier. Se va returna token-ul către client, urmând ca acesta, prin același procedeu, să trimită datele în lanț către nodurile interne. La nivelul nodurilor interne, această operație va fi echivalentă cu suprascrierea versiunii anterioare a fișierului.

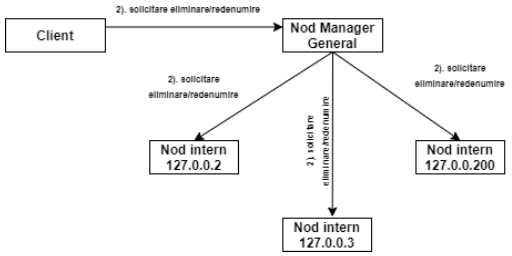

Figura 1.3: Procesul pentru eliminarea/redenumirea unui fișier

O altă caracteristică a sistemului este cea de versionare. Utilizatorul va putea vizualiza detalii referitoare la toate versiunile unui fișier. În cazul ultimelor două operații (redenumire și actualizare), după finalizarea cu succes a procesului, se va înregistra nouă versiune a fișierului. Pentru fiecare fișier din sistem, se va crea un fișier cu metadate. Acesta va conține detalii referitoare la data adăugării sau actualizării fișierului, numărul versiunii, o scurtă descriere a evenimentului, descriere realizată de către utilizator și o sumă de control corespunzătoare fișierului. La calcularea acestei sume de control, se ține cont de o modalitate de identificare a stării fișierului (întregului conținut) la un moment dat. În acest sens, se folosesc codurile redundante ciclice, calculate pe întregul conținut al fișierului, rezultând o valoarea unică, ce va fi corespunzătoare fiecărei versiuni. Acest mecanism este eficient doar în cazul modificărilor apărute la nivelul conținutului fișierului, ceea ce nu este valabil și în situația redenumirii fișierului. Se are în vedere faptul că numele fișierelor nu fac parte din conținutul fișierului, fiind doar identificatori la nivelul sistemului de fișier, pentru regăsirea fișierului. Astfel, modalitatea de identificare în mod unic a unei versiuni a unui fișier se va face prin perechea [număr de versiune, sumă de control].

Toate aceste detalii referitoare la fișierele prezente în sistem trebuie înregistrare de către nodul general. La instanțierea nodului general se va crea o tabelă de conținut care va cuprinde toate fișierele prezente în sistem. Pe lângă detalii de bază ce descriu fișierul (nume, dimensiune, proprietar), sunt prezente și detalii referitoare la numărul de replici existente și ultima versiune a fișierului (număr de versiune și sumă de control). În urma fiecărei actualizări a stării sistemului, se va modifica această tabelă cu noua versiune a unui anumit fișier. Această tabelă are un rol

foarte important, fiind singurul mod prin care nodul general poate identifica fișierele utilizatorilor și starea acestora.

Unul dintre avantajele unui sistem distribuit și a unei astfel de arhitecturi de tip master-slave este faptul că toate nodurile din sistem pot comunica între ele, prin intermediul unei comunicații *multicast*, o comunicare de tip grup în care datele sunt transmise de la un nod către un grup de noduri din aceeași rețea. Pentru a putea comunica prin multicast, nodurile trebuie să deschidă un socket specific pe această adresă și să adere la grupul de multicast. În cazul acestui sistem, această modalitate de comunicare este folosită pentru a asigura comunicarea dintre nodul manager general și nodurile interne, în ceea ce privește monitorizarea activității și verificarea integrității. Pentru a trimite mesaje prin multicast, atât nodul manager general, cât și nodurile interne trebuie să adere la grupul de multicast, însă, aceasta presupune și recepția tuturor mesajelor trimise în grup. Având în vedere că un nod intern nu are putere decizională asupra celorlalte noduri interne, nu este nevoie ca acesta să interpreteze mesajele primite prin multicast de la celelalte noduri interne, ci doar mesajele de la nodul general. Acesta, fiind singurul cu putere decizională, trebuie să primească și să interpreteze mesaje de la toate nodurile interne.

În acest sens, fiecare nod intern va trimite un *heartbeat*[3] la intervale regulate de timp, încă de la începutul activității. Acestea vor fi recepționate de către nodul general. Astfel, prima caracteristică a acestei abordări este de *service discovery*[4]. Nodul general va putea identifica și înregistra apariția unui nou nod în sistem. În aceeași măsură, vor fi identificate nodurile care sunt deconectate din sistem. A doua caracteristică esențială este de monitorizare a activității nodurilor. Nodul general trebuie să cunoască în permanență statusul nodurilor interne, din punct de vedere al disponibilității, al fișierelor stocate și al integrității acestora. Nodurile interne vor include în fiecare mesaj trimis, statusul stocării. Se vor trimite datele referitoare la toate fișierele înregistrate în sistem. Aceste date fac referire la utilizatorul care deține fișierul, numele fișierului, dimensiunea și versiunea actuală a fișierului. Pe baza acestor caracteristici, nodul manager general va construi și actualiza o tabelă de status. În această tabelă vor fi centralizate toate fișierele existente în sistem, împreună cu nodurile pe care sunt stocate. Totodată, pe măsură ce fișierele sunt vehiculate prin sistem, nodul general va alcătui o altă tabelă de conținut.

Cu ajutorul acestui mecanism, în cazul în care nodul general detectează un nod care s-a deconectat din sistem (prin faptul că nu mai trimite mesaje de tip heartbeat), acesta va iniția procesul de replicare astfel încât, pentru un fișier, numărul de replici existente în sistem să fie mereu același. În acest sens, nodul general va identifica noduri candidat care vor putea stoca fișierele nodului oprit. Se vor identifica nodurile cu cea mai multă memorie disponibilă și se vor trimite fișierele către acestea. În aceeași măsură, se dorește evitarea supra replicării. Astfel, dacă un nod care a fost oprit va reporni, nodul general va iniția procesul de eliminare a fișierelor supra replicate de la anumite noduri. Identificarea nepotrivirilor la nivelul sistemului se va face prin compararea tabelei de status cu tabela de conținut. Astfel, la nivelul sistemului va exista un mecanism de verificare a statusului sistemului. La anumite intervale regulate de timp se vor efectua următoarele comparații dintre cele două tabele :

- verificarea factorului de replicare. Dacă factorul de replicare corespunzător unui fișier din tabela de conținut nu este egal cu numărul de noduri care stochează un fișier din tabela de status, se va genera comanda aferentă, de replicare sau eliminare.
- verificarea versiunii. Se impune ca numărul de versiune corespunzător fiecărui fișier din tabela de conținut să fie același cu numărul de versiune al fișierului corespunzătoare fiecărui nod care stochează un fișier, din tabela de status.

3 Heartbeat: mesaj trimis la intervale regulate de timp pentru a indica funcționarea normală și a realiza sincronizarea cu sistemul.

4 Service discovery : procesul de detectare automată a dispozitivelor și serviciilor dintr-o rețea. [5]

- verificarea sumei de control. Prin intermediul acesteia se vor putea identifica coruperi ale fișierelor la nivelul anumitor noduri. Dacă suma de control a unui fișier din tabela de conținut nu corespunde cu suma de control a replicii corespunzătoare fiecărui nod, înseamnă că a apărut o problemă la nodul intern. În această situație, se va iniția o comandă de eliminare a fișierului de la nodul respectiv. Va apărea nevoia de replicare a fișierului (neîndeplinirea primei cerințe) și fișierul va fi replicat pe alt nod.

În verificarea integrității fișierelor se folosesc sumele de control bazate pe codurile de redundanță ciclică. Procedeul aplicării acestor sume de control implică parcurgerea conținutului întregului fișier ceea ce, în cazul unor fișiere de dimensiuni mari, presupune un timp relativ mare de calcul (vezi Figura 1.4). Se poate observa că timpul necesar calculării sumei de control crește exponențial, direct proporțional cu dimensiunea fișierului.

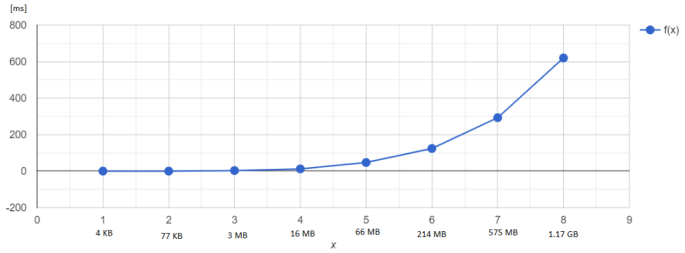

Figura 1.4: Graficului raportului dintre dimensiunea fișierului și timpul necesar calculării CRC32

Astfel, în situația în care sistemul ar deveni foarte complex din punct de vedere al cantității de date stocate la fiecare nod, timpul necesar calculării sumei de control ar fi prea mare. Având în vedere că frecvența de trimitere a mesajelor de tip heartbeat este mult mai mică, nu putem calcula suma de control la fiecare ciclu. Din acest motiv, la intervale regulate de timp, mult mai mari decât frecvența de trimitere a heartbeat-urilor, nodul general solicită nodurilor interne să calculeze și să furnizeze suma de control pentru a verifica integritatea datelor. Această cerere se realizează prin multicast. Nodul general trimite un mesaj pe adresa de multicast, mesaj care va fi recepționat de toate nodurile interne. Răspunsul nu va putea fi pregătit până la următorul ciclu de trimitere, așa că nodul intern va începe calcularea sumei de control și o va furniza nodului general imediat ce va termina calculul.

O altă caracteristică importantă a sistemelor distribuite este *scalarea*. Un sistem distribuit trebuie să fie scalabil, astfel încât să poată satisface nevoia de prelucrare a mai multor cereri și date, pe măsură ce sistemul devine mai complex. Arhitectura prezentată, de tip master-slave facilitează scalarea pe *orizontală*[5] *și verticală*[6]. Scalarea pe orizontală se realizează prin faptul că se pot introduce mai multe noduri interne care să stocheze fișierele. Se evită astfel aglomerarea nodurilor și se poate îmbunătăți disponibilitatea, prin creșterea numărului de replici ale fiecărui fișier. Scalarea pe verticală se realizează prin extinderea unităților de stocare disponibile la nivelul fiecărui nod, atât prin creșterea capacității de stocare, cât și prin adăugarea unor memorii mai rapide din punct de vedere a timpul de acces (scriere/citire).

Din punct de vedere al stării, nodul general are comportament cu stare, din cauza faptului că salvează datele între cereri consecutive ale clienților și oferă rezultate pentru cereri în funcție de aceste date salvate. Datele salvate fac referire la cele două tabele care înglobează statusul fișierelor și al nodurilor interne. Din acest motiv, eventuale erori la nivelul managerului general ar induce căderea întregului sistem. Însă, este inclusă o componentă de refacerea stării, pe baza heartbeat-urilor primite de la nodurile interne. În schimb, nodurile interne sunt fără stare. Acestea

5 Scalarea pe orizontală presupune extinderea sistemului prin adăugarea mai multor noduri de calcul.
6 Scalarea pe verticală presupune extinderea sistemului prin adăugarea de resurse (hardware) nodurilor de calcul existente.

nu salvează date referitoare la fișierele existente în sistem. La trimiterea fiecărui beat, se citește sistemul de fișiere și se identifică tot conținutul necesar (conținutul fișierelor și metadate). Însă, prelucrările efectuate de nodurile interne nu presupun salvarea anumitor date care să influențeze cererile clienților sau ale nodului general.

Sistemul distribuit prezentat îndeplinește principiile unui sistem de fișiere, prin faptul că asigură *mutabilitatea,* oferind posibilitatea modificării fișierelor, asigură *disponibilitatea*, prin mecanismul de replicare, ce garantează faptul că datele utilizatorilor vor fi mereu disponibile în sistem. Totodată, se asigură *integritatea*, prin verificarea frecventă a integrității fișierelor cu ajutorul sumei de control și a numărului de versiune.

1.3. Analiza aplicațiilor existente din aceeași categorie

Odată cu creșterea volumului de informație generată și consumată de utilizatori, creștere alimentată și de dezvoltarea canalelor de comunicație, a puterii de procesare și a dispozitivelor mobile, nevoia de a avea o unitate de stocare accesibilă prin intermediul rețelei a crescut semnificativ. Una dintre cele mai mari platforme online de stocare de date numită Google Drive a declarat în cadrul conferinței Google Cloud Next din San Francisco în 2018, că numărul total de fișiere stocate pe serverele platformei este de 2 trilioane de fișiere iar numărul de utilizatori activi este de 800 milioane [6]. În aceeași măsură, pe site-ul oficial, Dropbox declară peste 700 de milioane de utilizatori în 2021 [7]. Cu toate că sistemele de stocare a fișierelor sunt generale, în sensul că pot fi folosite în cadrul unui spectru foarte larg de aplicații, implementările multor sisteme distribuite de stocare de fișiere sunt optimizate pentru o clasă particulară de aplicații.

1.3.1. Andrew File System (AFS)

Sistemul Andrew File System (AFS), un sistem construit ca o extensie a nucleului sistemului de operare și optimizat pentru a stoca fișierele utilizatorilor folosind tehnici de caching [8]. Acest sistem a fost proiectat în jurul anilor 1980 de către Universitatea Carnegie-Mellon, având drept scop scalarea și servirea unui număr mare de clienți. Unul din principiile de bază ale acestui sistem, disponibil în cadrul tuturor versiunilor, este procesul „whole-file caching”, realizat pe unitatea de stocare locală a clientului care dorește să acceseze fișierul. La prima accesare a fișierului, acesta era adus de pe server, fiind stocat în memoria locală a clientului. Următoarele operații de citire și scriere a fișierului nu se trimiteau către server, ci se executau asupra fișierului local. Astfel, aceste operații nu necesitau comunicarea cu serverul prin intermediul rețelei, fiind mai rapide și mai sigure. Însă, versiunile ulterioare asigurau trimiterea unei cereri de la client către server, prin care solicitau eventuale modificări ale fișierelor efectuate la nivelul serverului, astfel încât să actualizeze cea mai recentă versiune. Ulterior, versiunile finale încercau eliminarea numărului mare de cereri dintre client și server, prin adăugarea unor funcții de tip callback la nivelul serverului, prin care era informat clientul la modificarea fiecărui fișier care a fost accesat. La finalul procesului, reprezentat de închiderea fișierului, dacă acesta a fost modificat, este trimis către server unde va fi persistat. Astfel, prin minimizarea numărului de interacțiuni dintre client și server, se asigura faptul că serverul poate trata un număr mare de clienți [9].

1.3.2. Hadoop File System

Un alt sistem distribuit de stocare, optimizat pentru a oferi un strat de persistență pentru

framework-ul MapReduce[7] este sistemul Hadoop, cunoscut drept HDFS (Hadoop File System) [8] Principalele aspecte considerate în proiectarea acestui sistem au fost toleranța la defecte și posibilitatea de a rula pe dispozitive hardware low-cost. Acest sistem adresează oferirea unui timp de răspuns foarte bun pentru aplicațiile care lucrează cu un volum foarte mare de date (fișiere cu dimensiuni exprimate în gigabytes și terabytes). Sistemul are o arhitectură de tip master/slave, organizat sub formă de clustere (vezi Figura 1.5 [10]). Un cluster conține un singur server master, numit *NameNode*, care coordonează întregul sistem de fișiere, prin gestionarea depozitului de metadate și prin comunicarea cu nodurile slave, numite DataNode, care gestionează unitatea de stocare locală, unde sunt stocate fișierele. La nivelul unui nod, fișierele nu sunt stocate sub forma unui singur bloc continuu de memorie, ci sunt împărțite în mai multe blocuri de dimensiune fixă, distribuite în memorie sub formă de seturi. Aceste noduri interne sunt responsabile atât pentru crearea, ștergerea și replicarea blocurilor ce compun un fișier, cât și pentru pentru servirea clienților, în cadrul operațiilor de citire și scriere. În mod general, în cadrul unui cluster, într-un mediu real de rulare, doar nodul master rulează pe o mașină dedicată. Nodurile interne pot rula pe aceleași mașini, în diferite instanțe software. Nodurile folosesc tehnologia Java pentru a asigura posibilitatea rulării pe o varietate de tipuri de mașini [11].

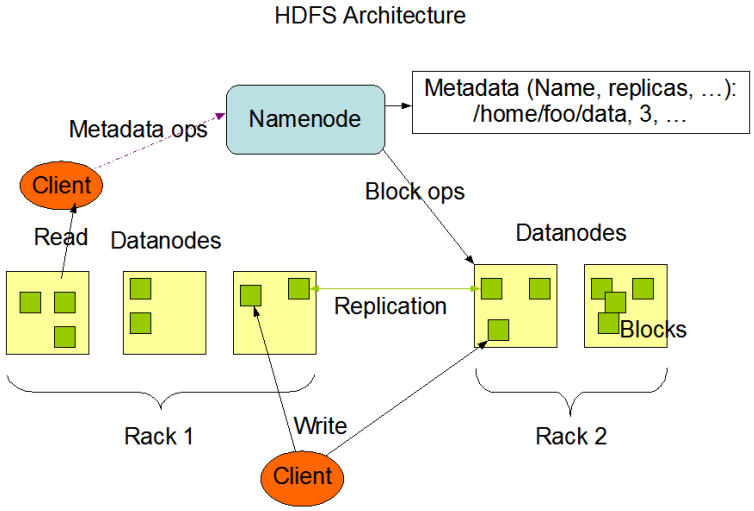

Figura 1.5: Arhitectura HDFS , preluată de la hadoop.apache.org

1.3.3. Google File System (GFS)

Un alt sistem distribuit de stocare de fișiere, mult mai cunoscut datorită organizației dezvoltatoare, este *Google File System* (GFS), folosit în cadrul uneia dintre cele mai mari platforme pentru stocare de fișiere, numită *Google Drive*. Sistemul este cunoscut datorită volumului mare de date stocate și a numărului mare de utilizatori. Pentru a putea gestiona o încărcare atât de mare, sistemul este alcătuit din clustere, de dimensiuni măsurate în sute de terabytes, compuse din peste 1 000 noduri, accesate de sute de clienți. Sistemul are o arhitectură de tip master/slave, organizată sub formă de clustere. Fiecare grup are un singur nod master și mai multe *chunkservers* ce sunt accesate prin intermediul unor noduri client. (vezi Figura 1.6).

7 MapReduce este o paradigmă de programare folosită în contextul procesării în mod paralel și distribuit a unui volum mare de date.

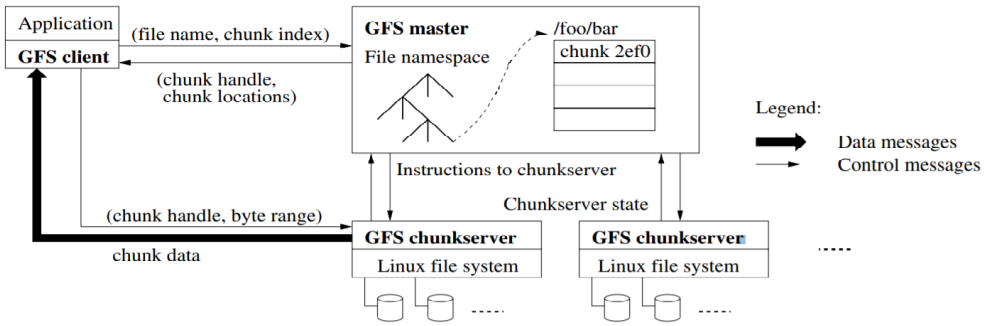

Figura 1.6: Arhitectura GFS

Nodurile client și *chunkserver* pot rula pe aceeași mașină, în timp ce nodul master va rula individual pe o mașină. Starea sistemului este menținută prin intermediul comunicației multicast, folosind mesaje de tip *heartbeat*. La nivelul fiecărui nod intern, fiecare fișier va fi stocat sub formă de blocuri de dimensiune fixă (pachete de 64 MB) care, la nivelul sistemului de operare, vor fi identificate ca fișiere independente. Pentru asigurarea disponibilității, fiecare bloc va fi replicat pe mai multe noduri *chunkserver*, cu un anumit factor de replicare. (implicit, valoarea factorului de replicare este 3). Astfel, pe baza mesajelor periodice vehiculate prin multicast, nodul master va colecta datele despre starea fiecărui nod intern și va alcătui metadatele, care vor fi stocate în memoria locală a nodului master. Pe lângă operațiile esențiale necesare pentru menținerea integrității sistemului (replicare, eliminare), sunt disponibile și operații pentru a reduce încărcarea nodurilor interne. Se folosesc mecanisme de *garbage collecting*, prin care sunt eliminate blocurile care nu mai fac parte din componența unui fișier.

Cu toate că un astfel de sistem atinge o încărcare foarte mare și timpul de acces la date trebuie să fie foarte mic, acest sistem nu folosește tehnici directe de *caching* la nivelul stocării datelor, din cauza dimensiunii mari a fișierelor și, implicit, a numărului de blocuri, pentru care, gestiunea cache-ului ar fi mult prea costisitoare, fără a aduce o îmbunătățire considerabilă a performanțelor. Cu toate acestea, se folosesc tehnici indirecte de caching, având în vedere că blocurile ce alcătuiesc un fișier sunt gestionate de către Linux ca fiind fișiere de sine stătătoare iar Linux salvează în buffer fișierele accesate cel mai frecvent [12]. Cu toate aceste, se poate observa în Figura 1.7, prin analiza comparativă a două clustere, că performanțele sistemului sunt foarte bune.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Figura 1.7: Metrici de performanță a două clustere din cadrul GFS

1.3.4. Ceph File System

În cadrul sistemelor prezentate, cu o arhitectură de tip master/slave, nodul master gestionează starea sistemului prin intermediul metadatelor. Aceste sisteme persistă metadatele în memoria locală, ceea ce induce o limitare a performanțelor atunci când încărcarea sistemului

crește. O alternativă la această abordare este reprezentată de sistemul *Ceph*. Acest sistem este proiectat astfel încât să suporte stocarea de fișiere, asigurând performanța, scalabilitatea și fiabilitatea, prin eliminarea elementului *single point of failure*. Sistemul de stocare de fișiere disponibil în cadrul sistemului expune o interfață POSIX[8] numită CephFS, folosind un driver nativ Linux dezvoltat de Linus Torvalds, integrat în versiunea 2.6.34 a nucleului de Linux [14].

Spre deosebire de sistemele prezentate, sistemul Ceph maximizează separarea dintre date și metadate, prin distribuirea dinamică a metadatelor în cadrul unui cluster, pe unul sau mai multe noduri [15]. Această abordare este mai complexă, întrucât toate nodurile sistemului care conțin metadatele sunt strâns cuplate. Toate nodurile trebuie să se sincronizeze pentru a stabili o singură stare a sistemului de fișiere. Un alt element distinctiv este modul în care se realizează maparea dintre un fișier și blocurile care îl compun. Acest sistem folosește algoritmul *CRUSH* (Controller Replication Under Scalable Hashing). Algoritmul primește ca intrare un fișier și, pe baza unei tabele de mapare ierarhică a clusterului, realizată de către administratorul grupului, generează o listă de noduri pe care vor fi distribuite blocurile ce compun fișierul. Această tabelă de mapare reprezintă elementul cheie a acestui algoritm întrucât o definire ineficientă sau eronată a acesteia poate îngreuna considerabil procesul de identificare a blocurilor fișierului. Tabela conține evidența resurselor de stocare, atât din punct de vedere al memoriei disponibile, cât și din punct de vedere al elementelor ce compun resursele și a modului de distribuire și organizare a acestora. Drept exemplu, se poate descrie o rețea de resurse organizate sub forma unei instalații mari, pe linii cu servere, la care sunt conectate rafturi întregi cu unități de stocare. Politica de distribuire a datelor este definită în termenii unor reguli de amplasare care specifică factorul de replicare ales pentru cluster și care sunt restricțiile impuse în procesul de replicare. De exemplu, datele pot fi replicate atât în cadrul aceluiași server, pe unități de stocare diferite, dar și în cadrul a servere diferite, care să nu împărtășească aceeași unitate de comandă. Astfel, ținând cont de această tabelă de mapare a resurselor și de modul de amplasare a datelor, se va genera o listă de noduri care vor putea stoca fișierul utilizatorului. Această listă va fi organizată sub forma unui arbore, având ca noduri intermediare serverele, iar ca frunze, unitățile de stocare din cadrul serverului, pe care se vor stoca datele. Organizarea sub forma unui arbore (Figura 1.8) eficientizează procesul de identificare a nodurilor [16].

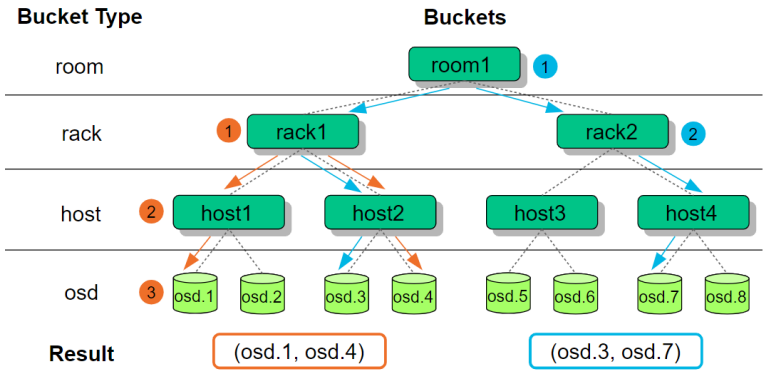

Figura 1.8: Organizarea ierarhică a resurselor de stocare și modalitatea de identificare a replicilor unui fișier

Acest algoritm a fost proiectat astfel încât să optimizeze distribuirea datelor în cadrul resurselor disponibile și să realizeze o reorganizare eficientă a datelor atunci când resursele de calcul sunt adăugate sau eliminate.

8 Portable Operating System Interface : familie de standarde specificate de IEEE pentru menținerea compatibilității dintre sistemele de operare [13].

Subiectul stocării distribuite a datelor este un aspect foarte relevant, având în vedere creșterea volumului de date, pentru care se impune și o adaptare a timpului de acces la date. Majoritatea sistemelor de stocare au ca arhitectură de bază master/slave și presupun distribuirea datelor pe nodurile slave, la coordonarea nodului master. Aceste sisteme diferă prin modul în care sunt distribuite datele (se stochează fișierul întreg pe un nod sau se partiționează în blocuri de dimensiune fixă, distribuite pe aceste noduri; se folosesc algoritmi specifici pentru maparea datelor, care să țină cont și de resursele de calcul sau de modul de organizare a acestora, sau se distribuie datele în mod aleator) și modul în care sunt organizate metadatele (în memoria internă a nodului master, astfel încât să crească viteza de acces, sau în mod distribuit, pe mai multe noduri, în cadrul unui server). Diferența dintre sisteme provine, în primul rând, din scopul pentru care a fost dezvoltat sistemul. Pot exista sisteme implementate pentru utilizare generală, pentru stocarea de fișiere. Aceste sisteme sunt orientate și pe interacțiunea dintre utilizatori, punând la dispoziție mijloace de partajare a datelor între utilizatori (*Dropbox* și *Google Drive*). Alte sisteme sunt orientate pe efectuarea de procese complexe. Un astfel de sistem este *Hadoop*, care oferă un nivel de persistență a datelor pentru algoritmul MapReduce. Aceste sisteme sunt scalabile și fac față volumului de mare de date stocate, oferind un timp de acces foarte bun.

1.4. Specificații privind caracteristicile așteptate

Sistemul distribuit de stocare de fișiere are ca obiectiv stocarea fișierelor utilizatorilor într-o manieră care să asigure disponibilitatea și integritatea datelor. Pentru a asigura disponibilitatea sistemului, trebuie îndeplinite următoarele specificații:

- sistemul trebuie să poată trata cereri în paralel. Din acest motiv, fiecare interacțiune cu un nou client trebuie realizată pe un thread separat.
- fiecare fișier al utilizatorului trebuie stocat în sistem pe nodurile interne, într-un număr de replici determinat de factorul de replicare corespunzător tipului utilizatorului.
- pe parcursul existenței fișierului în cadrul sistemului, nodul general trebuie să asigure menținerea constantă a factorului de replicare corespunzător fiecărui fișier. Nodul general trebuie să detecteze oprirea nodurilor interne și să declanșeze replicarea fișierelor pe alt nod. În aceeași măsură, nodul general trebuie să detecteze supra-replicarea și să elimine o replică de la un anumit nod.

Pentru a asigura integritatea datelor, trebuie îndeplinite următoarele specificații:

- Nodul general trebuie să persiste date referitoare la starea conținutului tuturor replicilor unui fișier. În acest sens, nodul general va interoga periodic nodurile interne pentru a verifica dacă starea conținutului nu s-a modificat. Se vor folosi coduri redundante ciclice. La apariția unei noi versiuni a unui fișier în sistem, suma de control trebuie actualizată la nivelul metadatelor nodului general. La intervale de timp regulate, nodul general trebuie să solicite nodurilor interne care stochează acea versiune, să calculeze suma de control a conținutului fișierului. Această sumă de control trebuie să corespundă cu cea de la nivelul nodului general. În caz contrar, replica respectivă trebuie eliminată, astfel încât să fie generată o altă replică validă.
- În cadrul operațiilor de adăugare/actualizare de fișiere, nu trebuie definite datele de identificare a stării și a noii versiuni a fișierului până nu se garantează salvarea fișierului într-o stare validă la cel puțin un nod intern. Dacă fișierul are cel puțin o replică validă, se pot genera și celelalte replici astfel încât să se atingă factorul de replicare corespunzător. Astfel, la adăugarea unui nou fișier, trebuie asigurate mecanisme de feedback în comunicarea dintre client și nodul intern și între client și nodul general. La primirea unui nou fișier, nodul general așteaptă feedback de la client. În acest timp, clientul trimite

fișierul către nodurile interne, în lanț, așteptând feedback de la toate nodurile, cu o anumită perioadă de expirare. Dacă nodul client primește un feedback valid de la cel puțin un nod intern, va trimite un feedback valid către nodul general, care va înregistra metadatele de identificare, versiune și stare ale noului fișier.

Totodată, fiind un sistem care poate atinge o încărcare foarte mare din cauza volumului mare de date stocate și având în vedere că pot interveni erori la nivelul nodurilor interne, trebuie asigurată eliminarea eventualelor inconsistențe.

- Dacă un nod a fost oprit și între timp un anumit fișier care conținea o replică pe acel nod a fost eliminat, la repornirea nodului trebuie detectată eliminarea fișierului și trebuie generată și eliminarea replicii respective.
- Dacă un nod a fost oprit și între timp un anumit fișier a fost redenumit, la repornire trebuie detectată această redenumire iar replica respectivă trebuie eliminată. Nu se realizează redenumirea replicii, întrucât s-ar genera o supra-replicare, ceea ce ar implica automat eliminarea unei replici, consecințele fiind aceleași.

O altă caracteristică a sistemului este reprezentată de componenta de versionare. Sistemul trebuie să înregistreze fiecare modificare (adăugare, redenumire, actualizare) a unui fișier sub forma unei noi versiuni, ce va fi identificată în mod unic, prin intermediul unui număr de versiune și a sumei de control. Datele despre versiunea unui fișier trebuie persistate, atât la nivelul nodului general în metadatele (tabelele de stare), cât și la nivelul fiecărui nod intern. Pentru fiecare replică a fișierului, trebuie să existe un fișier extern de metadate, care să conțină informații relevante despre versiune (identificarea unică a versiunii, data și descrierea evenimentului). Sistemul trebuie să identifice eventuale nepotriviri ale versiunilor, prin compararea datelor de la nivelul nodului general și nodurilor interne. Aceste nepotriviri trebuie rezolvate, prin eliminarea replicii respective și generarea altei replici valide.

Totodată, sistemul trebuie să asigure mecanisme eficiente de monitorizare pentru administrator. Fiecare eveniment apărut la nivelul nodurilor sistemului trebuie înregistrat printr-o modalitate insensibilă la erori apărute la nivelul nodurilor. Se vor persista date despre evenimentele apărute, într-o bază de date relațională de tip My*SQL*, gestionată de o aplicație externă cu o interfață REST. Aceste date vor sugera în mod evident cine este entitatea la nivelul căruia a apărut evenimentul, ce tip de eveniment este (eroare, atenționare, succes), o scurtă descriere a evenimentului și data apariției. Pentru aceste informații trebuie asigurate mecanisme de adăugare, extragere și eliminare, care să poată fi gestionate doar de către administratorul sistemului.

Pentru a asigura un anumit grad de securitate la nivelul comunicației dintre client și sistem, trebuie folosite mecanisme prin intermediul cărora datele sensibile ale utilizatorilor să fie codificate și semnate digital. Aceste date sensibile trebuie să includă detalii referitoare la identitatea și rolul utilizatorului, necesare în procesul de autentificare și autorizare. Acțiunile care vor fi disponibile pentru utilizator vor fi expuse în funcție de rolul utilizatorului. Astfel, prin folosirea standardului *JWT*, datele sensibile ale utilizatorului ce vor fi atașate cererii, vor fi codificate și semnate astfel încât să nu poată fi interceptate și interpretate de un intermediar nedorit. Acest token trebuie generat la autentificarea utilizatorului, trebuie să conțină identitatea și rolul utilizatorului și trebuie să fie disponibil pe durata unei sesiuni de lungime finită.

Capitolul 2. Proiectarea aplicației

2.1. Platforma hardware

Aplicația are la bază un sistem distribuit. Una dintre caracteristicile importante ale sistemelor distribuite este independența de platformă. Astfel, sisteme distribuite nu impun restricții în ceea ce privește platformele hardware pe care sunt găzduite nodurile rețelei. Principala trăsătură hardware a platformelor ce găzduiesc nodurile rețelei este posibilitatea de a comunica în rețea. Nodurile sistemului trebuie să poată vehicula mesaje în rețea, astfel încât să poată comunica cu celelalte noduri, care se pot afla pe stații diferite.

O caracteristică specifică aplicației, necesară pentru platforma hardware, este reprezentată de cantitatea de memorie disponibilă. Având ca obiect stocarea unui volum mare de date, se impune ca nodurile interne să aibă o cantitate mare de memorie de stocare disponibilă. Pentru nodul general nu se impune o astfel de restricție, având în vedere că acesta nu va stoca fișiere. Pe de altă parte, operațiile asupra fișierelor (calcularea sumei de control) trebuie să se efectueze rapid. Din acest motiv, sunt preferate memoriile performante, cu viteze de scriere/citire foarte bune. Principalele tipuri de unități de stocare sunt HDD și SSD. Se cunoaște faptul că latența este mai bună în cazul memoriei de tip *solid state drive* (0.031 ms, față de 15.785 ms, în cazul hard disk-ului), vitezele de scriere/citire sunt mai bune în cazul unui SSD (514.28 MB/s, comparativ cu 149.86 MB/s în cazul unui hard disk) [17]. Din acest punct de vedere, se preferă memoriile de tip solid state drive. Însă, acestea au un preț mai ridicat per TB, față de varianta hard-disk. Din acest motiv, trebuie să se facă un compromis. Viteza memoriei este foarte importantă însă, într-un asemenea sistem, este foarte importantă și cantitatea de stocare. Ca o soluție la această problemă, se pot desemna anumite noduri interne care să nu fie folosite pentru manipularea datelor, ci doar pentru replicare. Aceste noduri ar putea fi interogate mai rar, din punct de vedere al integrității.

2.2. Modulele generale și interacțiunile dintre ele

Sistemul este alcătuit dintr-o serie de module care pot fi grupate în două categorii. În primul rând, avem module specifice, care înglobează activitatea unei componente a sistemului. Aceste module sunt *GeneralManager*, *GeneralNode*, *FrontendProxy*, *RESTApi* și *ui-frontend*. Aceste module pot fi încărcate și rulate ca un proces de sine-stătător. Nu există interdependențe între modulele generale, în ceea ce privește implementarea. A doua categorie de module este reprezentată de modulele generale, care înglobează funcționalități necesare modulelor specifice sau altor module generale. Între aceste module pot exista dependențe, însă, spre deosebire de modulele specifice, nu pot rula ca un proces. Aceste module sunt *Beans*, *HTTPHandler* și *GeneralPurposeStructures* (vezi Figura 2.1).

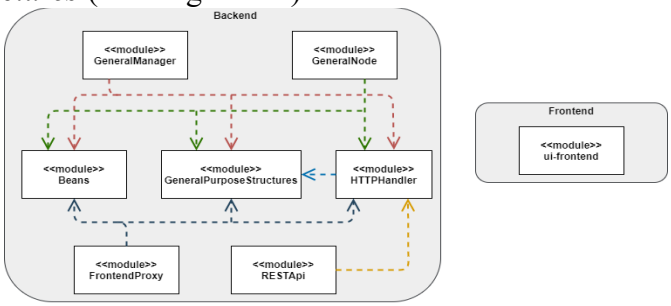

Figura 2.1: Modulele aplicației și interacțiunile dintre ele

Modulul *GeneralPurposeStructures* este un modul general care conține funcționalități sau elemente generale, necesare mai multor module.

- funcții generale de manipulare a diverse structuri de date.
- structuri de date specifice, pentru a caracteriza mai eficient un set de date.
- funcții de configurare a sistemului, care încarcă în aplicație parametrii sistemului, dintr-un fișier de configurare.
- funcții specifice sistemului de fișiere, pentru manipularea fișierelor (scriere, citire, redenumire, identificare de caracteristici, calcularea sumei de control)
- obiecte de tip înveliș pentru a caracteriza mai eficient anumite particularități și a expune anumite metode de validare și manipulare a caracteristicilor obiectului părinte (adresa unui nod de rețea sau socket-ul specific comunicației de tip multicast)
- funcții de conversie a datelor la un format specific comunicației în rețea (metode de serializare și deserializare a obiectelor).

Modulul *Beans* este un modul general și conține clasele ce definesc obiectele care vor fi vehiculate în rețea, între nodurile sistemului. Conform terminologiei specifice Java, acestea se numesc *Java Beans*. Caracteristica principală a acestor clase este faptul că au membri privați, constructor fără argumente, gettere și settere[9] pentru membrii privați [18]. Aceste clase definesc corpul cererilor vehiculate în rețea. De aceea, aceste obiecte trebuie să poată fi transformate într-un flux de octeți la sursă, și reconstruite la recepție. Din acest motiv, clasele care definesc aceste obiecte trebuie să fie serializabile. Aceasta se poate realiza prin moștenirea clasei generice *Serializable*, specifice Java. Un dezavantaj al acestei abordări este faptul că nodurile între care circulă aceste obiecte trebuie să fie implementate în aceeași tehnologie, întrucât serializarea obiectului se realizează conform reprezentărilor interne ale obiectelor specifice limbajului. Un obiect serializat și trimis pe un socket de date dintr-un client dezvoltat în Java, nu va putea fi deserializat și reconstruit la recepție pe un client dezvoltat în Python. Având în vedere acest aspect, nodul general, nodurile interne și intermediarul clientului vor fi dezvoltate în aceeași tehnologie.În sistem sunt prezente trei comunicații principale, determinate ce cele trei componente principale ale sistemului (nodul general, nodurile interne și clientul). Pentru fiecare dintre aceste tipuri de comunicații sunt definite obiecte, atât pentru cerere, cât și pentru răspuns.

Modulul *HTTPHandler* este un modul general care cuprinde funcționalități și obiecte specifice comunicării prin protocolul HTTP. La nivelul sistemului, acest tip de comunicație este prezent între client și server-ul cu interfață de tip REST și între client și intermediarul acestuia (acestea vor fi detaliate în modulele următoare). La nivelul acestui modul sunt prezente următoarele:

- metode prin care se trimit cereri *HTTP*, de tip *GET*, *PUT*, *POST* și *DELETE.*
- metode prin care se realizează înregistrarea activității componentelor sistemului, prin trimiterea unor cereri de salvare a descrierii evenimentelor, într-o bază de date, gestionată de către modulul *RESTApi*.
- componenta *JWT*[10] de securizare a parametrilor vulnerabili ai cererilor HTTP dintre client și componentele sistemului. În cadrul standardului JWT, datele sunt reprezentate în format *JSON* și conțin un set de elemente ce descriu o anumită identitate electronică [19]. De cele mai multe ori, se folosesc pentru a descrie identitatea unui utilizator, în contextul autorizării acestuia. Totodată, acest *token* poate fi semnat electronic. Prin intermediul

9 Getter și Setter sunt nume generice date funcțiilor deaccesare (scriere și citire) a membrilor privați ai unei clase. Fiind privați, aceste funcții sunt singurele modalități prin care se oferă acces direct la membru respectiv.

10 JSON Web Token

semnăturii electronice, se va asigura integritatea și autenticitatea token-ului, din punct de vedere al stării rezultate în urma transferului între parteneri (a fost sau nu corupt sau identifică entitatea sursă) [19]. În cadrul sistemului, standardul JWT se folosește în contextul comunicațiilor dintre client și REST Api și între client și intermediarul acestuia, pentru a securiza informațiile de identificare ale utilizatorilor sistemului și pentru a autoriza identitatea acestuia în contextul operațiilor.

Modulul *RESTApi* este un modul specific, prin care sunt adresate anumite resurse din cadrul sistemului, fiind folosit stilul arhitectural *REST*. Acest stil arhitectural provine de la *REpresentational State Transfer* și reprezintă o abordare de creare a aplicațiilor client-server orientate pe transferul reprezentărilor resurselor în cadrul cererilor și răspunsurilor, prin intermediul HTTP [20]. Fiind orientat pe resursă, comunicațiile dintre client și server reprezintă operațiile ce se pot efectua asupra resurselor. Accesul la acestea se face prin intermediul URI [21]. În cadrul sistemului, acest modul expune reprezentarea a patru resurse ale sistemului. În ceea ce privește interacțiunea cu clientul și identificarea acestuia, este prezentă reprezentarea resursei *User* și *UserTypes*. Pe lângă accesarea datelor utilizatorului, se expune și o funcționalitate de autentificare, în urma căruia se va genera un *token* care va fi returnat clientului, fiind disponibil pe durata unei sesiuni și fiind folosit în cadrul cererilor următoare pentru autorizare. După expirarea token-ului, utilizatorul va trebui să se reautentifice. Astfel, acest modul depinde de modului *HTTPHandler,* care expune toate metodele creării și validării acestui *token*. În ceea ce privește interacțiunea cu nodurile sistemului, se expune reprezentarea resursei *InternalNode* și a resursei *Log*. Prima resursă descrie identitatea și starea nodurilor interne. A doua resursă descrie evenimentele înregistrate în sistem, cu scopul de a fi expuse administratorului sistemului pentru a facilita monitorizarea aplicației.

Un alt aspect important referitor la acest modul este reprezentat de comunicarea cu serverul de baze de date. Toate aceste resurse vor fi persistate într-o bază de date relațională de tip MySQL. De aceea, acest modul expune și funcționalități de transfer de date cu serverul de baze de date. Sunt disponibile operațiile de tip *CRUD*[11] asupra datelor.

Modulul *FrontendProxy* reprezintă un intermediar între nodurile sistemului și aplicația de tip client. Acest modul va conține funcționalități de interpretare a cererilor *HTTP* primite de la client. În urma interpretării cererilor, se vor instanția obiectele *java beans* corespunzătoare, expuse de modulul *Beans* și se vor trimite către nodurile aplicației, în funcție de tipul cererii și de operația solicitată. Comunicațiile au ca scop, atât efectuarea operațiilor de adăugare, eliminare, prelucrare și extragere de fișiere, cât și a operațiilor de preluare a stării sistemului, în ideea construirii unei interfețe de monitorizare, pentru administrator.

Principalul argument pentru folosirea unui modul de intermediere între aplicația client și nodurile generale ale sistemului este principiul separării responsabilităților. Nodurile sistemului au doar rolul de a gestiona logica de administrare a fișierelor utilizatorilor. Nodul general trebuie să se ocupe de gestiunea stării sistemului din punct de vedere al fișierelor ce trebuie să fie prezente în sistem și al stării nodurilor interne, în ceea ce privește fișierele stocate și integritatea acestora. Astfel, se evită supraîncărcarea prin adăugarea unei componente de tip server HTTP pentru gestiunea acestor cereri, preferându-se păstrarea tuturor comunicațiilor cu celelalte componente ale sistemului, prin intermediul socket-urilor, reprezentând mesajele prin intermediul obiectelor java beans. Acest fapt reprezintă un dezavantaj din punct de vedere al interoperabilității însă este mai eficient din punct de vedere al cantității de informație vehiculate prin canalele de comunicație, întrucât dimensiunea unui obiect va fi mai mică decât a unui mesaj *HTTP* care, pe lângă informațiile strict necesare prelucrării cererii (corpul mesajului și parametrii din *URI*), conține și un antet cu multe câmpuri pentru date referitoare la cerere. Totuși, abordarea

11 **C**reate **R**etrieve **U**pdate **D**elete

folosind *HTTP* reprezintă o soluție care poate asigura securitatea datelor mult mai eficient.

Un alt argument pentru folosirea intermediarului este eficiența sporită a transmiterii conținutului binar al fișierelor între componente dezvoltate în aceeași tehnologie. În cadrul sistemului, aplicația de tip client este dezvoltată în *JavaScript* folosind *React Framework,* iar nodurile interne cu care se va realiza transferul de fișiere sunt implementate în folosind *Java*.

Modulele *GeneralManager* și *GeneralNode* sunt module specifice care conțin funcționalitățile corespunzătoare nodului manager general și a nodurilor interne. Aceste tipuri de noduri comunică continuu în scopul efectuării de operații asupra fișierelor și validării stării acestora. Ambele module conțin funcționalități specifice comunicării în grupul de multicast, folosind obiectele corespunzătoare ce definesc un socket de multicast, din modului *GeneralPurposeStructures*. Totodată, ambele module conțin o componentă de comunicare cu clientul, prin vehicularea obiectelor, expuse de modulul *Beans.* Pe baza acestor cereri, se vor lua decizii în ceea ce privește starea sistemului și a fișierelor. Toată această activitate va genera anumite evenimente ce vor fi înregistrate în baza de date pentru a fi utilizate de administrator pentru monitorizare. Pentru a persista detalii despre aceste evenimente, se va folosi funcționalitatea expusă de modulul *HTTPHandler*.

Modulul *ui-frontend* este un modul specific și reprezintă aplicația de tip client, fiind implementată într-o tehnologie diferită de cea cu care au fost implementate celelalte module ale sistemului. Acest modul va expune, prin intermediul unei interfețe grafice de tip web, toate funcționalitățile sistemului, atât cele specifice utilizatorului obișnuit, referitoare la manipularea de fișiere, cât și cele specifice administratorului sistemului, referitoare la monitorizarea stării. Diferența dintre aceste două tipuri de utilizatori se va realiza prin intermediul rolului atribuit la crearea contului, rol ce va fi atașat în *token*-ul corespunzător fiecărei cereri trimise către sistem, folosit pentru autorizarea utilizatorului.

2.3. Tipuri de comunicații

În cadrul sistemului, se poate observa existența a trei tipuri principale de comunicații (vezi Figura 2.2).

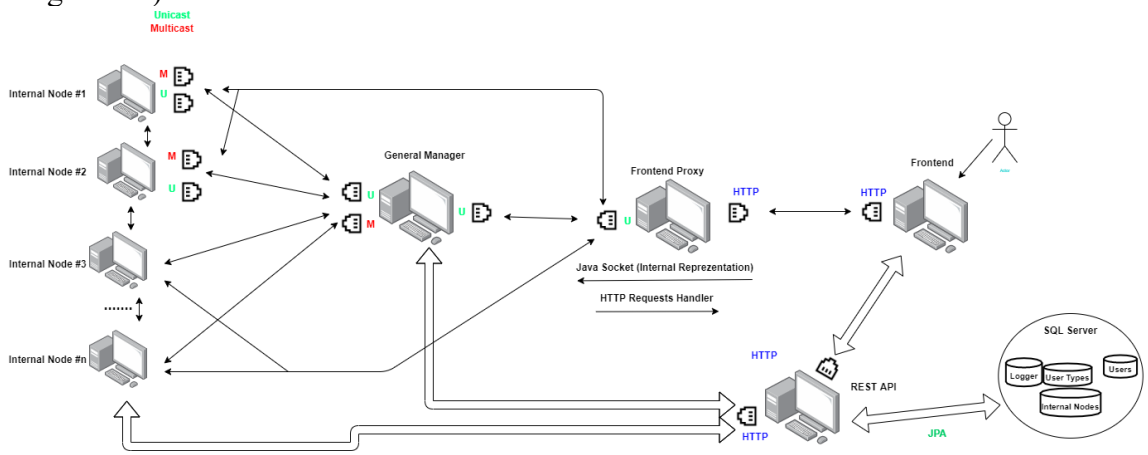

Figura 2.2: Modulele sistemului și tipuri de comunicatii

- Comunicarea de tip *multicast* se realizează între nodul general și nodurile interne, în vederea transmiterii de mesaje de tip *heartbeat*. Prin intermediul acestei comunicații se menține starea sistemului. Mesajele trimise de fiecare nod intern conțin detalii referitoare la cantitatea de memorie ocupată și la totalitatea fișierelor stocate, împreună cu versiunile și datele de integritate ale acestora. Mesajele trimise de nodul general reprezintă cereri de

verificare a integrității fișierelor. Nodul general solicită fiecărui nod intern suma de control corespunzătoare fiecărui fișier.

- Comunicarea de tip *unicast* se realizează între nodul general, nodurile interne și intermediarul clientului. Această comunicație are ca obiectiv lucrul cu fișiere (adăugare, actualizare și accesare) sau monitorizarea stării sistemului de către administrator. În cadrul acestei comunicații se transmit cereri și răspunsuri sub forma unor obiecte serializabile sau sub forma unui flux de octeți reprezentând conținutul fișierului.
- Comunicarea prin cereri de tip *HTTP* este folosită în interacțiunea cu clientul. În cadrul sistemului sunt puse la dispoziție două puncte de acces pentru aplicația de tip client. Aplicația de tip intermediar pentru client, care gestionează toate cererile de prelucrare de fișiere și monitorizare a stării și aplicația cu interfața de tip *REST* pentru gestionarea utilizatorilor, a detaliilor despre nodurile interne și a jurnalului evenimentelor apărute în sistem. Pentru a expune toate aceste detalii către utilizator, aplicația cu interfață REST primește cereri și de la nodurile sistemului, pentru înregistrarea evenimentelor. Astfel, și nodurile interne trebuie să poată trimite cereri de tip HTTP.

Pe lângă aceste comunicații între componentele sistemului, trebuie să se țină cont și de comunicarea cu serverul de baze de date, realizată din serverul cu interfață de tip REST. Această interacțiune se realizează prin intermediul JPA[12], o interfață care facilitează maparea obiectual-relațional, ajutând la persistarea obiectelor Java în baza de date, într-un mod transparent pentru utilizator [22].

2.4. Avantajele și dezavantajele metodei alese

Sistemul prezentat este un sistem distribuit cu o arhitectură de tip master-slave. Unul dintre mecanismele de bază ale sistemului este reprezentat de mecanismul de replicare a fișierelor pe nodurile interne ale sistemului (slave). Astfel, se poate deduce un prim avantaj al sistemului, acela de disponibilitate a datelor. Prin intermediul mecanismului de replicare, se asigură faptul că numărul de replici ale unui fișier este menținut constant în sistem pe întreaga durată a fișierului. Sistemul nu este sensibil la defecte singulare, întâmpinate la nivelul nodurilor interne. Prin mecanismul de verificare a integrității datelor folosind suma de control, se pot detecta fișierele deteriorate și se poate realiza replicarea pe alt nod, asigurând faptul că toate fișierele replicate au același conținut și că toate nodurile pot reprezenta un candidat pentru accesarea unui fișier, la solicitarea indirectă a utilizatorului.

În plus, această disponibilitate poate fi extinsă într-un mod eficient. Un alt avantaj al unei astfel de arhitecturi este determinat de posibilitatea scalării pe orizontală, a extinderii sistemului prin adăugarea mai multor noduri de calcul. Având în vedere faptul că disponibilitatea sistemului crește direct proporțional cu factorul de replicare al fișierelor, se poate extinde numărul de noduri ale sistemului și, explicit, factorul de replicare. Existând mai multe noduri în sistem, se poate asigura o replicare mai puternică a datelor. Drept consecință, în cazul situației de replicare, există mai mulți candidați care ar putea furniza fișierul ce trebuie replicat. Eventual, prin această scalare pe orizontală, s-ar putea distribui uniform nodurile din punct de vedere geografic, astfel încât, în cazul replicării să se considere nodurile cele mai apropiate geografic.

Principala resursă a sistemului este reprezentată de spațiul de stocare. Un alt avantaj este reprezentat de posibilitatea scalării de verticală, prin extinderea resurselor hardware disponibile la nivelul nodurilor interne, resurse ce adresează, în principal memoria. Prin adăugarea sau extinderea unității de stocare, nu sunt afectate performanțele nodului intern, ci se oferă

12 Java Persistence Api

posibilitatea stocării unui volum mai mare de date la nodul respectiv.

Astfel, sistemul poate fi extins foarte eficient folosind cele două modalități de scalare orizontală și verticală. Ambele metode îmbunătățesc disponibilitatea sistemului și măresc dimensiunea spațiului de stocare. În plus, un alt avantaj este reprezentat de mecanismul de echilibrare a încărcării nodurilor interne din punct de vedere a cantității de memorie ocupate. La apariția unui fișier nou în sistem, acesta este stocat pe nodurile care au cea mai multă memorie disponibilă, încercând să se păstreze un nivel constant al memoriei ocupate.

Această extindere și monitorizare a sistemului trebuie gestionată de nodul general. Extinderea numărului de noduri interne și mărirea volumului de date stocate în sistem presupun creșterea dimensiunii celor două tabele de conținut și status, pe care nodul general trebuie să le gestioneze, astfel încât să poată asigura monitorizarea eficientă a sistemului. Astfel, extinderea sistemului presupune creșterea încărcării nodului general. Din acest motiv, nodul general va persista aceste două tabele în memoria internă pentru a scădea timpul de acces la aceste date. Metoda nu este foarte eficientă întrucât pot apărea inconsistențe cauzate de multitudinea de operații efectuate în paralel pe aceste tabele, cu toate că se asigură mecanisme explicite de sincronizare. Eventualele inconsistențe pot duce la coruperea întregului sistem. O alternativă este reprezentată de persistarea acestor tabele într-un sistem separat de stocare, cum ar fi un sistem de baze de date însă, comunicarea cu un sistem de baze de date presupune scăderea performanțelor sistemului prin creșterea timpului de acces la resurse.

Se observă dependența sistemului de nodul master. Dacă acest nod devine inactiv, întregul sistem va fi oprit, până la revenirea nodului general. Astfel, se identifică un prim dezavantaj al sistemului, reprezentat de nodul general, care este un *single point of failure*[13]. Mai mult decât atât, cele două tabele pe baza cărora se face monitorizarea și gestionarea sistemului, alcătuiesc starea întregului sistem. Oprirea nodului general ar reprezenta și pierderea stării sistemului. Totuși, ținând cont de faptul că, încă de la începutul activității, nodul general primește mesaje *heartbeat* de la nodurile interne conținând starea stocării, starea sistemului se poate reface pe baza acestor date caracteristice adunate de la fiecare nod. Astfel, odată repornit, nodul general reface starea sistemului cu ajutorul mesajelor primite de la nodurile interne.

Un alt element de tip *single point of failure* este reprezentat de intermediarul dintre client și sistem, ce rulează într-o singură instanță. Oprirea acestui proces presupune imposibilitatea comunicării clientului cu sistemul. Totuși, acest proces este fără stare, tratând individual fiecare cerere. Astfel, oprirea și repornirea acestui serviciu nu ar introduce inconsistențe ulterioare ale stării sistemului.

Un alt dezavantaj este reprezentat de faptul că toate componentele interne ale sistemului (nodul general, nodurile interne și intermediarul clientului) sunt implementate în aceeași tehnologie și comunică prin obiecte specifice limbajului, serializate conform reprezentării interne a obiectelor corespunzătoare limbajului. Lipsa unui limbaj comun, universal, pentru comunicarea dintre componente reduce posibilitatea extinderii sistemului prin adăugarea unui modul implementat în altă tehnologie, care să ofere o performanță sporită. Totodată, se îngreunează procesul de implementare a mecanisme de securizare a datelor comunicate.

2.5. Limitele sistemului

Funcționarea corectă a sistemului implică, în primul rând, funcționarea corectă a mecanismului de replicare, prin menținerea numărului de replici ale unui fișier, constant pe întreaga durată de viață a fișierului. Acest mecanism asigură replicarea în cazul în care un nod

13 „Single point of failure” este o componentă din cadrul unui sistem, a cărei disfuncționalitate duce la inactivitatea întregului sistem.

intern pică. Însă, dacă pică mai multe noduri și numărul de noduri active rămase este mai mic decât factorul de replicare al unui fișier ce trebuie replicat, procesul de replicare nu va putea fi executat cu succes. Cu toate acestea, se va încerca replicarea până când un nod va deveni disponibil. Se observă astfel, o primă limitare. Sistemul funcționează corect și disponibilitatea este asigurată atâta timp cât numărul de noduri care pică nu este mai mare decât factorul de replicare maxim. Această limitare a sistemului devine nesemnificativă, pe măsură ce numărul de noduri ale sistemului devine din ce în ce mai mare.

În același context al nodurilor interne, o altă limitare este reprezentată de capacitatea de stocare. Sistemul nu mai poate suporta adăugarea de fișiere noi atâta timp cât nu există memorie disponibilă la nodurile interne. Cu toate că se încearcă echilibrarea stocării nodurilor prin redirectarea persistării noilor fișiere și a replicilor către nodurile cu cea mai puțină memorie ocupată, rămâne valabilă posibilitatea umplerii stocării tuturor nodurilor interne.

Nodul general implică anumite limitări prin starea sa. Sistemul funcționează corect cât timp nodul general funcționează corect, fiind singurul nod care persistă și monitorizează starea sistemului. Inactivitatea acestui nod implică oprirea întregului sistem. Prin repornirea nodului general și primirea de mesaje *heartbeat* de la nodurile interne, se poate recompune starea sistemului. Însă, starea sistemului, în forma de dinaintea eșuării nodului general se poate realiza doar dacă nu au apărut și alte erori la nivelul nodurilor interne. În timpul funcționării corecte a nodului general, factorul de replicare al unui fișier este determinat pe baza tipului utilizatorului, la adăugarea fișierului. În schimb, la repornirea nodului general, factorul de replicare al unui fișier este determinat de numărul de noduri care au raportat acel fișier. Astfel, dacă între timp un nod a picat, la repornire nodul general nu va primi un heartbeat de la acesta și va considera că fișierul are un factor de replicare mai mic cu o unitate, față de cel corect. Un astfel de scenariu reduce disponibilitatea fișierelor.

Intermediarul dintre client și nodurile sistemului reprezintă o limitare, prin faptul că este singurul mediator dintre cele două entități. Dacă acest nod va deveni inactiv, comunicarea dintre client și nodurile sistemul va fi imposibilă.

2.6. Componente software

2.6.1. Diagrame UML

Sistemul este alcătuit din șapte module, care interacționează prin intermediul unor relații de dependență, agregare, compunere sau moștenire. Modulele pot fi clasificate în două categorii: module generale și module specifice. Modulele specifice descriu funcționalitatea completă a unei componente din cadrul aplicației.

- GeneralManager, care înglobează toată funcționalitatea caracteristică nodului manager general.
- GeneralNode, care înglobează toată funcționalitatea caracteristică nodurilor interne.
- FrontendProxy, care înglobează toată funcționalitatea caracteristică intermediarului dintre client și celelalte componente interne ale sistemului
- RESTApi, care înglobează toată funcționalitatea caracteristică reprezentării obiectelor folosite în interacțiunea cu clientul (reprezentarea entității utilizator, tip de utilizator, nod intern sau jurnal).

Caracteristica principală a acestor module este că pot fi încărcate și rulate de pe mașini diferite. Astfel, interacțiunea dintre aceste module se realizează doar prin intermediul comunicației în rețea. Nu există o relație specifică programării orientate obiect între aceste

module. Însă, la nivelul fiecărui modul, sunt definite anumite relații între clase.

Modulele generale descriu funcționalități necesare în contextul modulelor generale. Atât între modulele generale, cât și între modulele specifice și cele generale, există relații între clase. Clasele ce vor descrie doar funcționalități vor fi referite prin relații de dependență, iar clasele ce vor descrie anumite obiecte necesare în definirea anumitor clase, vor fi referite prin relații de asociere, agregare sau compunere. Pentru a permite reutilizarea codului și păstrarea anumitor caracteristici în cadrul unor clase, se vor folosi relații de moștenire.

2.6.2. Modulul nodului general

Modulul *GeneralManager* are o arhitectură complexă deoarece este responsabil de menținerea stării întregului sistem (vezi Anexa 1.). Acesta conține o clasă principală *GeneralManager* și mai multe clase secundare care expun funcționalități specifice.

- Clasa *ClientCommunicationManager* expune funcționalitățile necesare interacțiunii cu clientul. Acesta are la dispoziție o gamă largă de operații disponibile. Fiecare operație va fi identificată și diferențiată prin intermediul obiectelor de tip *java beans*. Fiecare obiect primit, va avea corespondent un membru din enumerația *ClientRequest* pentru a eficientiza procesul de identificare a tipului operației. Totodată, operațiile de adăugare și actualizare a fișierelor sunt condiționate de existența fișierului. Astfel, se definește o altă enumerație *ClientRequestStatus,* având ca membri starea fișierului. Această clasă *depinde* de clasa *FeedbackManager,* responsabilă de gestionarea buclei de feedback în contextul relației cu clientul.
- Clasa *ReplicationManager* expune toată logică menținerii integrității sistemului, prin analiza stării și formularea unor cereri de replicare sau eliminare a unor fișiere din cadrul sistemului.
- Clasa *FileSystemManager* expune funcționalitățile necesare efectuării cererilor de prelucrare de fișiere, către nodurile interne. De această clasă depinde clasa ReplicationManager, pentru efectuarea cererilor de prelucrare a fișierelor.
- Clasa *HeartBeatManager* expune funcționalitățile specifice comunicației multicast. Expune metode de comunicare cu nodurile interne. În primul rând, primește mesaje *heartbeat* și alcătuiește starea sistemului și, în al doilea rând, trimite cereri către nodurile interne de solicitare a sumei de control pentru verificarea integrității fișierelor.

Cu excepția clasei *FilesystemManager,* toate clasele prezentate sunt *agregate* în clasa principală și rulate pe fire de execuție separate, astfel încât să execute funcționalitatea specifică în paralel, izolat de celelalte funcționalități. Pe lângă aceste clase, acest modul conține și tabelele de stare ale sistemului.

- Clasa *ContentTable* conține totalitatea fișierelor ce trebuie să existe în sistem, împreună cu toate operațiile aferente pentru actualizare și interogare. Această clasă trebuie să fie *serializabilă* pentru a putea fi trimisă către aplicația client, pentru monitorizarea efectuată doar de către administrator.
  - Toate fișierele conținute în această clasă sunt asociate utilizatorului corespunzător și conțin anumite detalii, descrise în clasa *FileAttributes,* ce este definită în modulul *Beans.* Fiind definită și folosită doar în acest context, între aceste clase avem o relație de compunere.
  - Fiecare fișier conține și anumite versiuni. Fiecare versiune trebuie să conțină anumite detalii precum numărul de versiune, suma de control, descrierea versiunii și data actualizării. O versiune este descrisă în clasa *VersionData*, fiind expuse și toate funcționalitățile specifice interogării și actualizării versiunii. Această clasă este

agregată în clasa ce definește tabela de content.

- Clasa *StorageStatusTable* înglobează starea sistemului de fișiere, pe baza mesajelor primite de la nodurile interne. Astfel, această clasă interpretează obiecte de tip *NodeBeat,* care definesc mesajul *heartbeat* al unui utilizator. Totodată, se expun metode de actualizare și interogare a acestei stări a sistemului. Toate fișierele conținute în această clasă sunt asociate utilizatorului corespunzător și conțin anumite detalii, descrise în clasa *FileAttributesForStorage,* definită în modulul *Beans*.
- Clasa *ConnectionTable* conține tabela de conexiuni cu nodurile interne. Tabela este actualizată pe măsură ce sunt recepționate mesaje de la nodurile interne. Dacă un nod nu mai trimite heartbeat, va fi eliminat din tabela.
- Clase *NodeStorageQuantityTable* conține tabela capacităților de stocare specifice nodurilor interne. Este actualizată la adăugarea sau eliminarea fiecărui fișier. Datele fiecărui nod intern sunt definite prin intermediul clasei *StorageQuantity.* Această clasă *compune* clasa NodeStorageQuantityTable și este serializabilă întrucât este trimisă către aplicația de tip client pentru monitorizarea stării sistemului de către administrator.

Aceste clase ce alcătuiesc starea sistemului sunt agregate în clasa GeneralManager, fiind folosite, în mod paralel și de celelalte clase de gestiune a sistemului (relație de dependență). Pe lângă aceste clase, se folosește și clasa *PendingQueue* în contextul adăugării unui nou fișier. Un nou fișier nu trebuie înregistrat în sistem imediat ce a fost realizată cererea de stocare, ci imediat după s-a primit confirmare de la nodurile interne, de stocare cu succes. Între aceste două momente, fișierul unui utilizator va sta într-o coadă de așteptare.

2.6.3. Modulul nodurilor interne

Modulul *GeneralNode* este alcătuit dintr-o clasă principală numită *GeneralNode,* în care sunt agregate și compuse toate celelalte clase necesare gestiunii sistemului și în care este interogată starea sistemului de fișiere, în vederea obținerii datelor ce trebuie trimise către nodul general prin heartbeat (vezi Anexa 2.).

- Clasa *ClientCommunicationManager*, agregată în clasa principală, este responsabilă de gestiunea comunicării cu clientul, pentru stocare de fișiere.
- Clasa *HeartBeatManager,* agregată în clasa principală, este responsabilă de gestiunea comunicării multicast. Sunt expuse funcționalități de trimitere a heartbeat-urilor către nodul general și de recepționare a cererilor de calculare și includere a sumei de control a fișierelor în cadrul următorului beat.
- Clasa *FileSystemManager* este agregată în clasa principală și este folosită în contextul comunicării cu nodul general, pentru efectuarea cererilor de prelucrare și interogare a sistemului de fișiere.
- Clasa *VersionControlManager* este agregată în clasa principală, fiind utilizată pentru gestionarea versiunilor fișierelor. Sunt expuse metode de înregistrare și interogare a versiunilor. Fiecare versiune a unui fișier este definită prin intermediul clasei serializabile *VersionData*.

Modulul conține și o clasă ce definește o listă de așteptare. Această clasă este agregată în clasa principală. Această listă de așteptare va fi folosită în contextul solicitării calculării sumei de control a unui fișier. Ținând cont de faptul că această calculare a sumei de control poate dura mai mult decât diferența dintre cererea efectivă și trimiterea următorului heartbeat, lista va persista cererea, până când suma de control va fi calculată.

2.6.4. Modulul intermediarului clientului

Modulul *FrontendProxy* este folosit ca un intermediar între client și nodurile sistemului, având ca principală caracteristică, transformarea cererilor HTTP în obiecte serializabile, ce vor fi trimise către nodurile sistemului (vezi Anexa 3.). Modulul este implementat folosind *SpringBoot* pentru a eficientiza procesul de tratare a cererilor HTTP. Astfel, framework-ul va gestiona componenta de tip controller, definită în clasa *FileController*, unde vor fi definite metodele corespunzătoare fiecărei cereri HTTP. Pentru fiecare solicitare din partea clientului, va fi formulat și un răspuns, sub forma unui dicționar cu perechi de tip cheie-valoare. Clasa *ResponseHandlerService* se va ocupa de gestiunea acestor răspunsuri, expunând metode de tratare în caz de succes, eroare sau particularizare. Fiecare cerere va solicita o prelucrare a sistemului de fișiere, care va fi efectuată de către nodurile interne. Astfel, controller-ul va depinde de clasa *FileService* care, pentru fiecare cerere HTTP, va crea un obiect serializabil, va trimite cererea către nodurile corespunzătoare și va întoarce un răspuns corespunzător către controller. Acest proces va fi facilitat de clasa *FrontendManager,* care va primi ca parametru obiectul de tip cerere, îl va trimite către nodurile interne și va întoarce un răspuns. În cazul operațiilor de adăugare și descărcare de fișiere, trebuie gestionat fluxul binar de octeți ce alcătuiesc fișierul. Astfel, clasa *FileService* depinde de clasa *FileSender*, care va gestiona trimiterea și primirea fluxului de octeți al fișierului. Totodată, va gestiona feedback-ul corespunzător operației de stocare. Acesta va fi extras dintr-o buclă de feedback gestionată în clasa *FeedbackManager,* agregată în clasa principală. Această clasă de control a feedback-ului va fi instanțiată și rulată pe un fir de execuție separat. Totodată, o caracteristică esențială a acestui modul este monitorizarea sistemului. La intervale regulate de timp, trebuie trimise către aplicația de tip client, starea sistemului (tabelele salvate la nivelul nodului general). În acest sens, se folosește un mecanism bazat pe WebSockets, o modalitate prin care o aplicație web poate recepționa anumite mesaje la anumite intervale de timp, ca urmare a unor evenimente interne apărute la nivelul serverului, și nu în urma unor cereri de tip HTTP. Această metodă are la bază modelul *publish-subscribe* conform căruia, după abonarea clientului la un anumit *topic,* prin intermediul metodei *subscribe*, acesta va recepționa toate mesajele trimise de server prin metoda *publish [23]*. Acest modul va conține o clasă de configurare a acestei comunicații, clasă numită *WebSocketConfig*, care va moșteni clasa *WebSocketMessageBrokerConfigurer*. După realizarea configurării, sunt expuse în cadrul clasei *WebSocketController* metode de interogare a stării sistemului, cu ajutorul clasei *FrontendManager* și trimiterii a stării către client. Aceste metode sunt apelate la intervale regulate de timp.

2.6.5. Modulul API-ului REST

Modulul *RESTApi* expune metodele necesare reprezentării obiectelor folosite în interacțiunea cu clientul, prin intermediul modelului arhitectural REST (vezi Anexa 7.). Acest modul este implementat folosind *SpringBoot*, care va facilita utilizarea *Spring Framework,* prin instanțierea un server de tip *Apache Tomcat* care va prelua cererile HTTP ale clienților, cereri în cadrul cărora vor fi solicitate diferite reprezentări ale obiectelor. *SpringBoot* va realiza de gestiunea fiecărui controller și injectarea dependințelor necesare, corespunzătoare fiecărui serviciu, astfel încât să se asigure o metodă de tratare a fiecărei cereri *HTTP [24]*. Totodată, acest modul se va ocupa de maparea obiectual-relațional. Fiecare obiect definit va putea fi persistat în baza de date, cu ajutorul interfeței *JPA*, specifice mapării obiectual relațional, în contextul folosirii bazelor de date relaționale de tip MySQL. În cadrul sistemului, sunt definite patru obiecte esențiale în interacțiunea cu clientul: utilizatorul (clasa *User*), tipul de utilizator (clasa *UserType*), nodul sistemului (clasa *InternalNode*) și evenimentul (clasa *Log*). Pentru fiecare

dintre aceste tipuri de obiecte, este definită o clasă care conține toți parametrii corespunzători, împreună cu metodele de acces. Aceste clase sun însoțite de adnotări specifice JPA, pentru a facilita maparea obiectelor la modelul relațional. Totodată, fiind folosit modelul arhitectural *REST*, sunt expuse toate metodele specifice creării, interogării, actualizării și eliminării de obiecte (*CRUD*). Aceste metode sunt definite în cadrul unui interfețe specifice fiecărui obiect (*UserDao*, *UserTypeDao*, *InternalNodeDao* și *LogDao*). Printr-o relație de moștenire, sunt implementate aceste interfețe în cadrul claselor de tip serviciu (*UserDaoService*, *UserTypeDaoService*, *InternalNodeDaoService* și *LogDaoService*). Aceste servicii expun metodele necesare efectuării tuturor operațiilor asupra obiectelor însă, având nevoie de persistarea datelor în baza de date MySQL, se definește o clasă parametrizabilă *MySqlManager* care expune toate metodele CRUD specifice bazei de date. Această clasă va folosi un obiect specific JPA, pentru tratarea unei sesiuni de comunicare cu serverul de baze de date. Clasa *EntityManagerHandler* va instanția un *EntityManager* cu ajutorul fabricii de entități, *EntityManagerFactory.* Astfel, clasa *MySQLManager* va prelua cererea de manipulare a obiectului și va apela metoda din EntityManager, care va efectua operația la nivelul serverului de baze de date. Având în vedere că fiecare serviciu folosește o instanță a clasei *MySQLManager,* pentru fiecare tip de obiect se va instanția câte un tip *EntityManager* care va efectua operații doar asupra tabelei corespunzătoare.

Pentru fiecare tip de obiect se definește o clasă de control (*UsersController*, *UserTypesController*, *InternalNodeController*, *LogController*), ce expune toate funcționalitățile specifice reprezentării obiectului. Având la dispoziție servicii care vor efectua aceste operații, se va folosi injectarea dependințelor, realizată în mod transparent de către framework. În cadrul clasei de control se va specifica doar interfața ce expune toate funcționalitățile, urmând ca serviciu care implementează funcționalitățile să fie injectat în mod automat.

2.6.6. Modulul structurilor de date și funcționalităților generale

Pe lângă aceste module specifice, sunt prezente și module generale folosite în contextul acestor module specifice. Modulul *GeneralPurpose* este alcătuit din mai multe pachete, fiecare expunând anumite funcționalități (vezi Anexa 4.)

- Pachetul *communication* conține metodele și obiectele specifice comunicării dintre nodurile sistemului (vezi Figura A.5). Este definită o clasă de tip înveliș pentru o adresă de rețea. Această clasă numită *Address* va expune cele două caracteristici ale unei adrese (adresa IP și portul) și va conține toate metodele de validare a acestora. Clasa va fi folosită în contextul modulelor generale, la definirea adreselor de rețea, și în contextul clasei *HeartBeatSocket*, prezentă în același pachet, responsabilă cu definirea tuturor caracteristicilor necesare comunicării multicast (funcții de trimitere și recepționare a mesajelor). Această clasă va fi folosită în contextul nodului general și al nodurilor interne, pentru gestiunea comunicării prin mesaje de tip heartbeat. Totodată, atât această clasă, cât și toate celelalte module interne, în contextul comunicării prin obiecte serializabile, folosesc clasa *Serializer*, definită în acest pachet. Clasa va expune toate metodele necesare serializării și deserializării obiectelor, pentru a fi vehiculate în rețea.
- Pachetul *config* conține o singură clasă *AppConfig* de configurare a sistemului, prin inițializarea tuturor parametrilor corespunzători fiecărui modul, în urma citirii unui fișier de configurare, existent la nivelul proiectului fiecărui modul (vezi Figura 2.3). Această clasă va fi referențiată de fiecare modul specific al sistemului.

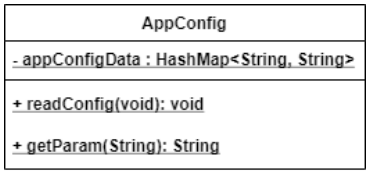

Figura 2.3: Clasa de configurare a fiecărui modul

- Pachetul *data* conține clasele ce definesc structuri de date generale, folosite la nivelul mai multor module (vezi Figura 2.4). Clasa *Pair* este o clasă serializabilă, folosită acolo unde se dorește definirea unei perechi de două valori. Fiind parametrizabilă, clasa suportă orice tip de dată pentru cei doi membri. Clasa *Time* conține funcționalități generale de lucru cu timpul. Expune mai multe formate de exprimare a timpului și este folosită, în special, în contextul versionării, pentru înregistrarea datei la care s-a produs modificarea sau în contextul jurnalizării evenimentelor apărute în cadrul nodurilor sistemului.

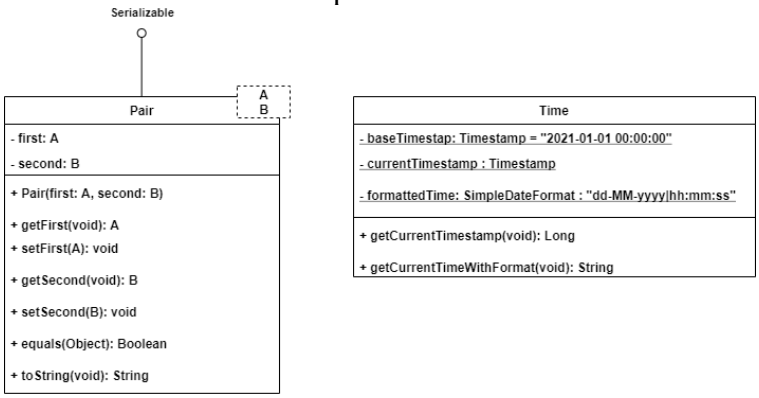

Figura 2.4: Clasele ce definesc structurile de date generale ale sistemului

- Pachet *os* conține clasa *FileSystem*, ce efectuează operații cu fișiere, la nivelul sistemului de operare. Sunt definite metode de creare de fișiere, eliminare, redenumire, actualizare, citire sau calcularea sumelor de control. Clasa este folosită în contextul tuturor operațiilor de lucru cu fișiere de la nivelul nodurilor interne (vezi Figura A.4 din Anexa 4.)

2.6.7. Modulul obiectelor serializabile

Modulul *Beans* este alcătuit din mai multe pachete, corespunzătoare capetelor celor trei comunicații principale din sistem.

- Pachetul *client_manager* conține clasele ce definesc obiectele java beans corespunzătoare comunicării dintre client și nodul general (vezi Figura A.7 din Anexa 5.).
- Pachetul *client_node* conține clasele ce definesc obiectele java beans corespunzătoare comunicării dintre client și nodurile interne (vezi Figura A.6 din Anexa 5.).
- Pachetul *node_manager* conține clasele ce definesc obiectele java beans corespunzătoare comunicării dintre nodul general și nodurile interne (vezi Figura A.8 din Anexa 5.)
- Pachetul *model* conține clasele ce definesc obiectele java beans corespunzătoare comunicării dintre componentele sistemului, în vederea definirii și reprezentării stării, pentru monitorizare (vezi Figura A.9 din Anexa 5.).

Fiind clase de tip *java beans*, aceste clase sunt serializabile, au membrii privați și modalități de accesare a acestora. Având o comunicare dintre două noduri, aceasta trebuie să fie bidirecțională. Astfel, obiectele pot reprezenta atât cereri, cât și răspunsuri. Pentru a reprezenta în

mod generic un tip de cerere la nivelul unei metode și pentru a permite reutilizarea codului, se folosește moștenirea. Astfel, în cazul unei cereri efectuate de la client către nodul general, cererea va fi de tipul clasei *ClientManagerRequest,* ce conține datele generale ale unei cereri (utilizatorul, numele fișierului și descrierea cererii). Pe baza acestei clase, prin moștenire, se vor crea mai multe clase, fiecare având anumite particularități. Spre exemplu, clasa *NewFileRequest* va conține dimensiunea și suma de control a fișierului. Astfel, în timpul execuției programului, programul va aștepta o cerere de tipul clasei părinte și va primi o cerere de tipul unei clase copil, pe baza căreia, va genera execuția acțiunii corespunzătoare. În aceeași măsură, în cazul răspunsului, se definește o clasă generică de răspuns, numită *ManagerResponse,* care va fi moștenită de cele două tipuri de răspunsuri, *ManagerTextResponse* și *ManagerComplexeResponse*, care diferă prin tipul de date al mesajului de răspuns. Această regulă se respectă și în celelalte cazuri.

2.6.8. Modulul caracteristicilor specifice HTTP

Modulul *HTTPHandler* expune obiectele și funcționalitățile caracteristice comunicării prin HTTP. În primul rând, pachetul *jwt* expune clasa *JWT*, ce definește caracteristicile principale ale unui jwt (vezi Figura A.10 din Anexa 7.). Această clasă este folosită în contextul clasei *AuthorizationService,* printr-o relație de dependență. Clasa asigură mecanisme de prelucrare a token-ului, în vederea gestionării identității și rolului utilizatorului. Sunt definite metode de codificare sau decodificare a token-ului, de verificare a valabilității și de autorizare efectivă. Clasa folosește o enumerație *UserType,* care expune toate rolurile posibile ale unui utilizator, precum și un membru *ALL,* care exprimă faptul că toți utilizatorii sunt autorizați să efectueze o anumită cerere. Aceste clase sunt folosite în contextul aplicațiilor ce expun o interfață pentru cereri de tip HTTP (modulul *RESTApi* și *FrontendProxy*), în vederea comunicării cu clientul, pentru autorizarea clientului în efectuarea operațiilor. Totodată, acest modul conține și funcționalități specifice efectuării de cereri HTTP și jurnalizării evenimentelor apărute la nivelul nodurilor sistemului (vezi Figura A.11 din Anexa 7.). Clasa *HTTPConnectionService* expune funcționalitățile necesare efectuării de cereri HTTP, prin intermediul unui obiect de tip *HttpURLConnection,* care generează o conexiune HTTP cu clientul. Sunt expuse toate metodele specifice de tip GET, PUT, POST și DELETE, precum și posibilitatea interpretării răspunsului. Cele patru tipuri de cereri vor fi identificate pe baza unei clase de tip enumerație *HTTPMethod.* Corpurile cererilor de tip *HTTP* vor fi reprezentate în format json. Pentru a facilita serializarea și deserializarea obiectelor în format json, se va folosi clasa *ObjectMapper,* importată din modulul *com.fasterxml.jackson*, corespunzător Maven. Aceste funcționalități vor fi folosite în cadrul clasei *LoggerService,* pentru jurnalizarea evenimentelor. Se vor trimite cereri HTTP către modulul *RESTApi,* pentru înregistrarea evenimentelor în baza de date.

2.6.9. Diagrame ER

Sistemul de baze de date folosit în cadrul sistemului este un sistem relațional de tip SQL. Obiectele sunt persistate în cadrul bazelor de date prin intermediul *JPA*, o interfață de mapare obiectual-relațional. Structura bazei de date este evidențiată în Figura 2.5.

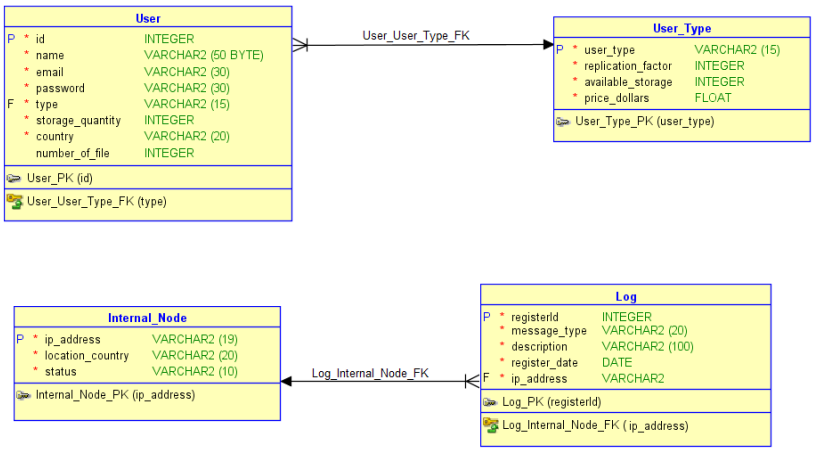

Figura 2.5: Diagrama ER

Tabela *User* este folosită pentru persistarea utilizatorilor sistemului. Fiecare utilizator este identificat pe baza cheii primare, câmpul *id* de tip întreg. De asemenea, utilizatorul are anumite date de contact, cum ar fi numele (*name*), email-ul (*email*), parola (*password*), țara de proveniență (*country*), cantitatea de memorie ocupată (*storage_quantity*) și numărul de fișiere pe care le deține (*number_of_file*). Totodată, sistemul poate avea mai multe tipuri de utilizatori. De aceea, tabela conține un câmp *type* ce identifică tipul utilizatorului. Acest câmp reprezintă o cheie străină ce referențiază cheia primară a tabelei *User_Type*, cheie ce reprezintă numele tipului (*user_type*). Corespunzător fiecărui tip de utilizator sunt factorul de replicare (*replication_factor*), cantitatea totală de memorie disponibilă (*available_storage*) și prețul exprimat în dolari pentru a avea tipul respectiv de utilizator (*price_dollars*). Astfel, un utilizator este de un anumit tip predefinit, însă poate extinde facilitățile tipului, crescând factorul de replicare și cantitatea de memorie disponibilă, prin plata unui anumit abonament lunar.

Tabela *Internal_Node* conține date despre toate nodurile interne ale sistemului. Fiecare nod este identificat în mod unic prin intermediul cheii primare, adresa IP (*ip_address*), exprimată sub forma unui șir de 19 caractere. Totodată, fiecare nod intern se află într-o anumită țară (*location_country*). Persistarea datelor referitoare la locația utilizatorului și a nodurilor interne va facilita optimizarea accesului la date în contextul scalării orizontale. Se va urmări ca un utilizator să acceseze fișierul propriu de la nodul cel mai apropiat din punct de vedere geografic. Totodată, fiecare nod intern are un *status,* ce exprimă dacă acesta este pornit sau oprit. Această tabelă este folosită atât la interogări directe din partea administratorului sistemului, cât și în cadrul tabelei *Log,* pentru a referenția nodul în cadrul căruia a fost generat un anumit eveniment. Astfel, tabela *Log* conține o cheie străină *ip_address* care va referenția cheia primară cu același nume din tabela *Internal_Node*. Tabela *Log* va fi folosită pentru a jurnaliza toate evenimentele apărute în cadrul sistemului. Fiecare eveniment va fi identificat în mod unic prin intermediul unei chei primare *registerId*, sub forma unu întreg, va conține descrierea evenimentului (*description*) și data efectuării evenimentului (*register_date*).

2.6.10. Tehnologia aleasă

Sistemul este alcătuit din mai multe module, implementate folosind diverse tehnologii. Cu excepția aplicației de tip client, toate modulele sunt implementate în limbajul *Java*. Principalul argument pentru folosirea acestui limbaj este independența de platformă. Fiind un sistem ce poate fi foarte eficient scalat pe orizontală, se oferă posibilitatea ca nodurile interne să poată fi rulate de pe mașini diferite, pe anumite sisteme de operare sau în cadrul unor sisteme de containere. Astfel, rularea aplicației trebuie să fie independentă de mediul în care rulează, asigurând totodată comunicarea cu aplicații din medii diferite fără nevoia unui intermediar.

Modulele generale ale sistemului trebuie să comunice prin intermediul rețelei, folosind obiecte serializabile de tip Java Beans. Cu ajutorul acestor obiecte, vor fi reprezentate cererile utilizatorilor. Având în vedere că, din perspectiva aplicației, toate cererile trebuie să fie interpretate la fel, trebuie să folosim conceptele de moștenire și polimorfism. Se va defini o clasă generică de tip cerere a clientului și, prin moștenire, se vor crea diverse cereri specifice. Pe baza tipului cererii (a clasei copil), se va putea diferenția între diverse tipuri de operații (polimorfism). Limbajul Java facilitează astfel de operații întrucât este un limbaj orientat obiect în care, fiecare entitate a sistemului este reprezentată sub forma unei clase, care poate fi moștenită și extinsă. Se poate folosi avantajul polimorfismului, pentru a defini un parametru ca fiind o clasă părinte, urmând ca, în timpul execuției, să se furnizeze ca parametru o clasă copil, astfel încât să se efectueze o altă operație [25].

În aceeași măsură, aceste clase trebuie să fie serializabile. Limbajul Java oferă suportul nativ pentru a defini obiecte serializabile, ce pot fi transmise ca flux binar de octeți în rețea. Aceasta se poate realiza prin simpla moștenire a clasei *Serializable*. Totodată, având un sistem a cărui principal obiectiv este lucrul cu fișiere la nivelul sistemului de operare, sunt necesare metode de prelucrare a fișierelor. Limbajul Java oferă toate funcționalitățile dorite (creare de fișiere și directoare, citire completă, pe linii, sau caracter cu caracter a fișierului, modificarea, redenumirea, eliminarea și, foarte important, calcularea sumei de control).

Sistemul trebuie să gestioneze simultan mai mulți utilizatori. Din acest motiv, sunt necesare atât tehnici de calcul paralel, cât și tehnici de sincronizare a obiectelor folosite în mod paralel. Limbajul Java oferă suport pentru efectuarea operațiilor pe mai multe fire de execuție cu posibilitatea gestionării acestora. Pe lângă aceasta, Java pune la dispoziție și o gamă foarte variată de structuri de date. Aceste structuri de date pot avea suport implicit pentru sincronizare (*ConcurrentHashMap*) sau se pot defini mecanisme explicite de sincronizare pentru calculul paralel (*synchronized*) [26].

Sistemul trebuie să faciliteze comunicarea în rețea, atât prin intermediul socket-urilor de date, cât și prin intermediul HTTP. Limbajul Java oferă suport pentru definirea mecanismelor ce facilitează diverse tipuri de comunicații. Pentru comunicații *unicast,* se pot defini componente de tip *Socket* (client) și *ServerSocket* (server), împreună cu toate metodele necesare conectării și transmiterii de mesaje. Pentru comunicații de tip *multicast* se pot defini componente specifice de tip socket (*MulticastSocket*), a adresei specifice care va identifica grupul de multicast (*InetAddress*), împreună cu toate mecanismele necesare alăturării (*joinGroup*) și părăsirii (*leaveGroup*) grupului. Sunt puse la dispoziție și obiecte pentru înglobarea unui mesaj ce va fi transmis pe adresa de multicast (*DatagramPacket*). Totodată, se oferă suport și pentru comunicarea prin intermediul cererilor HTTP. În cadrul limbajului Java, sunt disponibile mecanisme de crearea unei conexiuni HTTP (clasa *HttpURLConnection*), pentru care se pot defini tipul cererii, anumite headere și corpul mesajului. Pentru fiecare cerere trimisă, se poate primi și un răspuns care poate fi deserializat și interpretat.

Un alt argument pentru alegerea limbajului Java este SpringBoot Framework, care

facilitează dezvoltarea aplicațiilor care tratează cereri de tip HTTP, prin intermediul unei aplicații de control care va defini o metodă de tratare pentru fiecare cerere. Avantajul acestui framework este injectarea dependințelor. Se pot defini interfețe care să descrie comportamentul unui serviciu, iar implementarea efectivă a serviciului, realizată prin moștenirea interfeței, să fie injectată direct în contextul aplicației de control. Prin intermediul acestui framework se vor putea descrie și anumite standarde arhitecturale, precum *REST*. Se poate defini reprezentarea unui obiect, împreună cu toate metodele de prelucrare a acestuia. Această reprezentare a obiectului va putea fi vehiculată foarte eficient prin intermediul cererilor *HTTP*. În contextul folosirii acestui standard arhitectural, se va putea realiza și o mapare obiectual-procedural, care să faciliteze salvarea reprezentării obiectelor într-o bază de date relațională. Limbajul Java oferă suport pentru comunicarea cu serverele de baze de date și pentru maparea obiectual-procedural prin intermediul unor interfețe precum JPA care, prin definirea claselor și adăugarea adnotărilor corespunzătoare, poate asigura persistarea obiectelor în baza de date, în mod transparent [22]. De asemenea, în contextul cererilor de tip HTTP, se pot asigura mecanisme de securizare a datelor sensibile ale utilizatorilor, prin intermediul unor standarde precum *Json Web Token.* Limbajul Java pune la dispoziție mecanisme de codificare/decodificare și semnare digitală.

Avantajul major al acestui framework dezvoltat pentru java este buna integrare cu sistemele de împachetare, precum *Maven*. O caracteristică importantă a unui sistem de build este posibilitatea de adăugare de funcționalități noi, specifice, din depozitele disponibile online pentru sistemul de împachetare. Drept exemplu, pentru a facilita interpretarea corpului cererilor HTTP în contextul modelului arhitectural REST, se va folosi modulul *Jackson* (clasa *ObjectMapper*), care va facilita serializarea și deserializarea unor corpuri de cereri, în obiectele corespunzătoare, definite de utilizator sau în structuri de date generale ale limbajului (*HashMap*). O altă caracteristică a unui astfel de sistem este procesul de împachetare, în cadrul căruia se pot atașa toate dependințele necesare împachetării și rulării aplicației în mod independent [27].

Spre deosebire de aceste module, aplicația de tip client, ce expune interfața web cu care va interacționa clientul în vederea comunicării cu sistemul, va fi implementată folosind framework-ul React. Acesta expune toate elementele necesare dezvoltării de aplicații web de tip *Single Page Application*. Sunt puse la dispoziție elemente DOM[14] specifice care pot fi folosite în combinație cu elementele DOM generale ale limbajului *HTML.* Pot fi folosite elemente de stil, care sunt definite în limbajul CSS. Logica aplicației va fi definită în limbajul Javascript. Pe lângă elementele specifice limbajului, acest framework adaugă componentele de stare. La efectuarea unui eveniment care implică actualizarea unor elemente de stare, se vor apela anumite funcții specifice, al căror comportament va putea fi definit de utilizator. Aceste evenimente vor genera și re-randarea paginii și actualizarea în conformitate cu noua stare a sistemului. Acest aspect conferă o manevrabilitate foarte eficientă a elementelor grafice din cadrul interfeței [28]. Acest framework oferă și posibilitatea definirii de servicii care să implementeze o anumită funcționalitate necesară, mai ales în contextul comunicării cu sistemul. Prin intermediul acestor servicii, se vor defini toate metodele necesare extragerii stării sistemului și efectuării de operații cu fișierele utilizatorilor. Totodată, acest framework expune și metodele de definire și manipulare a componentelor de tip *websocket*. Odată definit un astfel de websocket, se definesc topic-urile la care clientul se abonează și metodele care vor fi apelate atunci când un mesaj va fi recepționat.

2.6.11. Descrierea claselor dezvoltate

Modulele specifice implementează o anumită funcționalitate a sistemului. Nodul general va gestiona starea sistemului și coordonarea nodurilor interne. Intermediarul clientului va media

14 Document Object Model este o interfață de programare specifică HTML care definește structura logică a documentelor și modul în care un document este accesat și manipulat.

interacțiunea dintre client și nodurile sistemului. Nodurile generale vor stoca și gestiona replicile fișierelor și își vor transmite starea către nodul general. La nivelul acestor module, aceste funcționalități sunt implementate în cadrul claselor, fiind gestionate la nivelul modulului ținând cont de tehnicile de paralelizare și sincronizare astfel încât fiecare componentă să poată trata mai multe cereri și operații simultan.

2.6.12. Clase specifice nodului manager general

La nivelul modulului *GeneralManager*, care va conține logica nodului general, sunt disponibile clase care implementează gestionarea stării sistemului și coordonarea nodurilor interne. Clasa *ReplicationManager* expune toate funcționalitățile specifice menținerii disponibilității, prin menținerea factorului de replicare constant pentru un anumit fișier. (Anexa 8.). Acest mecanism va trebui să ruleze în mod continuu, fără a fi blocat de celelalte evenimente și activități din sistem. Din acest motiv, tot acest proces va fi rulat pe un fir de execuție separat, ce va fi pornit la nivelul clasei principale. Pentru a expune funcționarea paralelă, va moșteni clasa *Runnable* și va suprascrie funcția *run,* în care se va defini tot comportamentul acestui mecanism. Acesta se va executa cu o frecvență indicată prin *replicationFrequency*, membru al clasei ce va fi inițializat din fișierul de configurare. Deciziile de replicare sau eliminare a unor replici se vor lua pe baza comparării celor două tabele de stare ale sistemului, atât prin identificarea modificărilor factorului de replicare, cât și prin modificarea sumei de control sau a versiunii fișierului (funcția *checkForFileCorruption*). Operația de replicare va presupune găsirea unor noduri care să poată furniza o replică (funcția *searchCandidatesForReplication*) iar operația de eliminarea va presupune găsirea nodurilor care trebuie să elimine replicile (*searchCandidatesForDeletion*). Cele două operații vor fi implementate în funcțiile *replication* și *deletion*.

O altă clasă importantă în cadrul acestui modul este clasa HeartbeatManager care expune toată logică gestionării comunicației de tip multicast (vezi Anexa 9.). În mod similar, va trebui să ruleze în mod paralel, fără a afecta celelalte activități ale nodului general, motiv pentru care va implementa clasa *Runnable* și va conține comportamentul în funcția *run*. Totodată, nodul general trebuie să gestioneze recepționarea și înregistrarea mesajelor heartbeat de la nodurile interne, (funcția *receivingLoop*), verificarea și curățarea tabelei de conexiuni (funcția *cleanUp*) și trimiterea cererilor de calculare a sumelor de control pentru verificarea integrității fișierelor (*requestCRC*). Aceste trei operații trebuie să se execute în paralel, independent unele de celelate. Din acest motiv, fiecare va returna un obiect de tip *Runnable* va rula pe un fir de execuție separat. Fiecare operație va fi executată în mod continuu, cu frecvențe indicate de membrii clasei, instanțiați pe baza fișierului de configurare.

2.6.13. Clase specifice nodurilor interne

Modulul GeneralNode va expune comportamentul nodurilor interne. Acestea trebuie să gestioneze atât comunicarea cu clientul pentru a recepționa și trimite fișiere, cât și comunicarea cu nodul manager general. Logica interacțiunii cu clientul va fi expusă în clasa *ClientCommunicationManager* (vezi Anexa 10.). Pentru a putea primi cereri de la client, nodul general va trebui să instanțieze un server. Fiecare cerere a clientului va fi gestionată în mod paralel, pentru a putea permite recepționarea și gestionarea mai multor cereri simultan (*clientCommunicationLoop*). Gestionarea operației solicitate se va efectua în funcția *clientCommunicationThread.* Se vor aștepta doar două tipuri de cereri din partea clientului, de încărcare (obiectul *FileHeader*) sau descărcare (*DownloadFileRequest*) de fișier. Operația de încărcarea va fi presupune recepționarea fișierului de la client sau de la alt nod intern și

trimiterea către următorul nod. Datele despre nodurile următoare din lanț vor fi prezente în token-ul din *FileHeader*, care va trebui verificat (*validateToken*) și curățat (*cleanChain*) înainte de folosire. Totodată, operația de încărcare a fișierului va fi urmată de o operație de trimitere a statusului operației către client, pentru a se putea înregistra fișierul la nivelul nodului general (funcția *sendFeedbackToFrontend*).

Pe lângă aceste operații solicitate de client, nodul general va efectua anumite operații solicitate de nodul general, de prelucrare a fișierelor (clasa *FileSystemManager*). Totodată, una din caracteristicile nodului intern este de a persista și furniza versiunile fișierelor. Aceste versiuni vor fi stocate în fișierul de metadate corespunzător fiecărui fișier. Logica gestionării acestui fișier va fi expusă în clasa *VersionControlManager* (vezi Anexa 11.). Sunt disponibile metode pentru înregistrarea unei versiuni (funcția *registerFileVersion*) și pentru extragerea versiunilor (funcțiile *getLastVersionOfFile* și *getVersionsForFile*).

2.6.14. Clase specifice intermediarului clientului

Modulul FrontendProxy expune toată logica intermediarului dintre client și sistem. Acesta are rolul de a primi cereri HTTP din partea clientului și de a formula cererile către sistem, în formatul intern al reprezentării obiectelor. Principalele operații expuse de acest intermediar, care presupun interacțiune atât cu nodul general cât și cu nodurile interne sunt operațiile de încărcare și descărcare de fișiere, a căror logică este expusă în clasa *FileSender* (Anexa 12.). Operația de încărcare a unui nou fișier în sistem (funcția *sendFile)* are nevoie de lista nodurilor care vor putea stoca replicile. Această listă primită de la nodul general sub forma unui token, va trebui validată (*validateToken*). Adresele nodurilor interne care vor stoca replicile vor fi extrase din token prin intermediul funcției *getAddressesFromToken*. După ce fișierul a fost trimis către nodurile interne, se va aștepta un răspuns de la acestea (funcția *waitForFeedback*). În urma recepționării răspunsurilor, se va identifica răspunsul general și se va trimite un răspuns către nodul general (*sendFeedbackToGM*). În schimb, operația de descărcare a unui fișier (*downloadFile*) va primi ca parametru adresa nodului intern către care va trimite cerere de descărcare. Apoi, în urma recepționării fișierului, îl va stoca în memoria internă și va returna calea spre locația fișierului către client. Pe lângă aceste operații care se deosebesc prin faptul că presupun și interacțiune cu nodurile interne, sunt disponibile și operațiile de prelucrare a fișierelor și versiunilor acestora, dar și a stării sistemului. Cererile corespunzătoare acestor operații vor fi trimise către nodul general. Tratarea tuturor acestor cereri se va realiza în clasa *FrontendManager*, prin intermediul funcției *managerOperationRequest*, care va profita de avantajele polimorfismului pentru a identifica tipul cererii solicitate și al răspunsului furnizat. Obiectul generic de tip cerere va fi *ClientManagerRequest,* iar răspunsul furnizat va fi de tip ManagerResponse. Cererea se va trimite către nodul general, iar răspunsul primit de la acesta, împreună cu eventualele excepții generate vor fi returnate către client, excepție făcând doar operațiile de încărcare și descărcare de fișier, care vor folosi acest răspuns pentru comunicarea cu nodurile interne (funcția *MainActivity* pentru gestiunea operației de încărcare a unui fișier).

Capitolul 3. Implementarea aplicației

3.1. Descrierea generală a implementării

Implementarea sistemului a fost realizată, în principal, folosind limbajul Java, excepție făcând doar aplicația de tip client care expune interfața de tip web cu care va interacționa utilizatorul. Folosind limbajul Java, un limbaj orientat-obiect, toate modulele interne ale sistemului au fost organizate și implementate folosind clase care interacționează prin relații de moștenire, asociere, agregare, compunere sau dependință. Pentru a profita de reutilizarea codului, au fost definite anumite module generale, care au fost referențiate și folosite în cadrul mai multor module interne specifice.

Sistemul trebuie să efectueze o serie de operații prin care este menținută starea sistemului, sunt generate comenzi de prelucrare a fișierelor, se asigură comunicarea cu clientul și se asigură sincronizarea cu celelalte noduri. Pentru a efectua toate aceste operații, modulele trebuie să aibă definită câte o clasă pentru fiecare proces în parte. Această clasă va trebui instanțiată iar acțiunile corespunzătoare noului obiect să fie rulate pe un fir de execuție separat, astfel încât să nu blocheze celelalte procese. La nivelul fiecărui modul, aceste obiecte vor trebui să interacționeze între ele. Din acest motiv, se vor referenția instanțele publice și statice ale acestor obiecte, definite la nivelul claselor principale.

Pentru a se putea realiza comunicarea în rețea, la nivelul fiecărei componente se vor deschide componente de tip *socket*. În funcție de tipul de comunicare (*unicast* sau *multicast*) sau de operația solicitată, se vor folosi diverse tipuri de obiecte de tip socket.

- *ServerSocket,* pentru a defini funcționalități de tip server, corespunzătoare comunicațiilor unicast.
- *Socket,* pentru a defini funcționalități de tip client, corespunzătoare comunicațiilor unicast.
- *MulticastSocket*, corespunzătoare comunicațiilor multicast, pentru a realiza transmiterea de mesaje de tip heartbeat (sub forma de *DatagramPacket*) între nodul general și nodurile interne.
- *WebSocket,* pentru implementarea paradigmei *publish-subscribe*, în comunicarea cu clientul, pentru transmiterea stării sistemului, în vederea monitorizării.

Fiecare interacțiune în rețea dintre două componente trebuie gestionată paralel, independent de celelalte interacțiuni. Astfel, la fiecare conectare a unui nou client, procesul de interacțiune cu acesta va fi executat pe un fir de execuție separat, astfel încât să nu se blocheze canalul de comunicație, ci să poată fi tratate și conectările altor clienți.

Starea sistemului va fi definită prin intermediul unor clase de tip înveliș, realizat asupra unor structuri de date. În cadrul nodului general, aceste tabele de stare, reprezentate sub forma unor clase, vor avea ca membri o structură de date (de tip *listă* sau *dicționar*), iar ca metode, vor fi disponibile toate operațiile necesare accesării și prelucrării acestor date. Având o singură stare a sistemului, fiecare dintre aceste tabele trebuie să existe sub forma unei singure instanțe. De aceea, fiecare clasă va fi instanțiată la nivelul clasei principale, cu o instanță statică, ce va fi referită de toate obiectele ce prelucrează starea sistemului. În aceeași măsură, toate operațiile ce presupun manipularea stării sistemului se vor putea executa în paralel. Din acest motiv, se asigură mecanisme explicite de sincronizare la nivelul acestor structuri de date din cadrul claselor. Astfel, toate operațiile de prelucrare a acestor structuri de date vor fi precedate de o operație de sincronizare (*synchronized*). Tot în ideea sincronizării, aceeași operație va fi executată și la nivelul componentei ce va realiza maparea obiectual-relațional, în vederea persistării datelor utilizatorului în baza de date. Având în vedere că cererile de stocare în baza de

date vor fi generate în mod concurent, trebuie asigurat faptul că acea componentă care asigură comunicarea cu serverul de baze de date și prelucrarea datelor (componenta de tip *EntityManager* specifică *JPA*), este accesată în mod sincronizat. Se va folosi același cuvânt cheie *synchronized*.

În cadrul cererilor transmise de client către sistem, sunt atașate informații sensibile, de identificare a clientului. Sunt asigurate mecanisme prin care aceste date sensibile ale utilizatorilor sunt codificate și semnate digital, prin folosirea standardului *JWT*. Pentru a facilita generarea unui token pentru fiecare sesiune de comunicare cu clientul, se folosește modulul *io.jsonwebtoken* din depozitul Maven. Acest modul expune funcționalități specifice de creare a acestui token (*JwtBuilder*), decodificare (*Jwts* și *Claims*) și semnare (*SignatureAlgorithm*, *Key* și *SecretKeySpec*) [29]. Aceste funcționalități vor fi folosite în contextul fiecărei cereri a utilizatorilor, pentru autorizarea efectuării unei anumite operații, token-ul fiind atașat în header-ul *Authorization* a cererii HTTP (*BEARER <token>*).

O altă componentă importantă a sistemului este reprezentată de asigurarea integrității, realizată prin intermediul calculării și comparării frecvente a sumei de control. Pentru fiecare fișier existent în sistem, se impune ca suma de control înregistrată la apariția unei versiuni a fișierului să rămână neschimbată pentru toate replicile, pe întreaga durată de viață a versiunii. Astfel, nodurile interne vor calcula suma de control a fiecărui fișier și o vor trimite către nodul general. Pentru a calcula suma de control se vor folosi metode specifice Java, precum clasa *CRC32* din modulul *java.util.zip*. Se va parcurge fișierul și, pentru fiecare pachet de date citit, obiectul instanțiat pe baza acestei clase va calcula și actualiza suma de control corespunzătoare fișierului.

Toate aceste elemente specifice vor contribui la realizarea tuturor operațiilor disponibile în cadrul sistemului.

3.2. Modalități de rezolvare, probleme speciale și dificultăți întâmpinate

Sistemul distribuit de stocare de fișiere expune toate operațiile corespunzătoare prelucrării de fișiere, astfel încât să se respecte și principiile de disponibilitate și integritate a datelor. Mai întâi, fiecare utilizator este identificat în cadrul sistemului prin intermediul unui cont de utilizator. Autentificarea utilizatorului se realizează prin efectuarea unei cereri către serverul ce expune o interfață de tip *REST Api*, care realizează o conexiuni cu serverul de baze de date relaționale de tip *SQL*, prin intermediul *JPA*. După ce sunt extrase și validate credențialele utilizatorului, se generează un *JWT* care va conține id-ul utilizatorului, numele și rolul acestuia. Token-ul va fi semnat cu o cheie privată, folosind algoritmul *HS512* și apoi va fi trimis către client în corpul răspunsului. Odată primit de către client, acest token va fi salvat în *localstorage* disponibil pentru browser, urmând să fie folosit pentru a identifica clientul care va solicita o anumită operație. Token-ul va fi atașat fiecărei cereri a clientului, în header-ul *Authorization* a cererii HTTP. După expirarea token-ului, utilizatorul va fi atenționat, trebuind să se reautentifice. Tot acest proces poate fi observat în Figura 3.1.

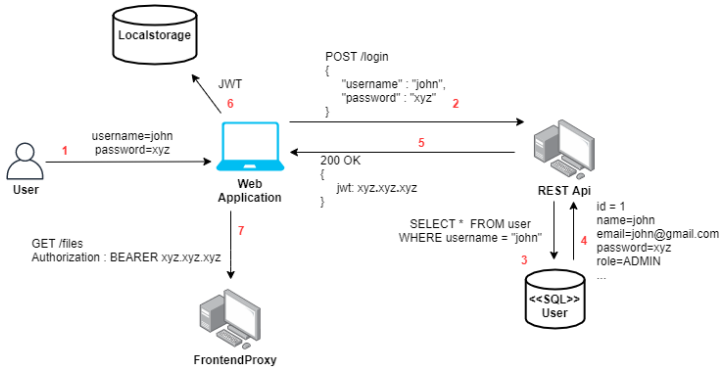

Figura 3.1: Procesul de autentificare a utilizatorului

După ce utilizatorul a fost autentificat, acesta va putea solicita efectuarea anumitor operațiuni la nivelul sistemului, în funcție de rolul acestuia, prin trimiterea unor cereri HTTP către intermediarul clientului. Pentru fiecare cerere de tip HTTP, se va crea obiectul serializabil corespunzător, care va fi trimis către componentele interne ale fișierului, urmând să se efectueze operația solicitată de client. Dacă va avea rolul de utilizator obișnuit, va putea efectua doar operațiuni de prelucrare a propriilor fișiere și a propriului cont de utilizator. În schimb, dacă va avea rolul de administrator, va putea monitoriza starea sistemului, prin interogarea serverului cu interfață de tip REST pentru a prelua detalii despre evenimentele jurnalizate în baza de date și prin crearea unei conexiuni cu intermediarul, folosind *websockets*. Această conexiune va facilita folosirea modelului *publish-subscribe*. Aplicația de tip client se va abona (subscribe) la cele 5 subiecte (*topics*) de stare .

- /*content* va expune tabela de conținut a sistemului. Aceasta va conține toate fișierele existene în sistem, împreună cu detalii de versiune, factor de replicare, sumă de control.
- */nodes* va expune tabela capacităților de stocare corespunzătoare nodurilor interne. Pentru fiecare nod, se vor prezenta detalii de identificare (adresa) și cantitățile de memorie, totală și disponibilă.
- */storage* va expune tabela de status a sistemului. Aceasta va fi alcătuită pe baza mesajelor de tip heartbeat primite de la nodurile interne și va conține detalii despre replicile existente în sistemul de fișiere a fiecărui nod, împreună cu detalii despre versiunile fișierului și suma de control.
- */replication* va expune mesajele rezultate în urma procesului de verificare a integrității sistemului. Pentru fiecare fișiere, se va evidenția dacă starea acestuia este validă, dacă trebuie replicare sau eliminarea unei replici.
- /*connection* va expune tabela de conexiuni, prin prezentarea tuturor nodurilor interne conectate la nodul general.

După ce a fost realizată conexiunea cu intermediarul, clientul va primi aceste mesaje la intervale regulate de timp, pe măsură ce serverul le publică. Însă, aceste date pot fi foarte voluminoase, iar administratorul sistemului putea monitoriza doar un anumit subiect la un moment dat. Astfel, pentru a elimina încărcarea canalului de comunicație, clientul se va abona doar la un anumit topic la un moment dat, în funcție de fereastra expusă în interfață. Acest comportament poate fi observat în Figura 3.2 (corespunzător utilizatorului obișnuit) și Figura 3.3 (corespunzător administratorului).

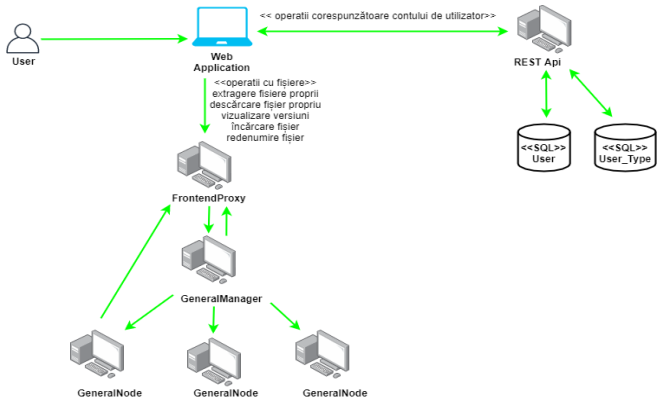

Figura 3.2: Interacțiunea unui utilizator simplu cu sistemul

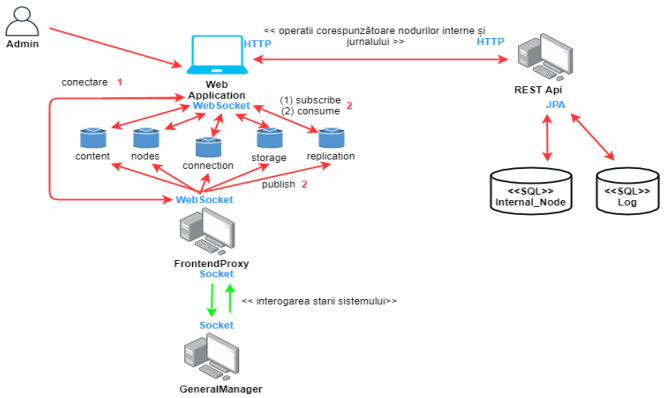

Figura 3.3: Interacțiunea unui administrator cu sistemul

În ceea ce privește comunicarea dintre nodul general și nodurile interne, aceasta se poate realiza în două moduri.

- Comunicația de tip *unicast*, utilizată cu scopul vehiculării cererilor de prelucrare a fișierelor și menținerii stării sistemului
- Comunicația de tip multicast, utilizată pentru a putea trimite un anumit mesaj de la un anumit nod la toate nodurile sistemului. Se va folosi în contextul trimiterii de mesaje de tip heartbeat de la nodurile interne către nodurile generale, pentru a comunica starea sistemului de fișiere (obiecte de tip *NodeBeat*). Totodată, se mai folosesc în contextul trimiterii cererilor de includere a sumei de control pentru fiecare fișier, cu scopul verificării integrității acestora (vezi Figura 3.4).

Comunicația de tip multicast dintre cele două aplicații se va putea realiza prin folosirea unui socket de tip *MulticastSocket,* prin intermediul clasei *HeartbeatSocket,* definită în modulul *GeneralPurposeStructures,* care va expune toate metodele necesare definirii unui socket de tip multicast și vehiculării de mesaje. La nivelul celor două aplicații, pentru fiecare instanță a aplicației, va fi instanțiat un *HeartbeatSocket.* Pentru a putea comunica în grupul de multicast, fiecare nod trebuie să se conecteze la grup, prin definirea unei adrese de tip *InetAddress,* pe o adresă de tip multicast (230.0.0.10). Fiind o comunicație de tip grup, un mesaj trimis de la un nod, va ajunge la toate nodurile din grup, inclusiv la nodul care a trimis mesajul. Pentru a evita acest lucru, nodul care dorește să trimită un mesaj va părăsi grupul (*leaveGroup*), va trimite mesajul (*send*) și se va realătura grupului (*joinGroup*) pentru a putea primi mesajele următoare, care vor fi împachetate sub forma unor obiecte de tip *DatagramPacket [30]*. Totodată, mesajele de tip heartbeat care înglobează starea sistemului de fișiere sunt de interes doar pentru nodul general. Cu toate că nodurile interne vor primi mesajele de stare și de la celelalte noduri interne, acestea vor fi ignorate. Acest lucru se va realiza prin faptul că funcția de recepție specifică nodurilor interne va aștepta doar obiecte de tip *RequestCRC*, specifice procesului de validare a integrității. Pentru orice alt tip de obiect, se va genera o excepție de tip *ClassCastException*, care va fi ignorată.

În cadrul procesului de verificare a integrității, nodul general va trimite un mesaj către toate nodurile, prin care solicită recalcularea valorii sumei de control pentru fiecare fișiere. Având în vedere că procesul de calculare a sumei de control poate fi costisitor, nu se impune includerea sumei de control pentru fiecare fișier în următorul heartbeat, ci atunci când s-a finalizat calcularea acesteia. Nu se impune finalizarea calculării sumei de control pentru toate fișierele înainte de includerea în mesaj, ci sunt incluse pe măsură ce sunt calculate. După prima

includere a sumei de control, se va trimite doar valoarea -1 până la următoarea verificare, pentru a indica nodului general că nu este necesar să actualizeze tabela de status, proces, de asemenea, costisitor. Pentru a grăbi procesul de înregistrare a unui nou fișier în sistem, nu se așteaptă până când se solicită suma de control pentru acesta, ci se trimite imediat ce fișierul a fost stocat (vezi Figura 3.4).

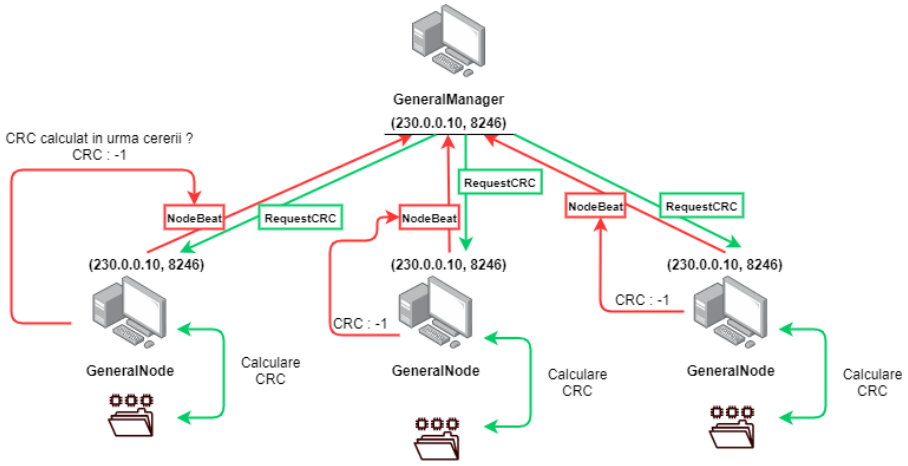

Figura 3.4: Comunicarea multicast și verificarea integrității

Operația de adăugare a unui fișier în sistem pornește de la aplicația de tip client. După ce utilizatorul a încărcat fișierul, printr-o operație de tip *drag and drop* și a specificat o descriere a versiunii fișierului, se va trimite către intermediar o cerere *HTTP* de tip *POST* în care fișierul va fi inclus în corpul mesajului, sub forma unui obiect de tip *FormData.* Se vor atașa cererii în header și token-ul utilizatorului, împreună cu cantitatea de stocare disponibilă pentru acesta, astfel încât să se decidă dacă acesta poate încărca fișierul. Această cerere va fi recepționată de către intermediarul clientului. Corpul mesajului va fi deserializat sub forma unui obiect de tip *MultipartFile* ce va expune metode pentru accesarea datelor generale despre fișier (nume, tip, dimensiune) și a conținutului acestuia. Acest conținut poate fi citit prin intermediul unui obiect de tip *BufferedInputStream*, creat pe baza unui obiect de tip *FileInputStream.* Un dezavantaj al folosirii acestor obiecte, în contextul fluxului binar conținut în *MultipartFile [31]* este faptul că nu facilitează citirea unui fișier de mai multe ori. Astfel, odată consumat fluxul de octeți printr-o operație de citire, acesta nu mai poate fi analizat încă odată. Citirea multiplă se realizează pentru a calcula suma de control a fișierului și pentru trimiterea efectivă a fișierului. Pentru a rezolva acest neajuns, se salvează fișierul la nivelul intermediarului clientului. După efectuarea cu succes a tuturor operațiilor necesare, fișierul este eliminat.

Pentru a persista fișierul în sistem, se va trimite o cerere către nodul general, cerere care va include id-ul utilizatorului, numele fișierului, dimensiunea și suma de control, sub forma unui obiect de tip *NewFileRequest.* Nodul general va identifica nodurile cele mai potrivite pentru a stoca fișierul, în funcție de cantitatea de memorie disponibilă, alegându-se cu prioritate nodurile cu cea mai puțină memorie ocupată. Se va întoarce către intermediar un token, reprezentând nodurile care vor putea stoca fișierul, organizate sub forma unui lanț. Astfel, intermediarul va trebui să trimită mai departe fișierul către nodurile interne. În același timp, nodul general va deschide o buclă de așteptare a feedback-ului de stocare cu succes a fișierului, din partea intermediarului. Nodul general va salva informațiile de stare despre fișier în tabela de conținut, însă va atașa o stare PENDING, până la primirea confirmării stocării cu succes. Această stare va indica mecanismului de replicare faptul că fișierul nu a fost încă stocat în întregime și nu trebuie replicat. Intermediarul va trimite fișierul către primul nod din lanț și un obiect *FileHeader* care va conține date despre fișier, precum și lanțul de noduri care va stoca fișierul. Fiecare nod intern

va trimite fișierul, către celelalte noduri din lanț, până la finalizarea acestuia. Pe lângă stocarea efectivă a fișierului, nodurile interne vor crea și un fișier de metadate, în care să fie salvate versiunile. Acest fișier de metadate va fi alcătuit pe baza numelui fișierului, cu adăugarea extensiei *.metadata.*

Pentru a asigura persistarea corectă a fișierului, intermediarul va deschide o buclă de feedback, unde va aștepta o confirmare de stocare cu succes din partea nodurilor interne. Această confirmare de salvare cu succes se va realiza pe baza comparației sumei de control. Astfel, după stocarea fișierului, fiecare nod intern va trebui să trimită suma de control către intermediar. În cazul în care cel puțin o sumă de control coincide, intermediarul va trimite confirmare către nodul general, precum că fișierul a fost stocat cu succes. Se impune primirea a cel puțin o confirmare întrucât, chiar dacă celelalte fișiere nu s-ar fi stocat cu succes, acestea ar putea ajunge pe aceste noduri prin mecanismul de replicare. În acest moment, nodul general va salva fișierul într-o coadă de așteptare (*PendingQueue*), păstrând starea *PENDING* a fișierului. Acesta va ieși din această coadă și va trece în starea valid abia atunci când nodurile interne vor include acest fișier în mesajele de tip heartbeat. După ce tot acest proces a fost realizat cu succes, se va trimite confirmare de înregistrare cu succes a fișierului către client (vezi Figura 3.5). Acest proces va fi folosit și în cazul actualizării unui fișier din punct de vedere a conținutului. Actualizarea presupune încărcarea unei noi versiuni a fișierului, furnizându-se noul fișier. Astfel, singura deosebire în acest proces va apărea la selectarea nodurilor interne care vor stoca fișierul. Nu se vor alege noduri noi, ci se vor folosi nodurile care deja stochează fișierul. Totuși se impune ca fișierul care conține noua versiune a fișierului să aibă același nume ca și versiunea anterioară, astfel încât să poată fi identificat la nivelul sistemului ca fiind noua versiune a unui fișier existent. În caz contrar, noua versiune va fi tratată ca un fișier nou în sistem.

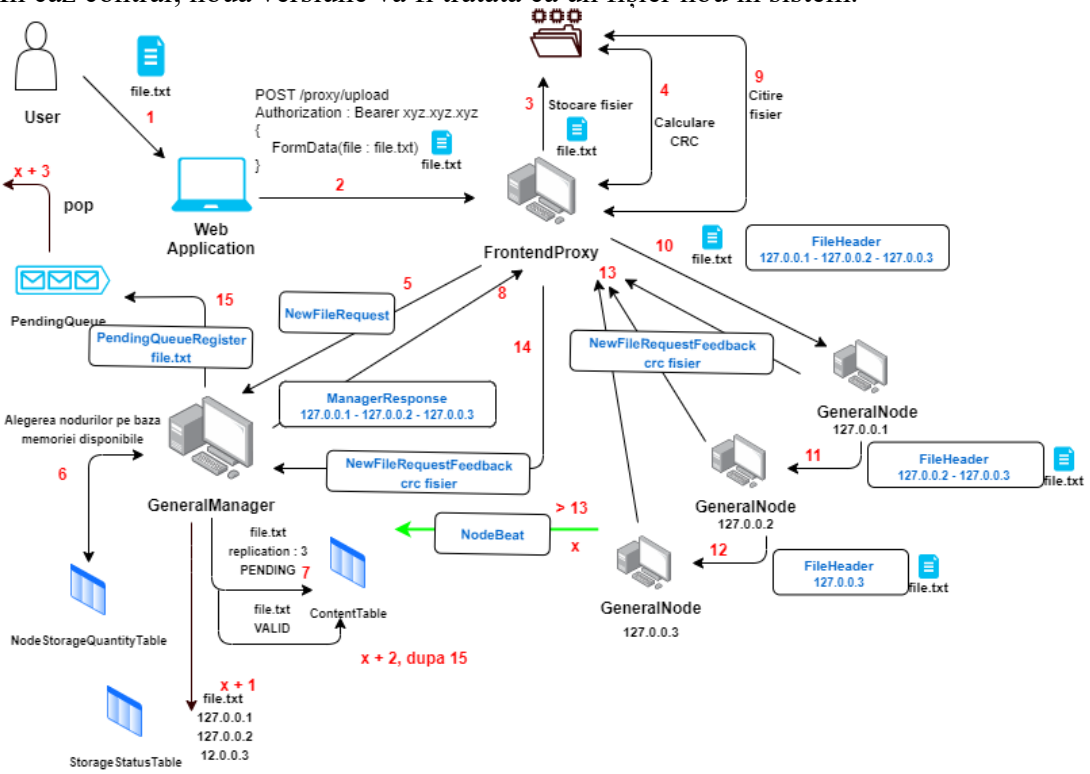

Figura 3.5: Procesul de adăugare a unui nou fișier

Operația de accesare a unui fișier de către utilizator reprezintă o operație de descărcare. Fișierul va trebui descărcat din sistem, de pe unul dintre nodurile interne și salvat la client. Acest

proces începe cu o cerere efectuată de client, prin simpla accesare a unui fișier din lista de fișiere disponibile. Se va efectua o cerere HTTP, de tip *GET /proxy/<filename.txt>* către intermediarul clientului. La primirea cererii, acesta va crea un obiect de tip java bean (*GetNodeForDownload*) în care va atașa identificatorul unic al utilizatorului și numele fișierului. Având în vedere că fișierul se află pe unul dintre nodurile interne, iar intermediarul nu cunoaște care dintre noduri stochează acest fișier, va trimite o cerere către nodul general, prin care va solicita adresa unui nod care va stoca acest fișier. Nodul general va răspunde cu un obiect de tip *ManagerTextResponse,* care va conține adresa primului nod găsit care stochează o replică a fișierului. După ce primește adresa nodului, intermediarul va formula o cerere către nodul intern, prin care va solicita primirea fișierului. Pentru aceasta, va trimite o cerere sub forma unui obiect *DownloadFileRequest*, în care va include identificatorul utilizatorului și numele fișierului, apoi va deschide o buclă de citire a fluxului binar trimis de către client, folosind un *DataInputStream*. În acest timp, nodul intern va primi cererea, va verifica existența fișierului și va începe trimiterea fișierului către client. Intermediarul va citi fluxul binar de octeți și va salva fiecare pachet de date în memoria de stocare, în fișier, folosind *FileOutputStream*. După finalizarea stocării, va răspunde către aplicația client cu calea unde a fost stocat fișierul. Aplicația de tip client va accesa fișierul, va oferi un previzualizare către utilizator și va oferi posibilitatea descărcării efective a fișierului pe mașina clientului.

O limitare care a intervenit în acest context este faptul că, pe măsură ce primește conținutul binar al fișierului, intermediarul nu poate trimite fișierul către aplicația de tip client, având în vedere că vor comunica prin cereri HTTP. Pentru a putea trimite fișierul, acesta va trebui citit integral. În cazul în care fișierele sunt de dimensiuni mari iar solicitările vin de la mai mulți clienți, salvarea fișierului în memoria internă de program nu este eficientă, motiv pentru care se preferă stocarea fișierului în memoria intermediarului. Pentru a evita trimiterea unei cereri HTTP care să conțină fișierul, s-a ales ca aplicația de tip client și aplicația de tip intermediar al clientului să ruleze pe aceeași mașină, pentru a partaja memoria de stocare. Astfel, fișierul stocat de intermediar va putea fi citit direct din memorie de către aplicația client. Această soluție împiedică scalarea aplicației de tip intermediar, însă oferă un timp de răspuns mai rapid.

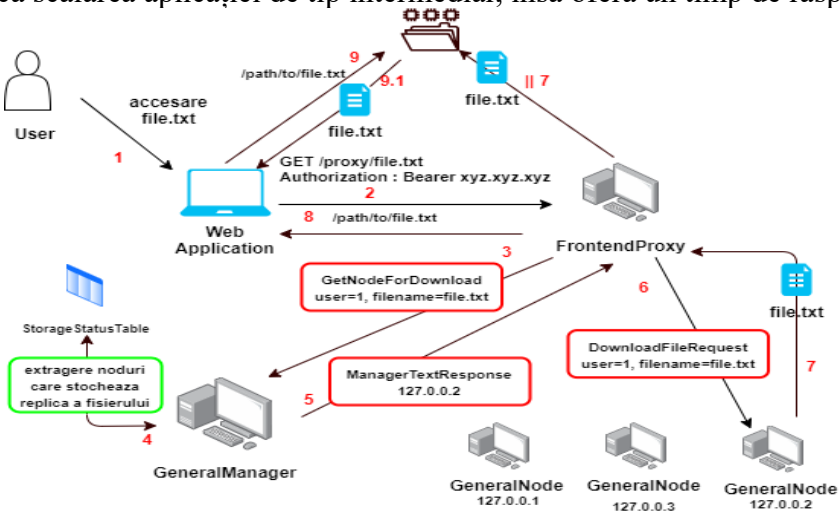

Figura 3.6: Procesul de descărcare a unui fișier

În cazul operațiilor de redenumire și eliminare a unui fișier, procesul este asemănător. Din aplicația de tip web se va trimite o cerere către aplicația intermediar. În cazul redenumirii, cererea va fi *PUT /proxy/<filename.txt>,* noul nume al fișierulu fiind plasat în corpul mesajului, iar în cazul ștergerii unui fișier, cererea va fi *DELETE /proxy/<filename.txt>*. La primirea acestei

cereri, nodul intermediar va crea obiectele serializabile corespunzătoare *RenameFileRequest* și *DeleteFileRequest,* care vor conține date despre identificarea utilizatorului, a fișierului și detalii despre versiune. Aceste obiecte vor fi trimise către nodul general care va identifica nodurile interne care conțin fișierul respectiv și va trimite către fiecare dintre acestea, cererea de actualizare sau eliminare, sub forma unor obiecte *RenameRequest* și *DeleteRequest*. În cazul operației de eliminare, nu se va trimite o cerere directă de la modulul care gestionează relația cu clientul, ci se va actualiza factorul de replicare al fișierului cu valoarea 0, ceea ce va genera eliminarea replicilor de pe toate nodurile. Nodul general va aștepta un răspuns de la aceste noduri interne (*FeedbackTextResponse*), pentru efectuarea cu succes a cererii. Și în acest caz, se impune primirea a cel puțin un răspuns favorabil astfel încât, dacă celelalte operații nu s-au efectuat cu succes, se va genera replicarea pe baza replicii valide.

În ceea ce privește nodurile interne, trebuie asigurate mecanismele necesare astfel încât, dacă unul dintre nodurile care persistau fișierul, a fost oprit, la repornire să poată vedea modificarea efectuată. Se va folosi elementul de stare a fiecărui fișier, din tabela conținut. În cazul operației de eliminare a unui fișier, starea acestuia va fi trecută în *DELETED* astfel încât, dacă un nod care va fi repornit va conține o replică, aceasta va fi eliminată și de pe noul nod. În aceeași măsură, la redenumirea unui fișier, starea va fi trecută în *DELETED* și se va adăuga o nouă înregistrare cu noul nume al fișierului, în starea *VALID*. Se va aplica același proces la repornirea unui nod, replica respectivă fiind eliminată. Se va evita redenumirea fișierului conform noii versiuni întrucât se va genera o supra replicare a fișierului, ceea ce va conduce la eliminare (vezi Figura 3.8 și Figura 3.7).

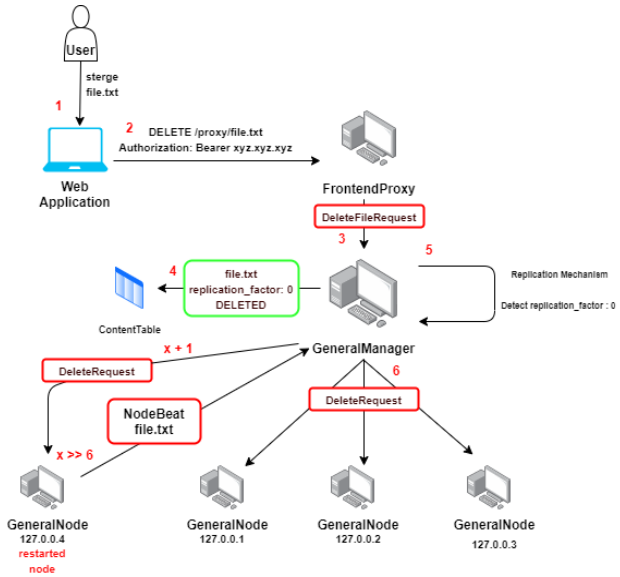

Figura 3.7: Procesul eliminării unui fișier

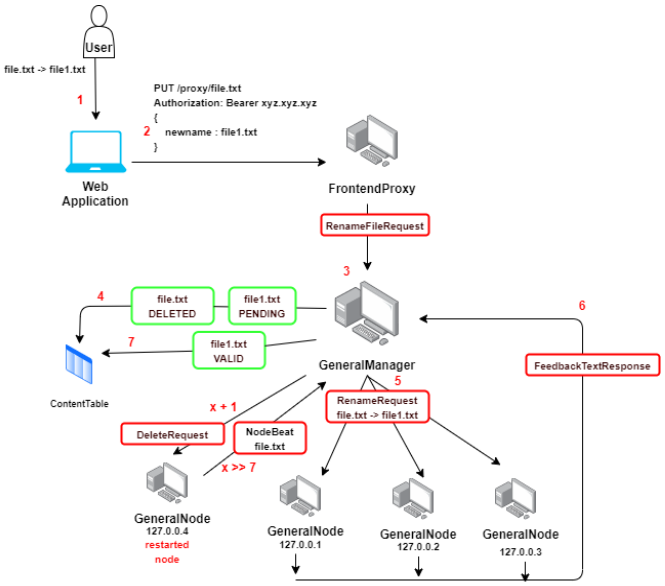

Figura 3.8: Procesul redenumirii unui fișier

În cazul procesului de replicare, nodul general va conține o clasă numită *ReplicationManager*, care va expune toate funcționalitățile necesare asigurării disponibilității sistemului. La intervale regulate de timp, se vor compara cele două tabele de stare ale nodului general (*StorageStatusTable* și *ContentTable*). În cazul unor inconsistențe, se va genera procesul de replicare sau eliminare a unei replici. Aceste inconsistențe sunt determinate de următorii factori.

- oprirea (va genera replicare) sau repornirea unui nod care a fost temporar oprit (eliminare)

- coruperi ale replicilor din punct de vedere al sumei de control sau a versiunii (va genera eliminarea, apoi replicarea)

În procesul de verificare a celor două tabele, se va ține cont de stările intermediare ale fișierelor (*VALID*, *PENDING*, *DELETED*). Astfel, dacă un fișier se află în starea *PENDING,* din cauza unui proces de replicare ce se execută deja sau fișierul este nou în sistem, însă nu a fost încă înregistrat complet, nu se va include în procesul de replicare, până când nu va trece în starea *VALID*. În schimb, dacă fișierul se află în starea *DELETED,* fiind eliminat sau redenumit, se va putea genera doar eliminarea unei replici de pe un nod care a fost oprit temporar.

Toate aceste operații vor genera anumite evenimente la nivelul nodurilor de care sunt executate. Pentru a putea asigura o monitorizare eficientă a sistemului, toate aceste evenimente vor fi jurnalizate în baza de date relațională, gestionată de către serverul cu interfață REST. La fiecare eveniment, cu ajutorul modulului HTTPHandler (clasa *LoggerService*), se va realiza o cerere de tip HTTP *POST /api/log,* care va conține, în corpul cererii, detalii despre nodul la nivelul căruia a fost generat evenimentul, tipul (SUCCESS, WARNING, ERROR) și o descriere a evenimentului. Serverul cu interfață REST va primi cererea, va extrage toate datele necesare, va adăuga datele temporale și, cu ajutorul interfeței *JPA,* va instanția obiectul de tip *Log,*adnotat corespunzător pentru a putea fi stocat în baza de date. Solicitarea de stocare în baza de date va fi preluată de obiectul de tip *EntityManager*. Se va genera o tranzacție (*getTransaction*), în cadrul căreia obiectul va fi persistat în baza de date (*persist* și *commit*). În cazul în care tranzacția a eșuat, se va genera o excepție și se va putea restaura starea anterioară a bazei de date (*rollback*). Evenimentele vor putea fi preluate de către aplicația de tip client, astfel încât să faciliteze monitorizarea sistemului de către administrator. Acesta va fi singurul care va putea curăța jurnalul de evenimente. (vezi Figura 3.9)

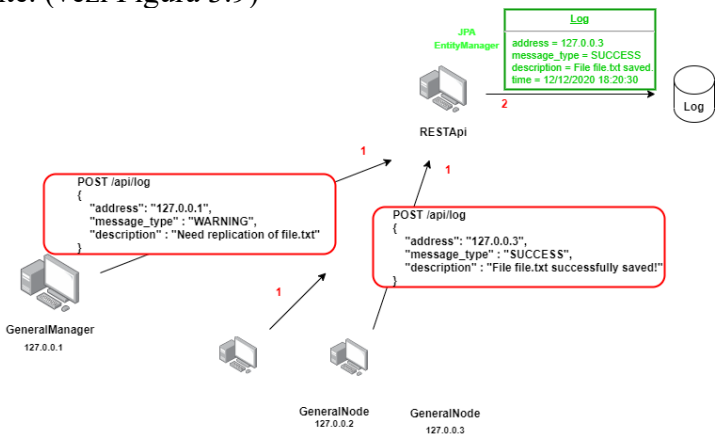

Figura 3.9: Procesul de jurnalizare

3.3. Efectuarea comunicării

Aplicația prezintă un sistem de stocare de fișiere, proiectat prin intermediul unui sistem distribuit. În cadrul unui astfel de sistem, comunicarea dintre nodurile sistemului se realizează prin intermediul rețelei. Astfel, în sistem sunt prezente trei modalități principale de comunicare în rețea.

- Comunicarea prin cereri HTTP, care are ca principal obiectiv interacțiunea cu clientul. Sunt vehiculate cereri HTTP între:
  - aplicația client și aplicația intermediar al clientului .
  - aplicația client și serverul cu interfață REST, care expune funcționalitățile necesare reprezentării obiectelor și persistării acestora în baza de date.

- Serverul cu interfață REST și toate nodurile sistemului, în vederea jurnalizării tuturor evenimentelor apărute în sistem.

În cadrul componentelor sistemului care primesc cereri de tip HTTP și sunt implementate în Java, s-a folosit SpringBoot, care va facilita interpretarea acestor cereri (prin intermediul adnotărilor sau fișierelor de configurare), atașarea funcțiilor de prelucrare a tuturor cererilor și crearea răspunsurilor corespunzătoare fiecărei cereri.

În ceea ce privește interacțiunea dintre componentele interne ale sistemului, aceasta se realizează prin două tipuri de comunicare:

- comunicația de tip unicast, realizată în vederea prelucrării fișierelor și interogării stării sistemului. În acest context se vor deschide conexiuni *TCP* de tip client (*Socket*) – server (*ServerSocket*), în cadrul cărora se vor vehicula obiecte de tip java beans.
- Comunicația de tip multicast, realizată în vederea menținerii stării sistemului, atât prin trimiterea de mesaje de tip heartbeat de la nodurile interne către nodul general, cât și prin trimiterea de cereri de calculare a sumei de control, pentru verificarea integrității. În acest context, se va deschide comunicare *UDP* prin deschiderea unor componente socket de tip *MulticastSocket,* definirea grupului de multicast, folosind o adresă specifică de multicast InetAddress (237.0.0.10) și prin vehicularea de pachete de date specifice UDP *DatagramPacket.*

Un ultim tip de comunicație prezent în sistem este reprezentat de comunicarea dintre serverul cu interfață de tip REST și serverul de baze de date. Prin intermediul *JPA,*se va crea o instanță de tip *EntityManager,* care va reprezenta conexiunea cu baza de date. Acest obiect va pune la dispoziție toate metodele necesare pentru a efectua operații la nivelul bazei de date. Pentru a eficientiza procesul de persistare a datelor și a evita blocarea conexiunii generate prin încercarea prelucrării simultane a datelor din mai multe tabele ale bazei de date, se vor defini mai multe astfel de obiecte, corespunzător fiecărei tabele, prin intermediul unui *EntityManagerFactory*. Astfel, se va evita blocarea efectuării operațiilor asupra unei tabele, din cauza unei tabele prelucrate mai frecvent (în cazul sistemului, tabela *Log*). Totodată, pentru a realiza sincronizarea efectuării de operații la nivel de tabelă, fiecare tranzacție va fi precedată de o sincronizare explicită a obiectului *EntityManager*, folosind cuvântul cheie *synchronized*.

3.4. Salvarea/stocarea informații

Sistemul prelucrează două tipuri de date. În primul rând, sunt prelucrate fișierele utilizatorilor. Acestea sunt stocate la nivelul nodurilor interne, în memoria de stocare de lungă durată. Pe lângă fișierele propriu-zise ale utilizatorilor, sunt stocate și metadate care să cuprindă informații despre versiunile fișierelor. Astfel, pentru fiecare fișiere existent în sistem, este creat un fișier de metadate, pe baza numelui fișierului, prin adăugarea extensiei *.metadata*. Crearea și scrierea acestor fișiere se face prin intermediul obiectului specific Java *FileOutputStream.* La primirea cererii de stocare a unui fișier la nivelul nodului intern (*FileHeader*) se va deschide fluxul de ieșire, se va crea fișierul și, pe măsură ce pachetele de octeți de dimensiune fixă, trimise în cadrul conexiunii TCP de la intermediar către nodul intern, sunt recepționate, se scriu în fișier, prin folosirea funcției *write*. După finalizarea scrierii fișierelor, acestea sunt accesate periodic, atât pentru operații solicitate de client, cât și pentru verificarea integrității.

În al doilea rând, sunt prelucrate date de identificare a unor entități din cadrul sistemului. Aceste date vor fi reprezentate sub formă de obiecte și vor fi persistate într-o bază de date relațională. Această persistare va fi posibilă datorită mapării obiectual-relațional, facilitată de către JPA. În acest sens, fiecare obiect va fi creat pe baza descrierii unei clase care va conține

toate atributele necesare obiectelor, împreună cu o serie de adnotări, prin care se va realiza maparea la relațional. Fiecare clasă va fi precedată de adnotarea *@Entity* pentru a sugera că, la nivelul modelului relațional, va reprezenta o entitate. Apoi, se va specifica numele entității, prin adnotarea *@Table.* Având în vedere că fiecare atribut la nivelul unei clase este echivalent cu un antribut la nivelul unei tabele, identificat prin numele coloanei, fiecare atribut din clasă va fi precedat de numele coloanei specifice tabelei din modelul relațional echivalent (*@Column*). Entitățile modelului relațional pot avea chei unice de identificare, numite chei primare. La nivelul clasei, acestea se vor specifica prin adnotarea *@Id*. Mai mult decât atât, aceste chei primare pot fi auto-generate, ceea ce, la nivelul clasei, se traduce prin adăugarea adnotării @GeneratedValue și specificarea unei strategii de generare (*GenerationType.Identity*) [32]. Pe baza acestei definiri și prin construirea metodelor corespunzătoare operațiilor CRUD, datele vor putea fi stocate în baza de date. Clasele corespunzătoare entităților sunt (vezi Figura 3.10):

- *User*, ce definește utilizatorul, prin intermediul datelor de contact, a tipului acestuia și a informațiilor referitoare la volumul de date și numărul de fișiere stocate.
- *User_Type,* ce definește tipul de utilizator disponibil, prin intermediul facilităților puse la dispoziție acestuia (factorul de replicare, cantitatea totală de memorie disponibilă și prețul ce trebuie plătit pentru actualizarea tipului de utilizator).
- *Internal_Node*, ce definește un nod al sistemului, prin intermediul datelor de identificare (adresa IP) și de stare (status ON sau OFF).
- *Log*, ce definește un eveniment apărut în sistem, prin intermediul datelor de identificare, a sursei, tipului și descrierii evenimentului, precum și a indicilor temporali.

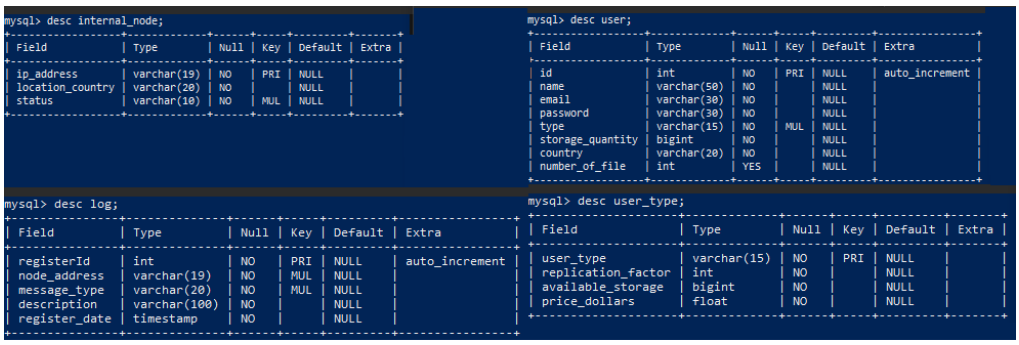

Figura 3.10: Tabelele din cadrul modelului relațional

3.5. Interfața cu utilizatorul

Interfața cu utilizatorul, expusă în cadrul aplicației client de tip web, a fost implementată folosind framework-ul *React* și limbajul *Javascript*. În alcătuirea interfeței utilizator s-au folosit elemente standard HTML (<*div*> pentru definirea unei noi secțiuni, <*p*> pentru definirea unui paragraf, <*table*> pentru a defini un tabel, <*a*> pentru a defini o ancoră, <*img*> pentru a defini o imagine, <*button*> pentru a defini un buton, <*select*> pentru crearea unei liste, etc), care au fost extinse cu funcții specifice React (*onClick*, *onMouseOver*, *onMouseLeft, etc.).* Una dintre caracteristicile React folosite în cadrul interfeței cu utilizatorul este caracterul dinamic. Se pot modifica sau redefini anumite elemente din interfață, pe măsură ce elementele de stare sunt actualizate. De exemplu, în cazul definirii unui tabel, se pot adăugă noi linii pe măsură ce datele au fost actualizate și informația de starea a fost modificată corespunzător. Aceste elemente pot fi salvate în cadrul unor obiecte specifice Javascript (o listă sau un dicționar), iar aceste structuri să fie referențiate în mod direct, în funcția de randare a interfeței grafice. (vezi Anexa 15.) În aceeași măsură, dacă a fost definit un element corespunzător afișării datelor despre un fișier, însă

acestea nu au putut fi extrase corespunzător, se poate înlocui acest element, cu unul de tip paragraf, care să conțină un mesaje de informare despre eveniment. Această înlocuire se poate face în mod efectiv, prin redefinirea elementului HTML, sau prin folosirea unei structuri condiționale (operatorul ternar „*?:*”) (vezi Anexa 14.)

Sistemul are două tipuri de utilizatori, administratorul sistemului și utilizatorul obișnuit. Pentru fiecare dintre aceștia, se va expune o interfață utilizator diferită. Administratorul sistemului va avea la dispoziție o fereastră de monitorizare în care va putea analiza și prelucra starea sistemului în timp real. Acesta va putea vizualiza și curăța jurnalul de evenimente, va putea solicita eliminarea unor fișiere de la anumite noduri pentru a verifica funcționarea corectă a mecanismului de replicare, va putea vizualiza nodurile interne ale sistemului, împreună cu fișierele disponibile în memoria acestora, va putea analiza activitatea mecanismului de verificarea a integrității sistemului. În schimb, utilizatorul obișnuit va avea acces la contul de utilizator și la propriile fișiere. În cazul contului de utilizator, vor fi disponibile datele de contact ale utilizatorului și date despre capacitatea de stocare disponibilă. Totodată, se va pune la dispoziția utilizatorului și gama de tipuri de utilizatori, împreună cu beneficiile aferente și prețul abonamentului. Utilizatorul va putea actualiza tipul de cont. În cazul fișierelor, utilizatorul va putea adăuga un nou fișier, printr-o operație *drag and drop*, urmată de completarea detaliilor despre versiune. Totodată, utilizatorul va putea vizualiza lista de fișiere. La accesarea unui fișier, se va încărca o previzualizare a fișierului, însă se va oferi și posibilitatea descărcării fișierului. De asemenea, se vor încărca și datele despre versiunile fișierului. Din punct de vedere a operațiilor disponibile asupra fișierului, se vor putea efectua operații de eliminare, redenumire și modificare. Operația de modificare va redirecta utilizatorul spre pagina de încărcare a fișierului, unde se va încărca noua versiune a fișierului, ce a fost actualizată local.

Capitolul 4. Testarea aplicației și rezultate experimentale

4.1. Lansarea aplicației și elemente de configurare

Sistemul este alcătuit din mai multe module, care pot fi împărțite în două categorii: module generale și module specifice. Modulele generale expun anumite funcționalități care sunt necesare în cadrul mai multor module. Aceste module nu conțin un comportament propriu și nu vor rula ca un proces de sine stătător. În schimb, modulele specifice sunt considerate aplicații de sine stătătoare, ce rulează în cadrul unui proces, deoarece expun o funcționalitate specifică a sistemului și au un comportament propriu. Fiind un sistem distribuit, se impune ca aceste componente să poată fi încărcate și rulate independent, pe mașini diferite. Având în vedere că aceste componente folosesc funcționalități expuse de modulele generale, trebuie ca aceste dependințe să fie încărcate împreună cu modulul specific componentei în cadrul aceluiași fișier executabil. Astfel, modulele interne ale sistemului, fiind implementate în limbajul Java, vor fi împachetate în cadrul unor arhive specifice limbajului, numite JAR[15]. În cazul componentelor executabile, se vor include și modulele generale folosite. Componentele dezvoltate cu ajutorul SpringBoot vor fi împachetate cu ajutorul sistemului de împachetare Maven. În acest caz, în cadrul arhivei va fi inclusă atât aplicația de sine stătătoare și o instanță a unui server de tip *Apache Tomcat*, cât și toate dependințele necesare. Pentru a încărca și dependințele, va fi necesară o configurație specială efectuată în fișierul de configurare specific Maven, *pom.xml*. În urma împachetării va rezulta un fișier cu extensia *.jar,* care, în cazul componentelor dezvoltate cu ajutor SpringBoot, va conține în cadrul numelui și expresia *jar-with-dependencies [33]*. Pentru a putea fi rulate, se va folosi comanda *java -jar <executabil>.jar* .

Fiecare componentă va conține o serie de parametri de configurare. Acești parametri pot fi modificați în următoarele circumstanțe:

- Modificarea mediului de rulare a aplicației implică modificarea parametrilor de tip cale în sistemul de fișiere. Calea este specifică sistemului de operare.
- Schimbarea performanțelor sistemelor de calcul și a canalelor de comunicație implică ajustarea dimensiunii pachetului de date trimis prin canalul de comunicație.
- Schimbarea adreselor IP și a porturilor componentelor de tip server și client.
- Schimbarea căilor de tip URI, expuse în cadrul API-ului de tip REST.
- Dorința de modificare a frecvenței anumitor mecanisme din cadrul sistemului.
    - Frecvența de trimitere a mesajelor de tip *heartbeat* de la nodurile interne la nodul general.
    - Frecvența de solicitare a sumei de control a fișierelor, pentru a verifica integritatea acestora.
    - Frecvența de verificare a tabelei de conexiuni dintre nodul general și nodurile interne.
    - Frecvența de execuție a mecanismului de verificare a stării sistemului, în vederea menținerii factorului de replicare constant.

Acești parametri vor fi definiți la nivelul fiecărei componente, într-un fișier de configurare, având extensia *.config* și vor fi încărcați în program la inițializare. Astfel, acest fișiere de configurare trebuie să fie plasat în aceeași locație cu fișierul executabil Java de tip JAR. În cazul componentelor dezvoltate cu SpringBoot, se vor inițializa anumiți parametri specifici în fișierul *application.properties*, din directorul de resurse al aplicației. Însă, după împachetarea aplicației, eventualele modificări ale parametrilor din acest fișier nu vor fi luate în considerare, decât la următoarea împachetare.

15 Java ARchive

4.2. Testarea software a sistemului

4.3. Încărcarea procesorului și a memoriei

Operațiile cele mai costisitoare din cadrul sistemului, din punct de vedere a încărcării procesorului și a memoriei sunt, operațiile de prelucrare a fișierelor și de menținere a stării sistemului. Din punct de vedere al prelucrării fișierelor, cea mai costisitoare operație este întâlnită la nivelul nodurilor generale, în procesul de calculare a sumei de control a fișierelor. În acest caz, încărcarea crește direct proporțional cu numărul de fișiere și cu dimensiunea acestora. În urma unor măsurători experimentale (vezi Figura 1.4) s-a constatat faptul că timpul necesar calculării sumei de control a unui fișier cu o dimensiune de 1,17 GB este de aproximativ 600 milisecunde. O analiză similară poate fi observată și în Figura 4.1. Se observă că timpul necesar calculării sumei de control crește direct proporțional cu dimensiunea fișierului. Mecanismul de calculare a sumelor de control presupune că, după finalizarea calculului, suma de control poate fi trimisă către nodul general. Fiecare calcul va fi realizat pe un fir de execuție separat astfel încât sumele de control ale fișierelor de dimensiuni mici să se calculeze rapid și să fie trimise imediat către nodul general. Problema rămâne în cazul fișierelor de dimensiuni mari. Din acest motiv, una dintre limitările sistemului introduse explicit în versiunea actuală este reprezentată de faptul că nu este permisă stocarea unor fișiere având dimensiuni mai mari de 100 MB, pentru a menține un timp de calculare a sumei de control relativ mic. Tot din acest motiv, frecvența cu care se verifică integritatea sistemului este foarte mică astfel încât, perioada de timp dintre două astfel de cereri să fie mult mai mare decât perioada de timp necesară calculării și furnizării tuturor sumelor de control.

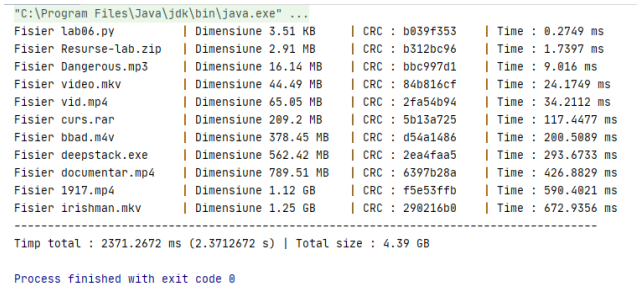

Figura 4.1: Metrici referitoare la timpul necesar calculării sumei de control

Un alt argument pentru limitarea impusă despre care s-a discutat anterior este reprezentat de încărcarea indusă la nivelul memoriei. Pentru a putea trimite un fișier, acesta trebuie încărcat în memoria internă a aplicației de tip client. Din acest motiv, fișierele de dimensiuni foarte mari induc supraîncărcarea memoriei clientului și determină o interacțiune greoaie cu aplicația. Astfel, s-a constatat că o dimensiune a fișierului de maxim 100 MB nu induce încărcarea memoriei și afectarea experienței utilizatorului în interacțiunea cu aplicația. Nodul general trebuie să gestioneze întreaga stare a sistemului. Acest proces se va realiza prin intermediul tabelelor de stare, care vor conține informații atât despre nodurile interne, împreună cu fișierele stocate la nivelul acestora, cât și despre fișierele ce trebuie să existe în sistem. Ținând cont de faptul că toate aceste tabele sunt stocate în memoria programului, încărcarea sistemului va crește direct proporțional cu numărul de fișiere din sistem și cu complexitatea metadatelor acestora. Încărcarea poate fi considerată atât din punct de vedere a cantității de memorie de program ocupată, cât și din punct de vedere a timpului necesar accesări datelor (citire și scriere), considerând că starea sistemului este actualizată la intervale regulate de timp. În Figura 4.2 se poate un exemplu al cantității de memorie ocupate de tabelele de stare la un moment dat. În alcătuirea exemplului, s-au considerat 4 noduri interne active și 14 fișiere înregistrate în sistem.

Se poate observa că tabela de conexiuni și cea a cantităților de stocare nu au dimensiuni foarte mari deoarece conțin doar adresele nodurilor interne și cantitățile de stocare ale acestora. În schimb, tabela de conținut și cea de status a stocării au dimensiuni mari deoarece conțin informații despre fișiere (nume, suma de control, versiune curentă, dimensiune). Observăm că tabela de status a stocării ocupă cea mai multă memorie deoarece conține maparea dintre un fișier și replicile acestuia (nodurile care stochează fișierele, împreună cu metadatele acestora).

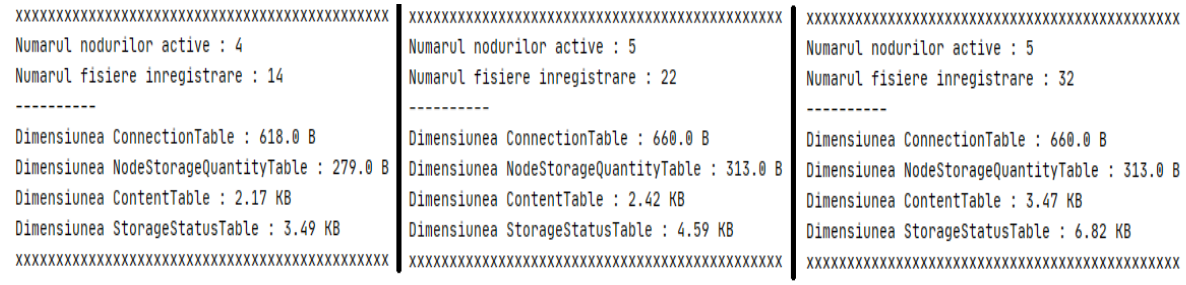

Figura 4.2: Cantitatea de memorie ocupată de tabelele de stare ale nodului general

4.4. Limitări în ceea ce privește transmisia datelor

Comunicarea dintre componentele interne ale sistemului se realizează prin intermediul conexiunilor TCP, în cadrul cărora sunt vehiculate obiecte de tip java beans. Aceste cereri au rolul de a solicita efectuarea anumitor operații. Unele dintre aceste operații, în special cele prin care se solicită starea sistemului, presupun transmiterea întregilor tabele de stare către client. Tabelele de stare vor fi alcătuite pe baza fișierelor din sistem și a mesajelor de tip heartbeat primite de la nodurile interne. Pe măsură ce încărcarea sistemului crește odată cu numărul de fișiere ce trebuie gestionat, dimensiunea acestor tabele crește și, implicit și a obiectelor ce vor fi trimise către client. Una din limitările sistemului în ceea ce privește transmisia datelor este reprezentată de modul în care sunt recepționate aceste obiecte. Această recepție se realizează în cadrul unei singure iterații de citire a datelor transmise, fiind citit un singur pachet de date a cărui dimensiune este egală cu valoarea parametrului definit în acest sens. Astfel, dacă dimensiunea unui astfel de obiect depășește dimensiunea unui pachet, nu se vor putea citi toate datele iar obiectul nu va putea fi deserializat. Această limitare va fi eliminată în versiunile viitoare ale aplicației. Principala modalitate de rezolvare a acestei limitări este reprezentată de creșterea numărului de iterații de citire a răspunsului cererii. Conform rezultatelor obținute în Figura 4.2 se poate observa că, în medie, tabela de status al stocării atinge o dimensiune de 4KB la înregistrarea a 20 de fișiere. Astfel, dacă se consideră o dimensiune a pachetelor de 16KB, se pot înregistra aproximativ 80 de fișiere.

4.5. Fiabilitate

Sistemul distribuit de stocare de fișiere are o arhitectură de tip master/slave în care nodul master (nodul manager general) are rolul de a asigura coordonarea nodurilor slave (nodurilor interne) și expunerea tuturor operațiilor existente, către client. Astfel, fiabilitatea sistemului este asigurată cât timp nodul general este activ. În caz contrar, nu se mai asigură coordonarea nodurilor interne iar interacțiunea clientului cu sistemul devine imposibilă. Totodată, această interacțiune dintre client și sistem este mediată de către aplicația de tip intermediar, care interceptează cererile HTTP ale clientului și efectuează cererile corespunzătoare către sistem,

prin intermediul obiectelor interne ale limbajului. acestui intermediar va face imposibilă comunicarea dintre client și sistem. Inactivitatea acestor două componente reprezintă singurul factor care afectează fiabilitatea sistemului.

Fiind un sistem de stocare de fișiere, numărul de cereri și interacțiuni crește direct proporțional cu numărul de utilizatori, numărul de fișiere stocate și numărul nodurilor interne active. Pentru a menține fiabilitatea sistemului, sunt asigurate mecanisme de calcul paralel și de sincronizare asupra obiectelor accesate concurent în cadrul mai multor operații simultane. Astfel, sunt evitate blocajele inutile ale resurselor aplicației. În același timp, se asigură separarea responsabilităților în cadrul modulelor, prin faptul că fiecare obiect este responsabil de o anumită sarcină. În cadrul modulului general, obiectul care va gestiona interacțiunea nu clientul nu va gestiona și mecanismele de comunicare cu nodurile interne sau de verificare a stării integrității sistemului. Fiecare cerere va fi delegată către un alt obiect, care va realiza operația solicitată pe un alt fir de execuție, independent de activitatea obiectului ce solicită operație. În această manieră sunt evitate blocajele inutile ale anumitor procese. Totodată, se asigură faptul că procesele costisitoare, care presupun și blocarea anumitor obiecte de lucru, sunt efectuate cu o frecvență relativ mică. Mecanismul de verificare a integrității fișierelor presupune calcularea sumei de control, un proces a cărui durată crește direct proporțional cu dimensiunea fișierului. Din acest motiv, acest mecanism va fi executat la intervale regulate de timp mari. În acest caz, fiabilitatea poate fi îmbunătățită și prin creșterea numărului de noduri interne. Cu cât un nod intern gestionează mai multe replici, cu atât procesul calculării sumei de control ocupă mai mult timp din totalul alocat pentru întregul proces al nodului intern. Astfel, prin creșterea numărului de noduri interne, numărul de replici stocate de fiecare nod s-ar reduce, având în vedere mecanismul de distribuire uniformă a capacităților de stocare, iar timpul total necesar unui nod intern pentru calcularea sumei de control se reduce.

Astfel, pentru a asigura și a menține fiabilitatea sistemului, trebuie adăugate componente de tip mirror pentru componentele de tip *single point of failure*. Totodată, se pot adăuga mai multe noduri interne, pentru a distribui uniform sarcina stocării și gestionării fișierelor.

4.6. Securitate

Unul dintre aspectele cele mai vulnerabile în cadrul unui astfel de sistem este reprezentat de comunicarea cu clientul și modalitatea de reprezentare a datelor sensibile ale acestora. În cadrul acestui sistem, securizarea interacțiunii cu clientul este asigurată prin folosirea standardului *JWT*, care va reprezenta identitatea utilizatorului într-un mod sigur. Acest token va fi alcătuit din trei componente dispuse sub forma structurii *header.payload.signature*.

- *Header:* va conține date despre *token* (algoritmul de semnare).
- *Payload*: va conține date despre cerere și solicitantul acesteia
- *Signature*: semnătura digitală utilizată de server și de client pentru a verifica autenticitatea cererii.

Acest mecanism asigură securitatea prin intermediul semnăturii digitale, realizate pe baza unei chei private cunoscute doar de către client și server. Astfel, prin intermediul semnăturii digitale se va verifica dacă cererea este autentică, dacă a fost trimisă de la una dintre cele două părți ale comunicării (client și server), ci nu de la un atacator [19]. În cadrul sistemului, fiecare cerere dintre client și server va avea atașat acest *token*, în header-ul de autorizare, folosind schema de autorizare *Bearer [34]*. Se vor include date precum identificatorul unic al utilizatorului și rolul acestuia, necesar în procesul de autorizare a clientului care solicită efectuarea unei anumite operații. Un utilizator obișnuit al sistemului nu va avea putea efectua operațiile caracteristice unui administrator, de monitorizare a sistemului.

4.7. Rezultate experimentale

4.7.1. Încărcarea unui fișier

Atunci când utilizatorul dorește să încarce un fișier, se trimite o cerere de la aplicația client cu datele despre fișier (nume, dimensiune, factorul de replicare) sub forma unui obiect *FileHeader* către nodul general. Acesta, după ce verifică statusul stocării, alcătuit pe baza mesajelor de tip *heartbeat* de la nodurile interne, identifică și returnează o listă sub forma unui token, ce conține adresele nodurilor care pot stoca fișierul. (vezi Figura 4.3). Apoi nodul general va înregistra noul fișier cu starea *PENDING (*vezi Figura 4.4) până când va primi confirmare de la client despre stocarea cu succes la cel puțin unul dintre nodurile interne. În tot acest timp, mecanismul de verificare de replicare va ignora fișierul (vezi Figura 4.5). După primirea acestei confirmări, se vor aștepta și mesajele *heartbeat* de la nodurile interne, care vor include și noul fișier, moment în care fișierul va fi trecut în starea *VALID* (vezi Figura 4.6).

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

Generam token-ul..
User uploaded a new file with size : 1107523 and replication factor : 3
====================================
127.0.0.2-127.0.0.3-127.0.0.4
====================================
Token-ul a fost trimis catre client : 127.0.0.2-127.0.0.3-127.0.0.4
Inregistram noul fisier.
User not found!
Cerinta clientului a fost realizata..
Feedback nou de la frontend!
Am primit feedback si il adaugam in lista!
Feedback valid de la frontend! Confirmam stocarea noului fisier.
```

Figura 4.3: Răspunsul nodului general la cererea de încărcare a unui fișier

```
-------------------------------------
Content Table
A_Survey_on_Distributed_File_System_Technology.pdf [repl. : 3] [CRC : bb79358 | VersionNo : v1] [Size : 1107523] [Status : [PENDING]]
-------------------------------------
```

Figura 4.4: Starea PENDING a fisierului incarcat

```
-------------------------------------
Replication Status
     User 3 | File : A_Survey_on_Distributed_File_System_Technology.pdf  -->  [UNKNOWN]

-------------------------------------
```

Figura 4.5: Raspunsul mecanismului de replicare în cazul unui fișier în starea PENDING

```
-----------------------------------
Storage Status Table
User id  : 3
    Filename : A_Survey_on_Distributed_File_System_Technology.pdf
        127.0.0.2  [CRC : bb79358 | VersionNo : v1]
        127.0.0.4  [CRC : bb79358 | VersionNo : v1]
        127.0.0.3  [CRC : bb79358 | VersionNo : v1]

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

-----------------------------------
Content Table
A_Survey_on_Distributed_File_System_Technology.pdf [repl. : 3] [CRC : bb79358 | VersionNo : v1] [Size : 1107523] [Status : [VALID]]
-----------------------------------
```

Figura 4.6: Modificarea stării fișierului încărcat, la primirea heartbeat-ului de la toate nodurile interne

După ce va primi acest *token*, aplicația de tip intermediar al clientului va trimite fișierul către primul nod intern din lanț care, la rândul său, va trimite următorului nod intern, și așa mai departe până la finalul lanțului de adrese. În același timp, intermediarul clientului va aștepta mesaj de confirmare de la toate nodurile interne. După primirea confirmării de la cel puțin unul dintre noduri, se va trimite confirmarea și către nodul general, care va înregistra fișierul (vezi Figura 4.7).

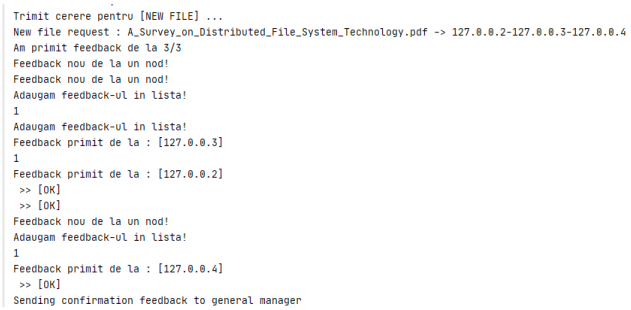

Figura 4.7: Activitatea intermediarului clientului la cererea de încărcare a unui fișier

Un alt aspect considerabil în procesul de adăugare a unui nou fișier este faptul că se realizează o echilibrare a încărcării din punct de vedere a memoriei ocupate la nivelul nodurilor interne. La adăugarea unui nou fișier, acesta va fi replicat pe nodurile care au cel mai mult spațiu de memorie disponibil (vezi Figura 4.8).

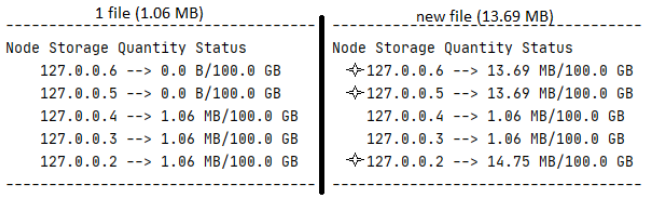

Figura 4.8: Echilibrarea încărcării cantităților de stocare ale nodurilor intern

4.7.2. Prelucrarea unui fișier

Operațiile de prelucrare a unui fișier (redenumire și eliminare) presupun interacțiunea intermediarului clientului doar cu nodul manager general. Cererea de prelucrare va fi trimisă către nodul general care va identifica nodurile care conțin replici ale fișierului. Se vor trimite cereri către aceste noduri interne (vezi Figura 4.9). În cazul operației de redenumire a unui fișier, tabela de conținut va fi actualizată prin setarea stării fișierului având versiunea anterioară în *DELETED,* astfel încât, la repornirea unui nod care conține fișierul, dar nu a fost activ pe parcursul operației, replica să fie eliminată astfel încât să se evite încărcarea inutilă a memoriei (vezi Figura 4.10).

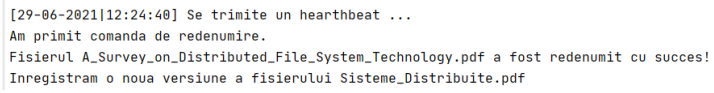

Figura 4.9: Operația de redenumire la nivelul unui nod intern

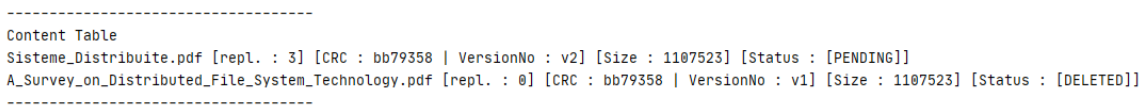

Figura 4.10: Starea tabelei de conținut la redenumirea unui fișier

Similar operației de încărcare a unui nou fișier, fișierul este trecut în starea *PENDING* astfel încât să fie ignorat de mecanismul de replicare. După ce toate nodurile includ noua versiune a fișierului în mesajul de tip heartbeat, fișierul va fi înregistrat ca fiind *VALID (vezi*. Figura 4.11).

```
-------------------------------------
Storage Status Table
User id  : 3
    Filename : Sisteme_Distribuite.pdf
        127.0.0.4  [CRC : bb79358 | VersionNo : v2]
        127.0.0.3  [CRC : bb79358 | VersionNo : v2]
        127.0.0.2  [CRC : bb79358 | VersionNo : v2]

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

-------------------------------------
Content Table
Sisteme_Distribuite.pdf [repl. : 3] [CRC : bb79358 | VersionNo : v2] [Size : 1107523] [Status : [VALID]]
A_Survey_on_Distributed_File_System_Technology.pdf [repl. : 0] [CRC : bb79358 | VersionNo : v1] [Size : 1107523] [Status : [DELETED]]
-------------------------------------
```

Figura 4.11: Starea tabelei de conținut la redenumirea fișierului, după primirea confirmării

4.7.3. Versionarea fișierului

În cadrul operației de încărcare a unui nou fișier, se va crea și fișierul de metadate, pe baza numelui fișierului, cu schimbarea extensiei în *.metadata.* Pe măsură ce fișierul va fi prelucrat, prin operații de redenumire și actualizare, se vor înregistra toate aceste versiuni în cadrul acestui fișier (vezi Figura 4.13). În mod constant, se va verifica dacă versiunile sunt aceleași la nivelul tuturor replicilor. În caz contrar, replica respectivă va fi eliminată, generându-se o replică ce va avea ultima versiune a fișierului. Astfel, în cadrul operației de replicare a unui fișier, va fi replicat și fișierul de metadate, astfel încât să fie înregistrate versiunile fișierului.

În cazul operației de redenumire, se observă că se înregistrează noua versiune a fișierului însă suma de control corespunzătoare fișierului rămâne aceeași din cauza faptului că efectuarea calculului sumei de control se va realiza doar asupra conținutului fișierului, care nu include și numele fișierului. În schimb, în cazul operației de actualizare a unui fișier, ce implică modificarea conținutului, actualizarea versiunii va implica și modificarea sumei de control. La nivelul interfeței utilizator, se vor expune toate aceste versiuni ale fișierului (Figura 4.12).

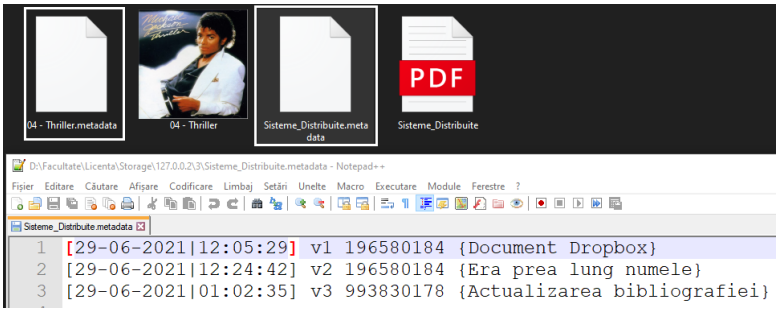

Figura 4.13: Fișierele de metadate care conțin informații despre versiunile fișierelor

Figura 4.12: Versiunile fișierului, la nivelul interfeței cu utilizatorul

4.7.4. Mecanismul de replicare

Atunci când se produc erori la nivelul nodurilor interne, care cauzează inactivitatea acestora sau coruperea unor fișiere, mecanismul de verificare a stării nodurilor interne va detecta acest comportament și lua deciziile de replicare corespunzătoare. Inactivitatea nodului intern va fi detectată prin lipsa mesajelor de tip heartbeat. În schimb, coruperile fișierelor vor fi detectate atât prin analiza versiunii fișierului, cât și a sumei de control. În aceste situații, se va realiza o

cerere de replicare a fișierelor respective pe alte noduri. Inițial, se va găsi o sursă de replicare, în mod aleatoriu și o destinație care va putea stoca replica fișierului. Se va trimite o cerere de replicare către nodul sursă, care va trimite replica și fișierul de metadate către noul nod destinație. (vezi Figura 4.14). În Figura 4.15 se poate observa comportamentul nodurilor interne.

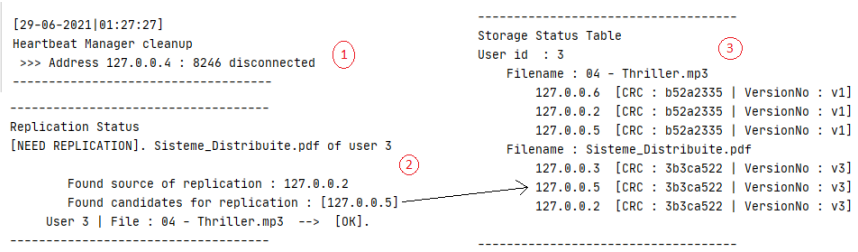

Figura 4.14: Replicarea unui fișier (nodul general)

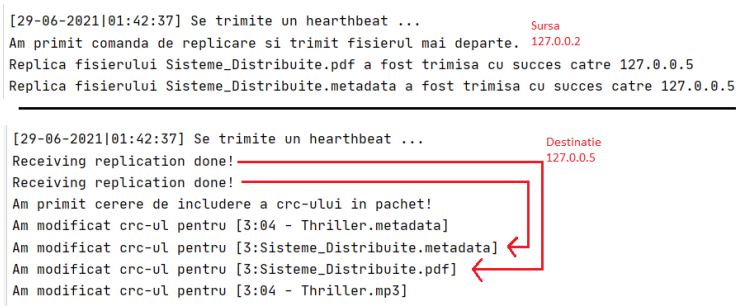

Figura 4.15: Replicarea unui fișier (nodurile interne)

În schimb, dacă unul dintre aceste noduri oprite va reporni, numărul de replici ale unui fișier va fi mai mare decât factorul de replicare destinat pentru fișier. Din acest motiv, se va genera eliminarea unei replici de la unul dintre nodurile interne. Se vor elimina atât fișierul propriu zis, cât și fișierul de metadate (vezi Figura 4.16).

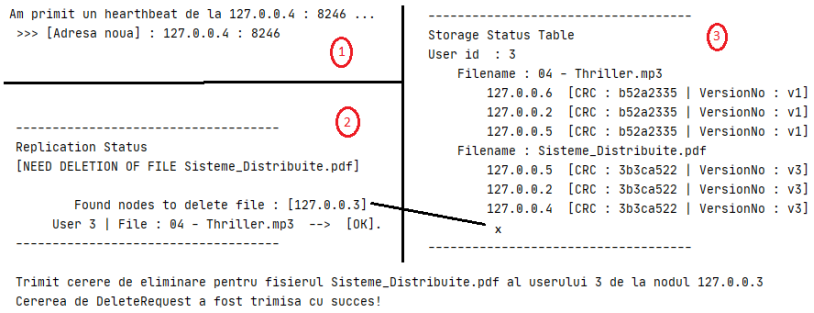

Figura 4.16: Eliminarea fișierului în urma supra-replicării

Concluzii

Principalul obiectiv de care s-a ținut cont în dezvoltarea sistemului distribuit de stocare de fișiere a fost îndeplinirea celor trei principii elementare ale unui sistem de stocare de fișiere. Accesibilitatea și disponibilitatea sistemului au fost asigurate prin mecanismul de replicare a fișierelor pe mai multe noduri ale sistemului, astfel încât starea sistemului să nu fie afectată de erori apărute la nivelul nodurilor interne. Integritatea datelor a fost asigurată prin mecanismele de vers ionare și verificare a sumei de control, astfel încât să se verifice faptul că toate replicile existente ale unui anumit fișiere împărtășesc aceeași versiune și sumă de control. Mutabilitatea este asigurată prin multitudinea de operații asupra fișierelor, puse la dispoziție pentru utilizator. Se pot efectua operații de adăogare, redenumire, eliminare și modificare a fișierului. Aceste operații sunt gestionate în mod independent și paralel la nivelul serverului.

Cu toate acestea, în dezvoltarea sistemului nu au fost tratate anumite aspecte importante. În primul rând, sistemul este vulnerabil al defecte apărute la nivelul nodurilor care rulează într-o singură instanță (intermediarul clientului și nodul manager general). Oprirea acestor noduri ar duce la inactivitatea sistemului. Dacă intermediarul clientului ar fi oprit, s-ar rupe conexiunea cu clientul, iar dacă nodul general ar fi oprit, s-ar pierde toată starea sistemului din punct de vedere al fișierelor stocate. În cazul intermediarului clientului, repornirea sistemului ar readuce o funcționare corectă a sistemului, având în vedere că arhitectura acestei aplicații este fără stare. În schimb, în cazul nodului general, repornirea sistemului nu ar putea garanta refacerea corectă a stării sistemului. Este asigurat un mecanism de refacere a stării sistemului pe baza mesajelor de tip *heartbeat* primite de la nodurile interne, însă nu se poate ține cont și de evenimentele apărute între timp (între momentul pornirii și opririi nodului general) la nivelul acestor noduri. Soluția în cazul acestor două aplicații ar fi crearea unui mecanism de *mirror,* o aplicație care rulează în umbră, în același timp cu aplicația principală, menține aceeași stare a sistemului, însă devine activă doar dacă aplicația principală a eșuat. Prin acest mecanism s-ar putea elimina elementul de tip *single point of failure*, sistemul fiind activ în permanență.

La solicitarea accesării unui fișier către nodul general, acesta va întoarce adresa unuia dintre nodurile interne care stochează o replică a fișierului și care ar putea servi cererea. Pentru a mări viteza de acces a utilizatorilor la fișierele acestora, se poate alege nodul intern cel mai apropiat de utilizator, din punct de vedere geografic. Cu toate că a fost implementat suportul pentru această operație, prin definirea țării în care se află utilizatorul și a țării în care este instalat nodul intern, nu s-a reușit implementarea completă.

Pentru a avea o arhitectură cât mai corectă, fără vulnerabilități, componenta pentru stocarea obiectelor prin care se realizează interacțiunea cu utilizatorul și componenta prin care este identificat și autorizat utilizatorul, trebuie să fie separate. Astfel, aplicația care persista datele de identificare ale utilizatorilor trebuie să fie separată de aplicația care realizează autentificarea utilizatorilor. Pentru mecanismele de persistare a informațiilor se poate folosi o interfață de tip REST, care să expună și funcționalități pentru persistarea datelor în baza de date, iar pentru mecanismele de autentificare a utilizatorului să se folosească o altă componentă. Mai mult decât atât, trebuie definită o separare logică între tipurile de obiecte la nivelul interfeței de tip REST, pentru a putea elimina eventualele vulnerabilități rezultate în urma vehiculării unei anumite categorii de obiecte. Astfel, componenta de persistență a utilizatorilor ar trebui separată din punct de vedere logic, și chiar și fizic, de componenta de persistență a evenimentelor care apar în sistem.

Sistemul distribuit are o arhitectură de tip master/slave ce poate fi ușor scalată, atât pe orizontală, cât și pe verticală. Însă, trebuie asigurate toate mecanismele prin care sistemul să nu

pice niciodată iar starea sistemului să fie mereu consistentă. Așadar, o primă direcție de dezvoltare este reprezentată de adăugarea unor componente de tip mirror la nivelul intermediarului clientului, pentru a păstra în permanență comunicarea cu client, dar, mai ales, la nivelul nodului general, pentru a asigura posibilitatea prelucrării de fișiere, a mecanismelor de replicare și verificare a integrității. Totodată, sistemul de fișiere oferă suport pentru versionare. Pentru fiecare fișier existent în sistem, sunt menținute detalii despre versiunea curentă și toate versiunile anterioare ale fișierului. Însă, momentan, aceste detalii conțin doar informații de identificare și descriere a versiunii. O direcție de dezvoltare este reprezentată de menținerea versiunilor anterioare, expunerea acestora către utilizator și posibilitatea de restaurare a unei anumite versiuni. Un astfel de comportament este foarte eficient în cazul în care fișierele, în special de tip text, suferă modificări foarte dese și ar trebui frecvent restaurate la o versiune anterioară.

În acest stadiu al aplicației, experiența utilizatorului, din punct de vedere al interacțiunii cu propriul conținut, este elementară, întrucât sunt disponibile doar operațiile de bază asupra fișierelor. Pentru a îmbunătăți experiența utilizatorului se poate adăuga posibilitatea grupării fișierelor în cadrul unor directoare la nivelul interfeței cu utilizatorul, pentru o mai bună organizare a conținutului. De asemenea, pentru a eficientiza procesul de modificare a unui fișier, se poate adăuga o fereastră complexă de editare, care permite utilizatorilor să prelucreze propriile fișiere direct în cadrul aplicației, fără a mai trebui să modifice fișierul local, apoi să îl reîncarce în sistem. Din perspectiva administratorului, experiența utilizării se poate îmbunătăți prin extinderea operațiilor disponibile asupra nodurilor sistemului. Astfel, sistemul se poate extinde prin adăugarea posibilității de pornire și oprire a nodurilor interne ale sistemului, decizie luată în urma analizării eventualelor erori apărute la nivelul componentelor.

Sistemul distribuit de stocare de fișiere a fost proiectat într-o manieră care să asigure disponibilitatea, integritatea și fiabilitatea. Fișierele utilizatorilor sunt mereu disponibile datorită mecanismului de verificare și replicare, care asigură distribuirea replicilor fișierului pe mai multe noduri ale sistemului. Integritatea fișierelor este verificată periodic, prin intermediul sumelor de control și a versiunilor fișierului. Se asigură menținerea sănătății stării sistemului prin comunicarea frecventă dintre nodurile sistemului, prin intermediul comunicației de tip multicast. Sistemul suportă conexiuni paralele pentru a putea trata cererile mai multor clienți simultan. Un astfel de sistem este esențial într-o perioadă în care volumul de date vehiculat prin rețea crește exponențial la fiecare an, odată cu nevoia utilizatorilor de a își menține datele într-un mod sigur într-o locație de unde să le poată accesa în mai multe moduri prin intermediul rețelei.

Bibliografie

[1] Arne, Holst, „Amount of data created, consumed, and stored 2010-2025” [Online], Disponibil la adresa: https://www.statista.com/statistics/871513/worldwide-data-created/, Accesat: 2021.

[2] Eduardo Pinheiro, Wolf-Dietrich Weber and Luiz Andre Barroso, „Failure Trends in a Large Disk Drive Population”, Google Inc, February 2007.

[3] Riccardo Pinciroli, Lishan Yang, Jacob Alter, and Evgenia Smirni, „The Life and Death of SSD and HDD: Similarities, Differences, and Prediction Models”, pp. 6, December 2020.

[4] Andrew S. Tanenbaum & Maarten Van Steen, „Distributed Systems. Principles and Paradigms”, Pearson Education, Vrije Universiteit, pp. 21, 2007.

[5] Matthias Klusch, „Service Discovery”, Encyclopedia of Social Networks and Mining , pp. 1, 2014.

[6] Frederic Lardinois, „Google Drive will hit a billion users this week” [Online], Disponibil la adresa: https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week/, July 2018.

[7] Dropbox Organization, „Dropbox Investor Relations” [Online], Disponibil la adresa: https://dropbox.gcs-web.com/, Accesat: 2021.

[8] J Blomer, „A Survey on Distributed File System Technology”, IOP Publishing, CERN, CH-1211 Geneve 23, Switzerland, , pp. 2-3, 6, 2015.

[9] Remzi Arpaci-Dusseau, „The Andrew File System”, Ostep, capitol 50.1, pp. 1-6, 2008.

[10] Apache Software Foundation, „HDFS Architecture” [Online], Disponibil la adresa: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, Accesat: June, 2021.

[11] Dhruba Borthakur,„HDFS Architecture Guide”, Apache Software Foundation, pp. 1-5, 2008

[12] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, „The Google File System”, Google Inc, pp. 4-10, October 2003.

[13] Andrew Josey, POSIX™ 1003.1 Frequently Asked Questions (FAQ Version 1.18) [Online], Disponibil la adresa: http://www.opengroup.org/austin/papers/posix_faq.html, Accesat: 2013.

[14] Giacinto Donvito, Giovanni Marzully, Domenico Diacono, „Testing of several distributed file-systems for supporting the HEP experiments analysis”, IOP Publishing, Journal of Physics : Conference Series 513, pp. 4, 2014.

[15] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn, „Ceph: A Scalable, High-Performance Distributed File System”, University of California, Santa Cruz, pp. 1-8, 2006.

[16] Sage A. Weil, Scott. A. Brandt, Ethan L. Miller, Carlos Maltzahn, „CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data”, University of California, Santa Cruz, , pp. 2-3, 2016.

[17] Anagha Ramnath Kadve, „Trade of between SSD and HDD”, Bharati Vidyapeeth Deemed University College of Enginners, pp. 474, July 2016.

[18] Oracle, „Trail: JavaBeans” [Online], Disponibil la adresa: https://docs.oracle.com/javase/tutorial/javabeans/ , Accesat: 2021 .

[19] M. Jones, J. Bradley, N. Sakimura, RFC 7519 : JSON Web Token (JWT) [Online], Disponibil la adresa: https://datatracker.ietf.org/doc/html/rfc7519, Accesat: 2021.

[20] „Architectural Styles and the Desing of Network-based Software Architectures, Doctoral dissertation”, University of California, Irvine, pp. 66-105, 2000.

[21] Oracle, „What are RESTful WebSerivces?” [Online], Disponibil la adresa: https://javaee.github.io/tutorial/jaxrs001.html, Accesat: 2021.

[22] Hebert Coelho, Byron Kiourtzoglou, „Java Persistence API - Mini Book”, Java Code Geeks, pp. 4-20, April 2017.

[23] Vanessa Wang, Frank Salim, Peter Moskovits, „The Definitive Guide to HTML5 WebSocket”, Apress, pp.32, 2013.

[24] Phillip Webb, Dave Syer, Josh Long , „Spring Boot Reference Documentation 2.5.2” [Online] , Disponibil la adresa: https://docs.spring.io/spring-boot/docs/2.5.2/reference/pdf/spring-boot-reference.pdf, Accesat: 2021 .

[25] Dan Umbarger, „An Introduction to Polymorphism in Java ”, AP Computer Science, Dallas Texas, pp. 2-20, 2008.

[26] Basic of Synchroniation in Java [Online], Disponibil la adresa: http://javajee.com/basics-of-synchronization-in-java, Accesat: 2021.

[27] Apache Software Foundation, „Apache Maven Project” [Online], Disponibil la adresa: https://maven.apache.org/guides/, Accesat: 2021.

[28] Facebook, React [Online], Disponibil la adresa: https://reactjs.org/, Accesat: 2021.

[29] JSON Web Token Support For The JVM [Online], Disponibil la adresa: https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt, Accesat: 2021.

[30] Oracle, „MulticastSocket Implementation” [Online], Disponibil la adresa: https://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html, Accesat: 2021.

[31] Juergen Hoeller, Trevor D. Cook, „MultipartFile Interface” [Online], Disponibil la adresa: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html, Accesat: 2021.

[32] Oracle, „The Java EE 6 Tutorial. Chapter 32. Introduction to the Java Persistence API.” [Online], Disponibil la adresa: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html, January 2013, Accesat: 2021.

[33] Building jar with dependencies using Maven [Online], Disponibil la adresa: https://maven.apache.org/plugins/maven-assembly-plugin/usage.html, Accesat: 2021.

[34] M. Jones, D. Hardt, „ The OAuth 2.0 Authorization Framework: Bearer Token Usage” [Online], Disponibil la adresa: https://datatracker.ietf.org/doc/html/rfc6750, Accesat: 2021.

Anexe

Anexa 1. Proiectarea modulului GeneralManager

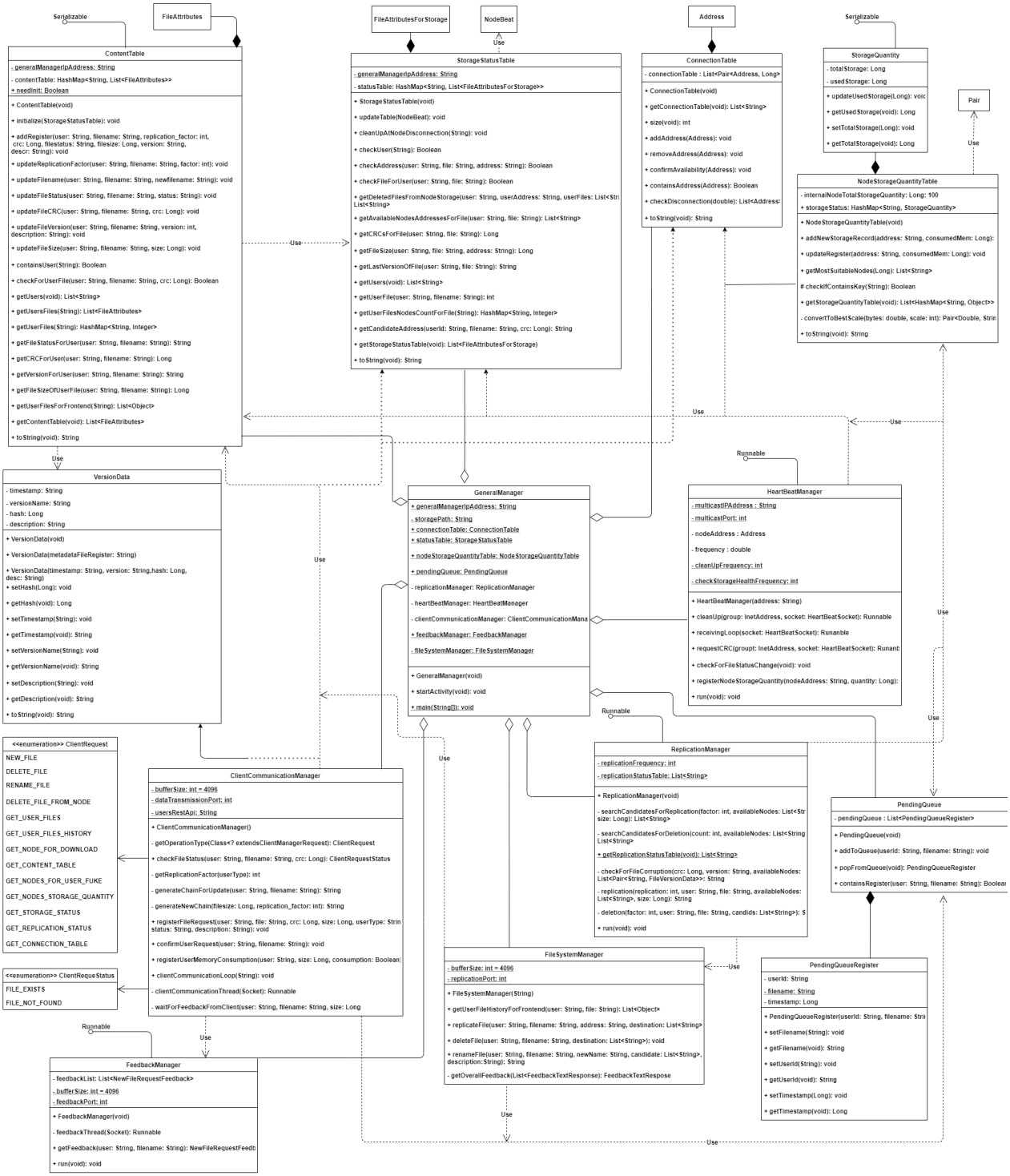

Figura A.1: Diagrama UML a modulului GeneralManager

Anexa 2. Proiectarea modulului GeneralNode

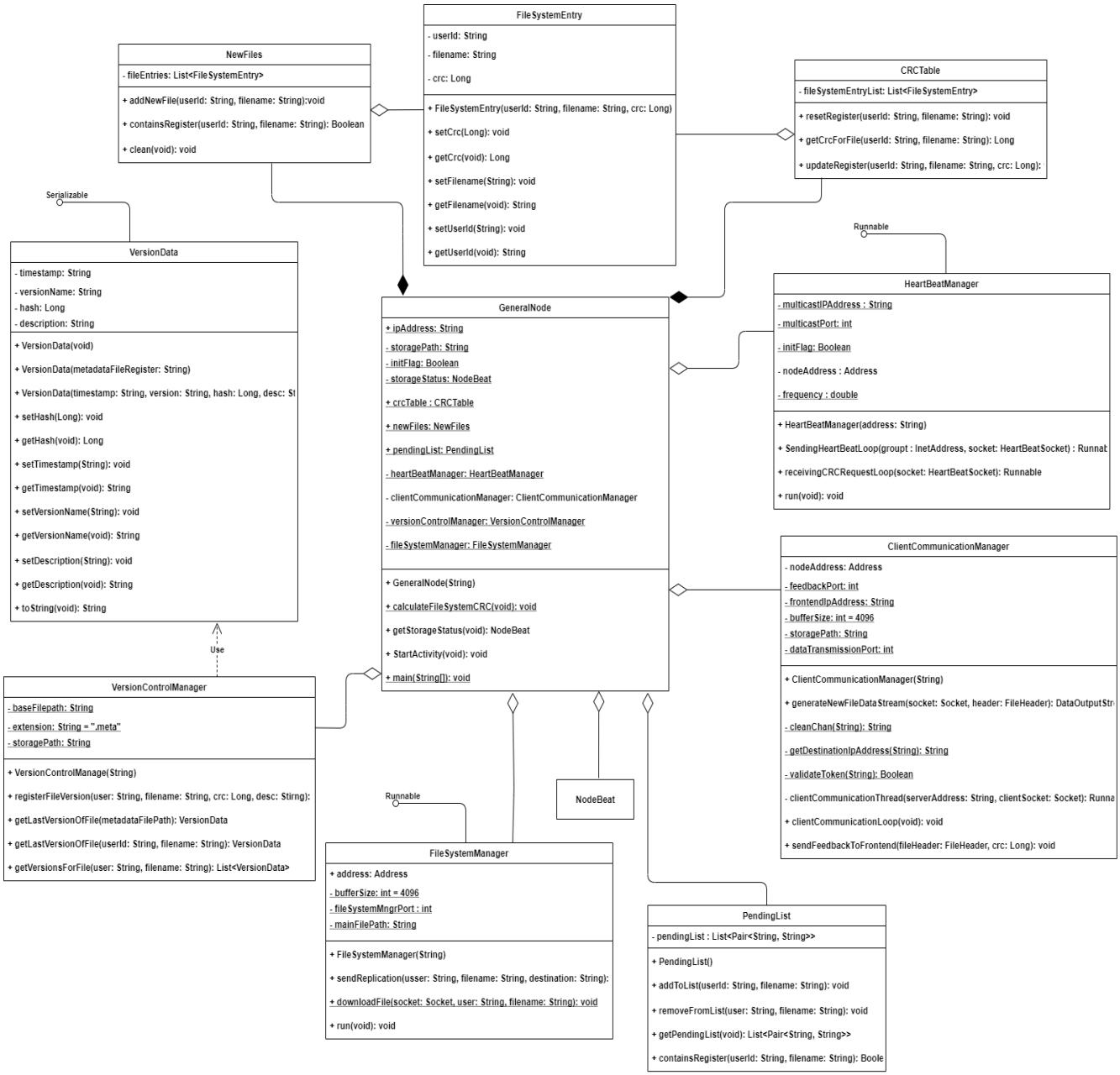

Figura A.2: Diagrama UML a modulului GeneralNode

Anexa 3. Proiectarea modulului FrontendProxy

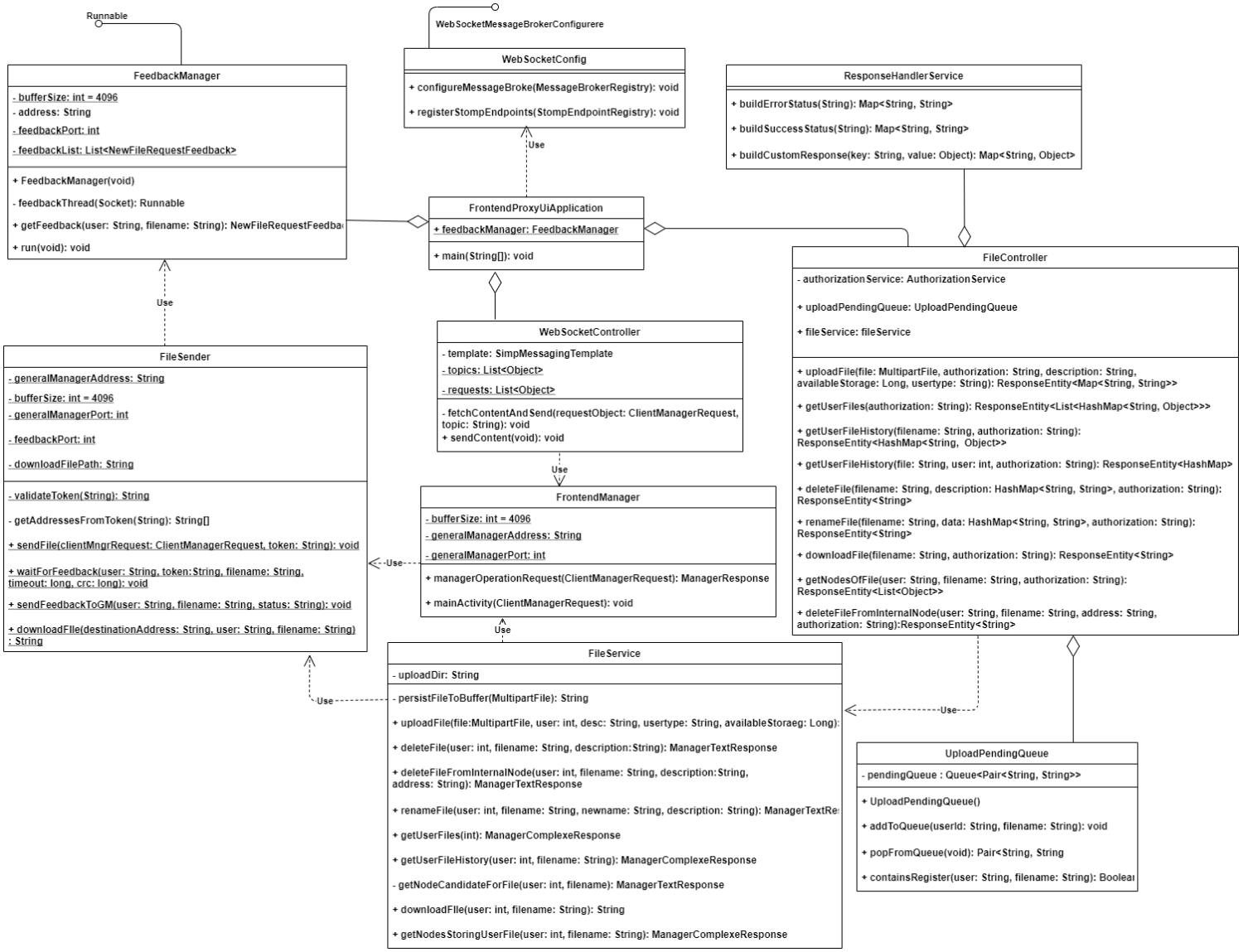

Figura A.3: Diagrama UML a modulului FrontendProxy

Anexa 4. Proiectarea modulului GeneralPurpose

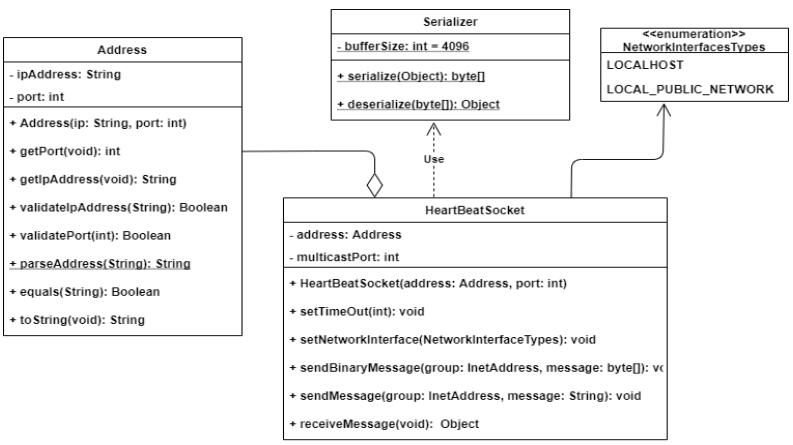

Figura A.5: Diagrama UML a pachetului de comunicații

Figura A.4: Diagrama UML a pachetului de efectuare a operațiilor specifice sistemului de operare cu fișiere

Anexa 5. Proiectarea modulului Beans

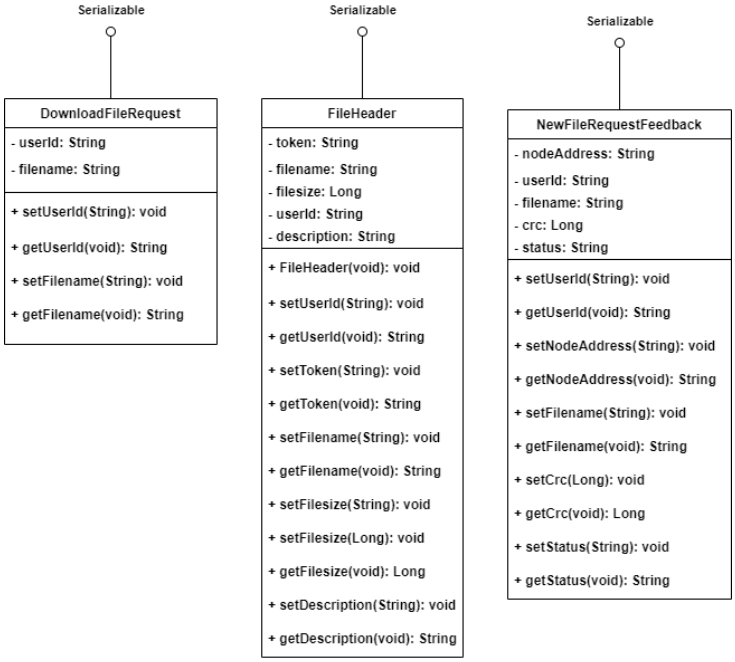

Figura A.6: Diagrama UML a pachetului de conține obiectele folosite în comunicarea dintre client și nodurile interne

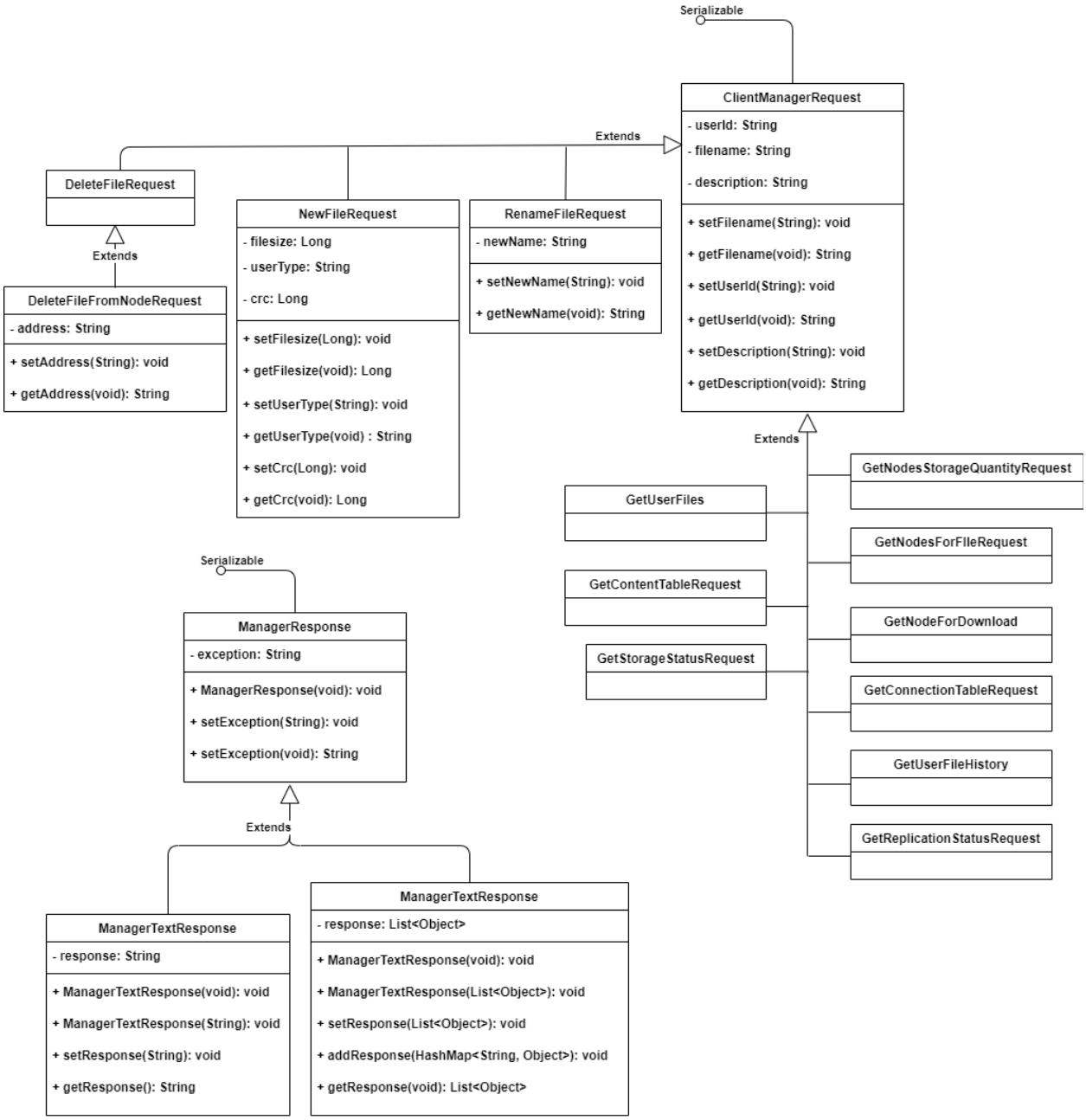

Figura A.7: Diagrama UML a pachetului ce conține obiectele beans folosite în comunicarea dintre client și nodul general

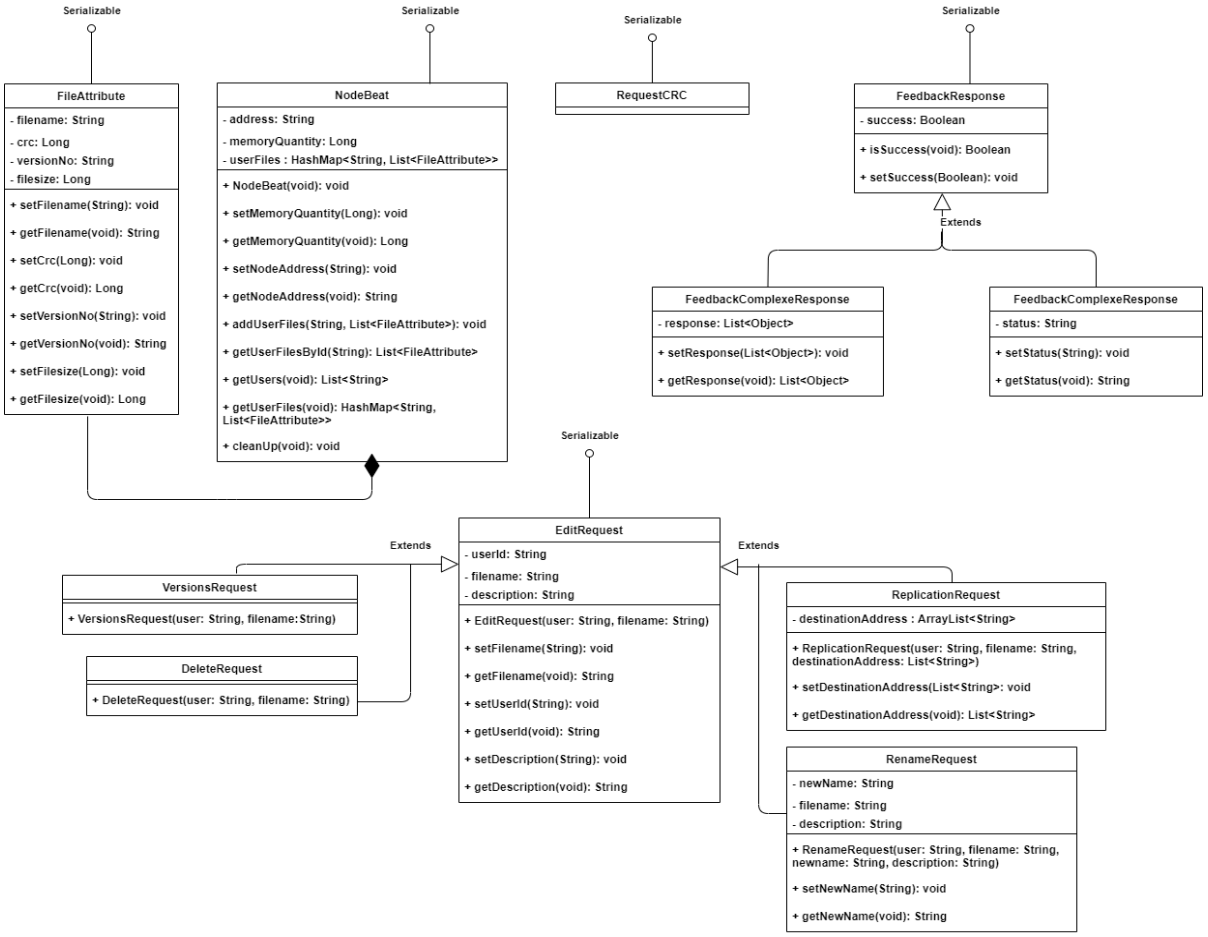

Figura A.8: Diagrama UML a pachetului ce conține obiectele folosite în comunicarea dintre nodul general și nodurile interne

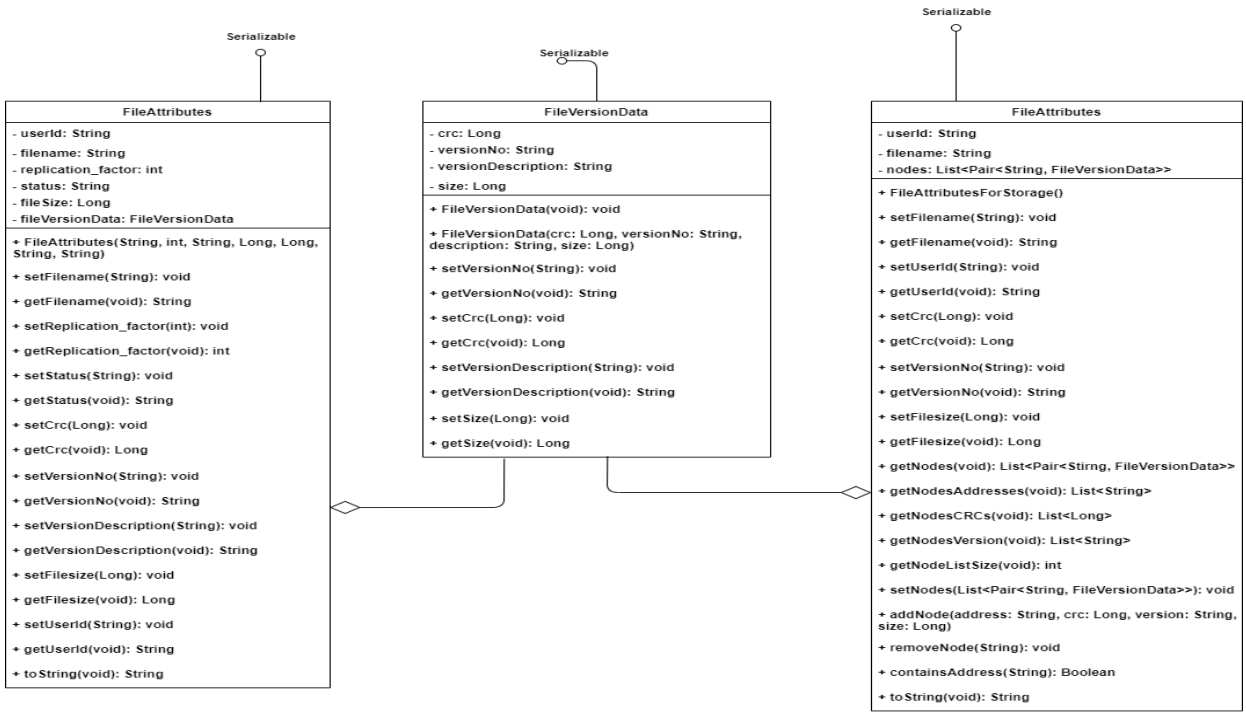

Figura A.9: Diagrama UML a pachetului ce conține obiectele folosite în reprezentarea entităților stării ce vor fi vehiculate între client și nodul general

Anexa 6. Proiectarea modulului HTTPHandler

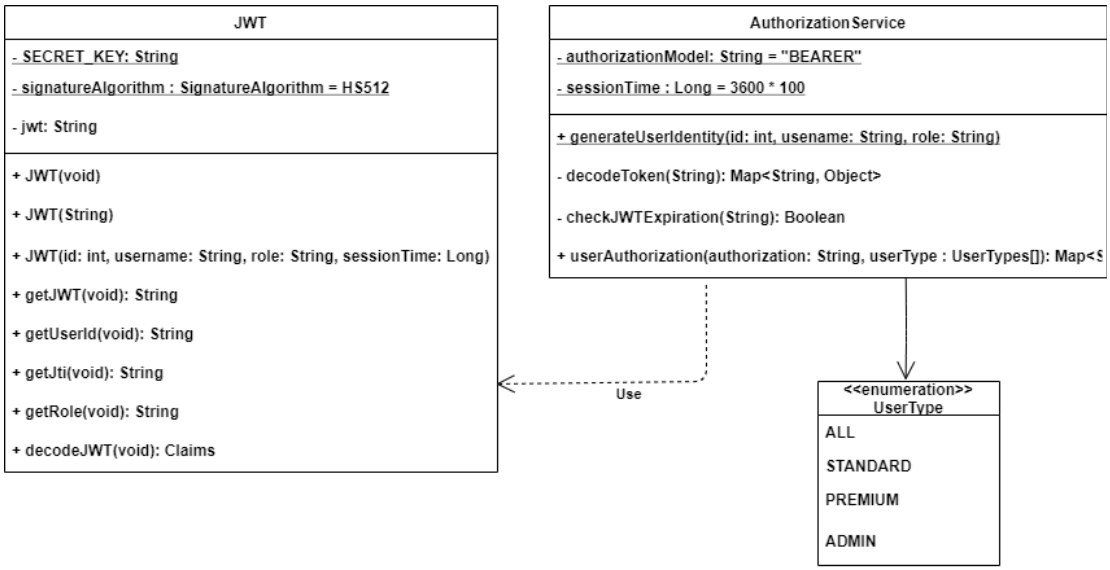

Figura A.10: Diagrama UML a pachetului ce definește elementele specifice *JWT*

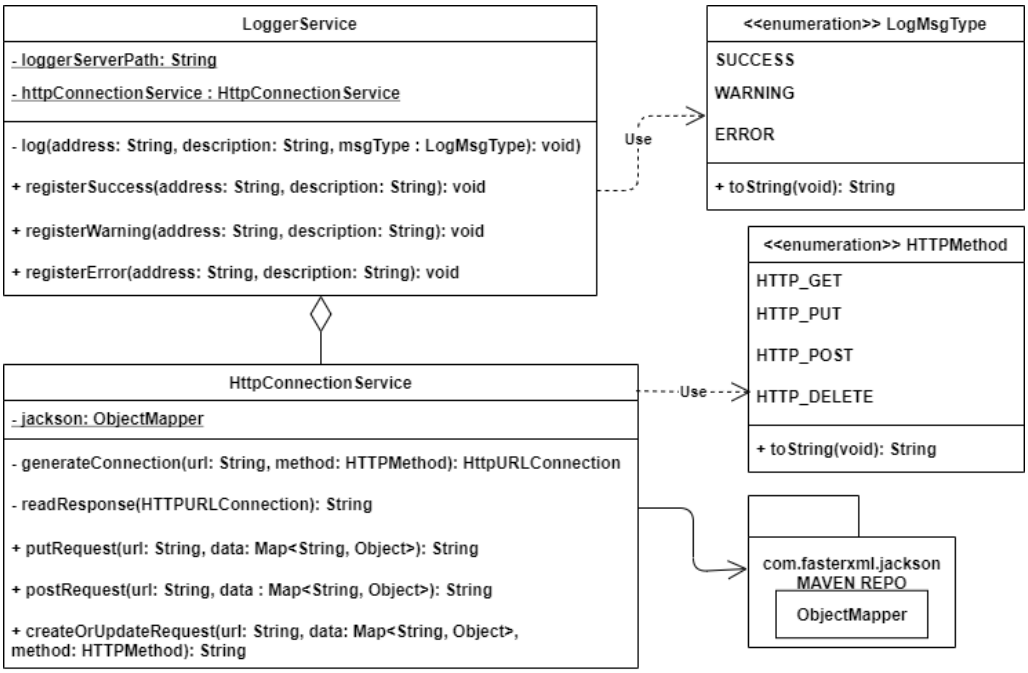

Figura A.11: Diagrama UML a pachetului ce definește funcționalitățile necesare efectuării de cereri HTTP și de înregistrare a evenimentelor din sistem

Anexa 7. Proiectarea modulului RESTApi

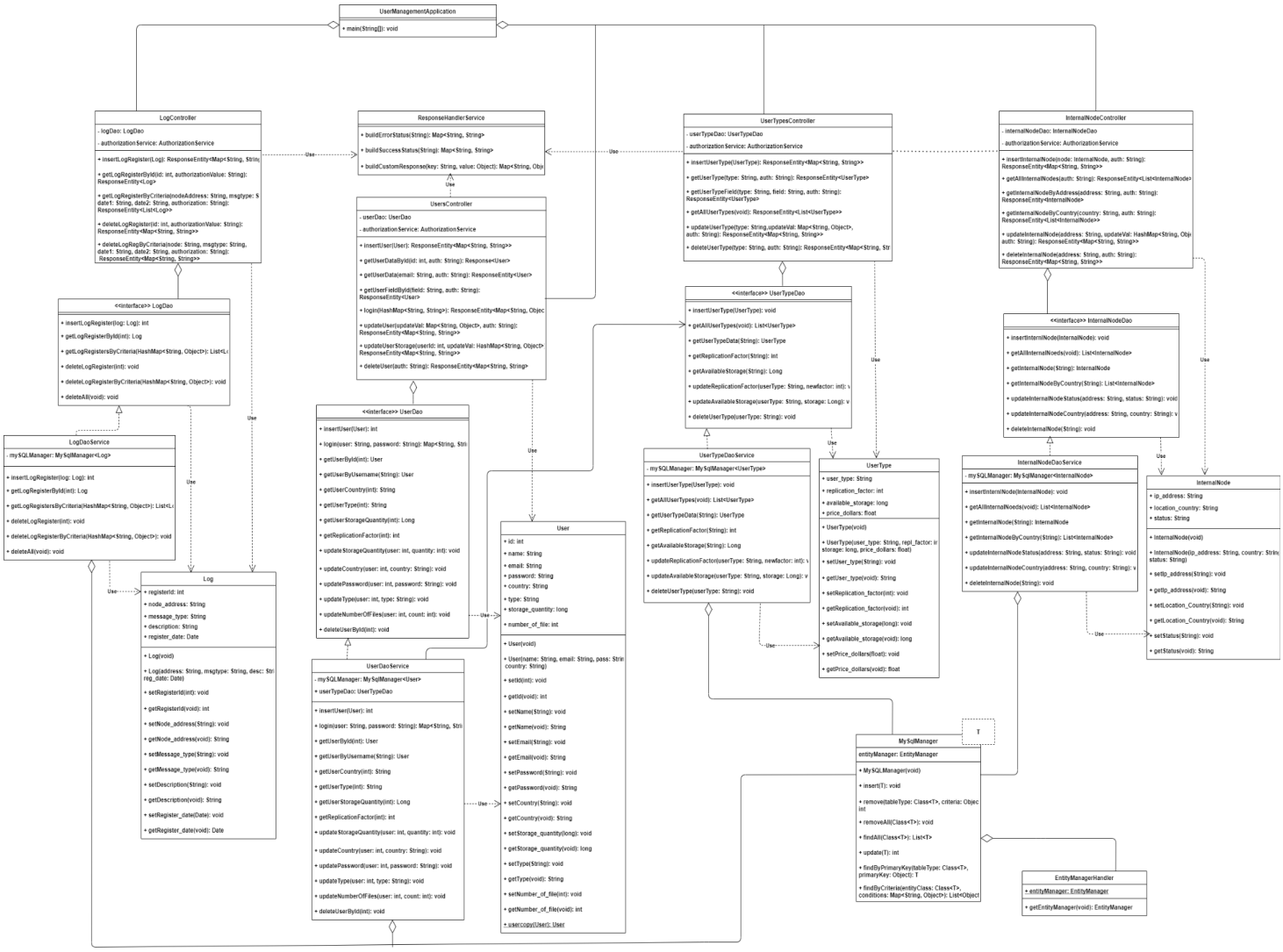

Figura 12: Diagrama UML a modulului RESTApi

Anexa 8. Codul clasei ReplicationManager din modulul GeneralManager

```
public class ReplicationManager implements Runnable{
  private static int replicationFrequency =
Integer.parseInt(AppConfig.getParam("replicationFrequency"));
  private static List<String> replicationStatusTable = new ArrayList<>();

  public ReplicationManager(){}

  private List<String> searchCandidatesForReplication(int replication_factor,
List<String> availableNodes, long filesize){[...]}

  private List<String> searchCandidatesForDeletion(int count, List<String>
```

```
availableNodes){[...]}

  public static List<String> getReplicationStatusTable(){[...]}

  private String checkForFileCorruption(long crc, String versionNo,
List<Pair<String, FileVersionData>> availableNodesForFile){
    if(crc == -1){
      return null;
    }
    for(Pair<String, FileVersionData> file : availableNodesForFile){
      if(file.getSecond().getCrc() != -1 && file.getSecond().getCrc() != crc &&
file.getSecond().getVersionNo() != versionNo){
        LoggerService.registerWarning(GeneralManager.generalManagerIpAddress,
"Found corrupted file at address : " + file.getFirst());
        return file.getFirst();
      }
    }
    return null;
  }

  private String replication(int replication_factor, String userId, String
userFile, List<String> availableNodesAddressesForFile, long filesize){
    String status = "[NEED REPLICATION]." + userFile+" of user " +  userId + "\n";
    LoggerService.registerWarning(GeneralManager.generalManagerIpAddress, status);
    List<String> candidates = searchCandidatesForReplication(replication_factor,
availableNodesAddressesForFile, filesize);
    if(replication_factor == 1 || candidates == null){
      LoggerService.registerWarning(GeneralManager.generalManagerIpAddress,"Nu se
poate realiza replicarea pentru fisierul curent. " +
          "Nu exista suficiente noduri pe care sa se faca replicarea.");
    }
    else {
      try{
        GeneralManager.contentTable.updateFileStatus(userId, userFile,
"[PENDING]");
      }
      catch (Exception exception){
        System.out.println("Replication : updatefilestatus1 : " +
exception.getMessage());
      }
      // cautam un criteriu pe baza caruia selectam nodul de la care se face
copierea
      String source = availableNodesAddressesForFile.get(0);
      status += "\t\tFound source of replication : " + source + "\n\t\tFound
candidates for replication : ";
      for (String candidate : candidates) {
        status += "[" + candidate + "] ";
      }
      GeneralManager.fileSystemManager.replicateFile(userId, userFile, source,
candidates);
```

```
    }
    return status;
  }

  private String deletion(int replication_factor, String userId, String userFile,
List<String> candidates) throws Exception {
    String status = String.format("[NEED DELETION OF FILE %s]\n", userFile);
    LoggerService.registerWarning(GeneralManager.generalManagerIpAddress,status);
    status += "\t\tFound nodes to delete file : ";
    for (String candidate : candidates) {
      status += "[" + candidate + "] ";
    }
    GeneralManager.fileSystemManager.deleteFile(userId, userFile, candidates);
    if(replication_factor == 0) {
      GeneralManager.contentTable.updateFileStatus(userId, userFile, "[DELETED]");
    }
    return status;
  }

  @Override
  public void run(){
    while(true) {
      replicationStatusTable.clear();
      try {
        for (String userId : GeneralManager.contentTable.getUsers()) {
          HashMap<String, Integer> userFiles =
GeneralManager.contentTable.getUserFiless(userId);
          for (String userFile : new ArrayList<>(userFiles.keySet())) {
            int replication_factor = userFiles.get(userFile);
            List<Pair<String, FileVersionData>> availableNodesForFile =
GeneralManager.statusTable.getAvailableNodesForFile(userId, userFile);
            List<String> availableNodesAddressesForFile =
GeneralManager.statusTable.getAvailableNodesAddressesForFile(userId, userFile);
            if(availableNodesForFile ==  null)
              // eroare de sincronizare; se va rezolva la iteratia urmatoare a
for-ului prin useri
              continue;
            String replicationStatus = "\t User " + userId + " | File : " +
userFile + "  -->  ";
            if (replication_factor == availableNodesForFile.size()) {
              if(GeneralManager.contentTable.getFileStatusForUser(userId,
userFile).contains("PENDING")) {
                replicationStatus += "[UNKNOWN]\n";
                replicationStatusTable.add(replicationStatus);
                continue;
              }
              long crc = GeneralManager.contentTable.getCRCForUser(userId,
userFile);
              String versionNo =
GeneralManager.contentTable.getVersionForUser(userId, userFile);
```

```
              String corruptedFileAddress = this.checkForFileCorruption(crc,
versionNo, availableNodesForFile);
              if(corruptedFileAddress == null) {
                replicationStatus += "[OK].";
                replicationStatusTable.add(replicationStatus);
              }
              else{
                List<String> candidatesForDeletion = new ArrayList<String>(){{
                  add(corruptedFileAddress);
                }};
                replicationStatus += this.deletion(replication_factor, userId,
userFile, candidatesForDeletion);
                replicationStatusTable.add(replicationStatus);
                continue;
              }
            }
            else if (replication_factor > availableNodesForFile.size() && !
GeneralManager.contentTable.getFileStatusForUser(userId,
userFile).contains("PENDING")) {
              long filesize =
GeneralManager.contentTable.getFileSizeOfUserFile(userId, userFile);
              replicationStatus += this.replication(replication_factor, userId,
userFile, availableNodesAddressesForFile, filesize);
              replicationStatusTable.add(replicationStatus);
            }
            else if(replication_factor < availableNodesForFile.size()){
              List<String> candidates =
searchCandidatesForDeletion(availableNodesForFile.size() - replication_factor,
availableNodesAddressesForFile);
              replicationStatus += this.deletion(replication_factor, userId,
userFile, candidates);
              replicationStatusTable.add(replicationStatus);
            }
            else{
              replicationStatus += "[UNKNOWN]\n";
              replicationStatusTable.add(replicationStatus);
            }
          }
        }
        Thread.sleep((int) (replicationFrequency * 1e3));
      }
      catch (Exception exception){
        System.out.println("Replication loop interrupted exception : " +
exception.getMessage());
      }
    }
  }
}
```

Anexa 9. Codul clasei HeartbeatManager din modulul GeneralManager

```
public class HearthBeatManager implements Runnable{
  private static String multicastIPAddress =
AppConfig.getParam("multicastIPAddress");
  private static int multicastPort =
Integer.parseInt(AppConfig.getParam("multicastPort"));
  private Address nodeAddress;
  private double frequency =
Integer.parseInt(AppConfig.getParam("hearthBeatFrequency"));
  private static int cleanupFrequency =
Integer.parseInt(AppConfig.getParam("cleanupFrequency"));
  private static int checkStorageHealthFrequency =
Integer.parseInt(AppConfig.getParam("checkStorageHealthFrequency"));

  public HearthBeatManager(String address) throws Exception{
    this.nodeAddress = new Address(address, multicastPort);
  }

  public Runnable cleanUp(){
    return new Runnable(){
      @Override
      public void run(){
        List<Address> disconnected;
        int cleanUpIndex = 1;
        while(true) {
          try {
            if(cleanUpIndex == cleanupFrequency)
{ LoggerService.registerSuccess(GeneralManager.generalManagerIpAddress, "Heartbeat
Manager cleanup");
              checkForFileStatusChange();
              disconnected =
GeneralManager.connectionTable.checkDisconnection(frequency);
              if(disconnected.size() != 0){
                for (Address disconnectedAddres : disconnected) {
            LoggerService.registerWarning(GeneralManager.generalManagerIpAddress,
                      " >>> Address " + disconnectedAddres + " disconnected");
                 GeneralManager.connectionTable.removeAddress(disconnectedAddres);
GeneralManager.statusTable.cleanUpAtNodeDisconnection(disconnectedAddres.getIpAddr
ess());
                }
              }
              cleanUpIndex = 0;
            }
            if(GeneralManager.connectionTable.size() == 0){
             LoggerService.registerWarning(GeneralManager.generalManagerIpAddress,
                  " >>> Niciun nod conectat!");
            }
            cleanUpIndex += 1;
            Thread.sleep((int) (frequency * 1e3));
            if(GeneralManager.contentTable.needInit)
```

```
              GeneralManager.contentTable.initialize(GeneralManager.statusTable);
          } catch (InterruptedException exception) {
            LoggerService.registerError(GeneralManager.generalManagerIpAddress,
                "InterruptedException occured. : " + exception.getMessage());
          }}}};
  }

  public Runnable receivingLoop(HearthBeatSocket socket){
    return new Runnable() {
      @Override
      public void run(){
        NodeBeat message;
        Address receivedAddress;
        while(true){
          try{
            message = (NodeBeat) socket.receiveMessage();
            receivedAddress = Address.parseAddress(message.getNodeAddress());
            System.out.println("Am primit un hearthbeat de la "+ receivedAddress);
            if(!GeneralManager.connectionTable.containsAddress(receivedAddress)){
             LoggerService.registerSuccess(GeneralManager.generalManagerIpAddress,
                  " >>> [Adresa noua] : " + receivedAddress);
              GeneralManager.connectionTable.addAddress(receivedAddress);
            }
            else
              GeneralManager.connectionTable.confirmAvailability(receivedAddress);
            GeneralManager.statusTable.updateTable(message);
            registerNodeStorageQuantity(receivedAddress.getIpAddress(),
message.getMemoryQuantity());
          }
          catch (ClassCastException exception){
          }
          catch (Exception exception)
{ LoggerService.registerError(GeneralManager.generalManagerIpAddress,"Hearthbeatma
nager receiveloop : " + exception.getMessage());
          }}}};
  }

  public Runnable requestCRC(InetAddress group, HearthBeatSocket socket){
    return new Runnable(){
      @Override
      public void run(){
        RequestCRC requestCRC = new RequestCRC();
        while(true) {
          try {
            Thread.sleep((int) (checkStorageHealthFrequency * 1e3));
            LoggerService.registerSuccess(GeneralManager.generalManagerIpAddress,
                Time.getCurrentTimeWithFormat() + " Se trimite cerere pentru
CRC ...");
            socket.sendBinaryMessage(group, Serializer.serialize(requestCRC));
          } catch (IOException exception) {
```

```
            socket.close();
            LoggerService.registerError(GeneralManager.generalManagerIpAddress,
                "IOException occured at requestCRC. : " + exception.getMessage());
          } catch (InterruptedException exception) {
            socket.close();
            LoggerService.registerError(GeneralManager.generalManagerIpAddress,
                "InterruptedException occured at requestCRC. : " +
exception.getMessage());
          }}}};
  }

  public void checkForFileStatusChange(){
    for(int i = 0; i < 3; i++){
      try {
        PendingQueueRegister updateRequest =
GeneralManager.pendingQueue.popFromQueue();
        GeneralManager.contentTable.updateFileStatus(updateRequest.getUserId(),
updateRequest.getFilename(), "[VALID]");
      }
      catch (NullPointerException exception) {
        break;
      }
      catch (Exception exception)
{ LoggerService.registerError(GeneralManager.generalManagerIpAddress,"checkForFile
StatusChange exception : " + exception.getMessage());
      }}
  }

  public void registerNodeStorageQuantity(String nodeAddress, long quantity)
{[...]}

  public void run(){
    try {
      System.out.println(String.format("Node with address [%s] started...",
nodeAddress));
      InetAddress group =
InetAddress.getByName(HearthBeatManager.multicastIPAddress);
      HearthBeatSocket socket = new HearthBeatSocket(nodeAddress, multicastPort);
    socket.setNetworkInterface(HearthBeatSocket.NetworkInterfacesTypes.LOCALHOST);
      socket.joinGroup(group);
      Thread cleanUpThread = new Thread(cleanUp());
      Thread receivingThread = new Thread(receivingLoop(socket));
      Thread requestCRCThread = new Thread(requestCRC(group, socket));
      cleanUpThread.start();
      receivingThread.start();
      requestCRCThread.start();
    }
    catch (IOException exception){
      System.out.println(exception.getMessage());
    }}
```

```
}
```

Anexa 10. Codul clasei ClientCommunicationManager din modulul GeneralNode

```
public class ClientCommunicationManager {
    private Address nodeAddress;
    private static int feedbackPort =
Integer.parseInt(AppConfig.getParam("feedbackPort"));
    private static String frontendIpAddress =
AppConfig.getParam("frontendAddress");
    private static int bufferSize =
Integer.parseInt(AppConfig.getParam("buffersize"));
    private static String storagePath = AppConfig.getParam("storagePath");
    private static int dataTransmissionPort =
Integer.parseInt(AppConfig.getParam("dataTransmissionPort"));

    public ClientCommunicationManager(String address) throws Exception{
        this.nodeAddress = new Address(address, dataTransmissionPort);
    }

    public DataOutputStream generateNewFileDataStream(Socket socket, FileHeader
header) throws IOException {
        DataOutputStream dataOutputStream = new
DataOutputStream(socket.getOutputStream());
        dataOutputStream.write(Serializer.serialize(header));
        return dataOutputStream;
    }

    private static String cleanChain(String token){[...]}

    private static String getDestinationIpAddress(String token) throws Exception{
        if(validateToken(token))
            return token.replace(" ","").split("\\-")[0];
        return null;
    }

    private static boolean validateToken(String token) throws Exception{[...]}

    private Runnable clientCommunicationThread(String serverAddress, Socket
clientSocket){
        return new Runnable() {
            @Override
            public void run(){
                try {
                    InputStream dataInputStream = new
DataInputStream(clientSocket.getInputStream());
                    OutputStream dataOutputStream = null;
                    FileOutputStream fileOutputStream = null;
```

```
                    Socket nextElementSocket = null;
                    FileHeader fileHeader = null;
                    String filepath = null;
                    byte[] buffer = new byte[bufferSize];
                    int read;
                    boolean header_found = false;
                    while((read = dataInputStream.read(buffer,0,bufferSize)) > 0){
                        if(!header_found) {
                            try {
                                fileHeader = (FileHeader)
Serializer.deserialize(buffer);
                                filepath = storagePath + serverAddress + "/" +
fileHeader.getUserId();
                                if(!Files.exists(Paths.get(filepath)))
                                    Files.createDirectories(Paths.get(filepath));

GeneralNode.pendingList.addToList(fileHeader.getUserId(),
fileHeader.getFilename());
                                filepath += "/" + fileHeader.getFilename();
                                fileOutputStream = new FileOutputStream(filepath);
                                String token = cleanChain(fileHeader.getToken());
                                if(token != null){
                                    fileHeader.setToken(token);
                                    String nextDestination =
getDestinationIpAddress(token);
                                    nextElementSocket = new
Socket(nextDestination, nodeAddress.getPort());
                                    dataOutputStream =
generateNewFileDataStream(nextElementSocket, fileHeader);
                                }
                                header_found = true;
                                continue;
                            }
                            catch (ClassCastException exception){
                                DownloadFileRequest downloadFileRequest =
(DownloadFileRequest)Serializer.deserialize(buffer);
                                FileSystemManager.downloadFile(clientSocket,
downloadFileRequest.getUserId(), downloadFileRequest.getFilename());
                            }
                            catch (Exception exception) {
                                LoggerService.registerError(GeneralNode.ipAddress,
"Exceptie la comunicarea cu clientul : " + exception.getMessage());
                            }
                        }
                        else {
                            fileOutputStream.write(buffer, 0, read);
                        }
                        if(nextElementSocket != null){
                            dataOutputStream.write(buffer, 0, read);
                        }
```

```
                    }
                    LoggerService.registerSuccess(GeneralNode.ipAddress, "File
write done");
                    dataInputStream.close();
                    fileOutputStream.close();
                    if(nextElementSocket != null) {
                        nextElementSocket.close();
                        dataOutputStream.close();
                    }
                    clientSocket.close();
                    GeneralNode.pendingList.removeFromList(fileHeader.getUserId(),
fileHeader.getFilename());
                    long filecrc = FileSystem.calculateCRC(filepath);

GeneralNode.versionControlManager.registerFileVersion(fileHeader.getUserId(),
fileHeader.getFilename(), filecrc, fileHeader.getDescription());
                    GeneralNode.crcTable.updateRegister(fileHeader.getUserId(),
fileHeader.getFilename(), filecrc);
                    sendFeedbackToFrontend(fileHeader, filecrc);
                }

                catch (Exception exception){

LoggerService.registerWarning(GeneralNode.ipAddress,String.format("Could not
properly close connection with my friend : [%s : %d]",
                            clientSocket.getLocalAddress(),
                            clientSocket.getLocalPort()));
                }}
        };
    }

    public void clientCommunicationLoop() throws Exception {
        ServerSocket serverSocket = new ServerSocket();
        try{
            serverSocket.bind(new InetSocketAddress(nodeAddress.getIpAddress(),
nodeAddress.getPort()));
            while(true){
                Socket clientSocket = serverSocket.accept();
                new
Thread(clientCommunicationThread(serverSocket.getInetAddress().getHostAddress(),
clientSocket)).start();
            }
        }
        catch (Exception exception){
            serverSocket.close();
        }
    }

    public void sendFeedbackToFrontend(FileHeader fileHeader, long crc){
        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                NewFileRequestFeedback feedback = new NewFileRequestFeedback();
                feedback.setFilename(fileHeader.getFilename());
                feedback.setUserId(fileHeader.getUserId());
                feedback.setNodeAddress(nodeAddress.getIpAddress());
                feedback.setCrc(crc);

                Socket frontendSocket = null;
                DataOutputStream dataOutputStream = null;

                try{
                    frontendSocket = new Socket(frontendIpAddress, feedbackPort);
                    dataOutputStream = new
DataOutputStream(frontendSocket.getOutputStream());

                    dataOutputStream.write(Serializer.serialize(feedback));

                    dataOutputStream.close();
                    frontendSocket.close();
                    LoggerService.registerSuccess(GeneralNode.ipAddress,"Feedback-
ul a fost trimis cu succes catre client");
                }
                catch (IOException exception){
                    LoggerService.registerWarning(GeneralNode.ipAddress,"Exceptie
IO la sendFeedBackToFrontend : " + exception.getMessage());
                }
                finally {
                    try{
                        dataOutputStream.close();
                        frontendSocket.close();
                    }
                    catch(IOException exception){

LoggerService.registerWarning(GeneralNode.ipAddress,"sendFeedBackToFrontend
exceptie la inchidere socket-uri");
                    }}
            }
        }).start();
    }
}
```

Anexa 11. Codul clasei VersionControlManager din modulul GeneralManager

```
public class VersionControlManager {
  private final static String extension = AppConfig.getParam("metadataExtension");
  private static String storagePath = AppConfig.getParam("storagePath");
  private static String baseFilepath;

  public VersionControlManager(String address){baseFilepath = storagePath +
```

```
address + "\\";}
  public void registerFileVersion(String userId, String filename, long crc, String
description){
    String filepath = baseFilepath + userId + "\\" + filename;
    if(!FileSystem.checkFileExistance(filepath)){
      LoggerService.registerError(GeneralNode.ipAddress,"Fisierul " + filename + "
nu a fost salvat inca; Nu putem scrie fisierul de metadate");
      return;
    }
    String metadataFilePath = filepath.substring(0, filepath.lastIndexOf(".")) +
extension;
    VersionData versionData = new VersionData();
    if(!FileSystem.checkFileExistance(metadataFilePath)){
      FileSystem.createFile(metadataFilePath);
      versionData.setVersionNumber(1);
      versionData.setHash(crc);
      LoggerService.registerSuccess(GeneralNode.ipAddress,"Inregistram prima
versiune a fisierului " + filename);
    }
    else{
      VersionData lastVersionOfFile = getLastVersionOfFile(metadataFilePath);
      versionData.setVersionNumber(lastVersionOfFile.getVersionNumber() + 1);
      if(crc == -1){
        versionData.setHash(lastVersionOfFile.getHash());
      }
      else{
        versionData.setHash(crc);
      }
      LoggerService.registerSuccess(GeneralNode.ipAddress,"Inregistram o noua
versiune a fisierului " + filename);
    }
    versionData.setTimestamp(Time.getCurrentTimeWithFormat());
    versionData.setDescription(description);
    FileSystem.appendToFile(metadataFilePath, versionData.toString() + "\n");
  }

  public VersionData getLastVersionOfFile(String metadataFilePath){
    try {
      List<String> metadataContent = FileSystem.getFileLines(metadataFilePath);
      String lastRegister = metadataContent.get(metadataContent.size() - 1);
      return new VersionData(lastRegister);
    }
    catch (IOException exception){
      LoggerService.registerError(GeneralNode.ipAddress,"Eroare la extragerea
continutului fisierului de metadate " + metadataFilePath);
      return null;
    }
  }

  public VersionData getLastVersionOfFile(String userId, String filename){
```

```
    String filepath = baseFilepath + userId + "\\" + filename;
    String metadataFilePath = filepath.substring(0, filepath.lastIndexOf(".")) +
extension;
    return getLastVersionOfFile(metadataFilePath);
  }

  public List<VersionData> getVersionsForFile(String userId, String filename)
throws IOException {
    String filepath = baseFilepath + userId + "\\" + filename;
    String metadataFilePath = filepath.substring(0, filepath.lastIndexOf(".")) +
extension;
    List<VersionData> versionData = new ArrayList<VersionData>();
    for(String versionLine : FileSystem.getFileLines(metadataFilePath)){
      versionData.add(new VersionData(versionLine));
    }
    return versionData;
  }
}
```

Anexa 12. Codul clasei FileSender din modulul FrontendProxy

```
public class FileSender {
  private static int bufferSize =
Integer.parseInt(AppConfig.getParam("buffersize"));
  private static int generalManagerPort =
Integer.parseInt(AppConfig.getParam("generalManagerPort"));
  private static String generalManagerAddress =
AppConfig.getParam("generalManagerAddress");
  private static int feedbackPort =
Integer.parseInt(AppConfig.getParam("feedbackport"));
  private static String downloadFilePath = AppConfig.getParam("filedownloadpath");

  private static boolean validateToken(String token) throws Exception{[...]}

  private static String[] getAddressesFromToken(String token) throws
Exception{[...]}

  public static void sendFile(ClientManagerRequest clientManagerRequest, String
token){
    try {
      String destinationAddress = getAddressesFromToken(token)[0];
      Socket socket = new Socket(destinationAddress, generalManagerPort);
      FileHeader fileHeader = new FileHeader();
      fileHeader.setFilename(clientManagerRequest.getFilename());
      fileHeader.setToken(token);
      fileHeader.setFilesize(new
File(clientManagerRequest.getFilename()).length());
      fileHeader.setUserId(clientManagerRequest.getUserId());
      fileHeader.setDescription(clientManagerRequest.getDescription());
      DataOutputStream outputStream = new
```

```
DataOutputStream(socket.getOutputStream());
      BufferedInputStream inputStream = new BufferedInputStream(new
FileInputStream(clientManagerRequest.getFilename()));
      outputStream.write(Serializer.serialize(fileHeader));
      byte[] binaryFile = new byte[bufferSize];
      int count;
      while ((count = inputStream.read(binaryFile)) > 0) {
        outputStream.write(binaryFile, 0, count);
      }
      inputStream.close();
      outputStream.close();
      socket.close();
    }
    catch (Exception exception){
      System.out.println("Exceptie la trimiterea unui nou fisier: " +
exception.getMessage());
    }
  }

  public static void waitForFeedback(String userId, String token, String filename,
long timeout, long CRC) throws Exception {
    int total_nodes = 0;
    int received_nodes = 0;
    final int[] valid_nodes = {0};
    boolean another_exception = false;
    String[] fnamelist = filename.split("\\\\");
    final String fname = fnamelist[fnamelist.length - 1];
    final List<Thread> threads = new ArrayList<>();
    try {
      final List<String> addresses = new
LinkedList<String>(Arrays.asList(getAddressesFromToken(token)));
      total_nodes = addresses.size();
      while(received_nodes != total_nodes){
        received_nodes += 1;
        Thread thread = new Thread(new Runnable() {
          @Override
          public void run() {
            try{
              NewFileRequestFeedback feedback;
              while ((feedback =
FrontendProxyUiApplication.feedbackManager.getFeedback(userId, fname)) == null);
              String nodeAddress = feedback.getNodeAddress();
              String fileName = feedback.getFilename();
              String userID = feedback.getUserId();
              long crc = feedback.getCrc();
              if (fileName.equals(fname) && userID.equals(userId) && CRC == crc) {
                valid_nodes[0] += 1;
              }
            }
            catch (Exception exception){
              System.out.println("Exceptie la primirea feedback-ului! : " +
```

```
exception.getMessage());
            }
          }
        });
        threads.add(thread);
        thread.start();
      }
    }
    catch (Exception exception){
      another_exception = true;
    }
    finally {
      for(Thread thread : threads){
        if(thread.isAlive())
          thread.join();
      }
      if(another_exception || received_nodes == 0 || valid_nodes[0] == 0) {
        sendFeedbackToGM(userId, fname, "ERROR");
        return;
      }
      if(valid_nodes[0] >= 1){
        sendFeedbackToGM(userId, fname, "OK");
        FileController.uploadPendingQueue.addToQueue(userId, filename);
      }
    }
  }

  public static void sendFeedbackToGM(String userId, String filename, String
status){[...]}

  public static String downloadFile(String destionationAddress, String userId,
String filename){
    try {
      DownloadFileRequest downloadFileRequest = new DownloadFileRequest();
      downloadFileRequest.setFilename(filename);
      downloadFileRequest.setUserId(userId);
      Socket socket = new Socket(destionationAddress, generalManagerPort);
      DataOutputStream outputStream = new
DataOutputStream(socket.getOutputStream());
      outputStream.write(Serializer.serialize(downloadFileRequest));
      String filepath = downloadFilePath + filename;
      FileOutputStream fileOutputStream = new FileOutputStream(filepath);
      InputStream dataInputStream = new DataInputStream(socket.getInputStream());
      byte[] binaryFile = new byte[bufferSize];
      int count;
      while ((count = dataInputStream.read(binaryFile)) > 0) {
        fileOutputStream.write(binaryFile, 0, count);
      }
      fileOutputStream.close();
      dataInputStream.close();
```

```
      outputStream.close();
      socket.close();
      return "/buffer/" + filename;
    }
    catch (Exception exception){
      return null;
    }}
}
```

Anexa 13. Codul clasei FrontendManager din modulul FrontendProxy

```
public class FrontendManager {
  private static int bufferSize =
Integer.parseInt(AppConfig.getParam("buffersize"));
  private static String generalManagerAddress =
AppConfig.getParam("generalManagerAddress");
  private static int generalManagerPort =
Integer.parseInt(AppConfig.getParam("generalManagerPort"));

  public static ManagerResponse managerOperationRequest(ClientManagerRequest
clientRequest) throws NullPointerException, IOException, ClassNotFoundException {
    Class operation = clientRequest.getClass();
    List<Class> specialOps = Arrays.asList(new Class[]{
      NewFileRequest.class, DeleteFileRequest.class, RenameFileRequest.class
    });
    if(specialOps.contains(operation))  {
      String[] fname = clientRequest.getFilename().split("\\\\");
      clientRequest.setFilename(fname[fname.length - 1]);
    }
    DataInputStream socketInputStream = null;
    DataOutputStream socketOutputStream = null;
    Socket generalManagerSocket = null;
    try {
      generalManagerSocket = new Socket(generalManagerAddress,
generalManagerPort);
      socketOutputStream = new
DataOutputStream(generalManagerSocket.getOutputStream());
      socketOutputStream.write(Serializer.serialize(clientRequest));
      socketInputStream = new
DataInputStream(generalManagerSocket.getInputStream());
      byte[] buffer = new byte[bufferSize];
      ManagerResponse userResponse = null;
      List<Class<? extends ClientManagerRequest>> complexeGetOperations =
Arrays.asList(GetUserFiles.class,GetUserFileHistory.class,
GetContentTableRequest.class,GetNodesForFileRequest.class,GetNodesStorageQuantityR
equest.class, GetStorageStatusRequest.class, GetReplicationStatusRequest.class,
GetConnectionTableRequest.class);
      while(socketInputStream.read(buffer, 0, bufferSize) > 0){
        boolean found_complexe = false;
        for(Class<? extends ClientManagerRequest> complexeOp :
```

```
complexeGetOperations){
          if(complexeOp == operation){
            userResponse = (ManagerComplexeResponse)
Serializer.deserialize(buffer);
            found_complexe = true;
            break;
          }
        }
        if(!found_complexe)
          userResponse = (ManagerTextResponse) Serializer.deserialize(buffer);
        break;
      }
      socketInputStream.close();
      socketOutputStream.close();
      generalManagerSocket.close();
      return userResponse;
    }
    catch (NullPointerException | ClassNotFoundException exception){
      socketInputStream.close();
      socketOutputStream.close();
      generalManagerSocket.close();
      if(exception.getClass() == NullPointerException.class)
        throw exception;
    }
    return null;
  }

  public static void mainActivity(ClientManagerRequest requestData){
    new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          String fullPath = requestData.getFilename();
          ManagerResponse response = managerOperationRequest(requestData);
          Class<? extends ClientManagerRequest> operation =
requestData.getClass();
          if(operation == NewFileRequest.class){
            String token = ((ManagerTextResponse)response).getResponse();
            requestData.setFilename(fullPath);
            long start = System.currentTimeMillis();
            FileSender.sendFile(requestData, token);
            long timeElapsed = System.currentTimeMillis() - start;
            FileSender.waitForFeedback(requestData.getUserId(), token,
requestData.getFilename(), timeElapsed, ((NewFileRequest) requestData).getCrc());
            FileSystem.deleteFile(requestData.getFilename());
          }
        }
        catch (Exception exception){
          System.out.println("Exceptie : " + exception.getMessage());
        }
```

```
      }
    }).start();
  }
}
```

Anexa 14. Codul React ce conține structuri condiționale, folosind operatorul ternar

```
{this.state.isUserConnected === true ?
    <div className="profile">
        <div className="profile_left">
            <img id="log_data_profile" src="images/user_logo.png" alt=""/>
            <p id="username">{this.userData.name}</p>
            <hr/>
            {generalData}
        </div>
        <div className="profile_right">
            <ul>
                <li><button className="a_redirector" id="selector_storage_status"
href="#" onClick={() => {
    this.setState({userDetailsCategory: ProfilePage.userDetailsCategories[0]})
}}>Storage Status</button>
                </li>
                <li><button className="a_redirector" id="selector_plan" href="#"
onClick={() => {
    this.setState({userDetailsCategory: ProfilePage.userDetailsCategories[1]})
}}>Plan</button>
                </li>
            </ul>
            <hr/>
            {userDetails}
        </div>
    </div>
:
    <p>Nu puteti accesa aceasta pagina daca utilizatorul nu este conectat</p>
}
```

Anexa 15. Codul React ce conține definirea dinamică a elementelor

```
 if(this.state.additionalUserData !== null){
     generalData =
         <div className = "accountData">
             <p className="accountDataField" >
                 Name
                 <span className = "accountDataValue">
                     {this.state.additionalUserData["name"]}
                 </span>
             </p>
             <p className="accountDataField" >
```

```
        […]
        </div>
    }
    else{
         userDetails = <p>Fetching user data..</p>
    }

    return (
        <div className="App">
           <div className="title">
              <img id="title_logo" src="images/logo.png" alt="Logo not found"/>
              <label id="title_text">Safestorage</label>
           </div>
           <hr/>
           {this.state.isUserConnected === true ?
               <div className="profile">
                   <div className="profile_left">
                       <p id="username">{this.userData.name}</p>
                       <hr/>
                       {generalData}
                   </div>
                   […]
```