# Cuprins

# Capitolul 1

# Introducere

## 1.1 Descrierea temei

În această eră a vitezei şi a informaţiei se pune foarte mult accentul pe acumularea şi partajarea resurselor pentru a putea avea o putere computaţională mare. Astfel au apărut sistemele distribuite, şi odată cu ele şi necesitatea de transfer de date între nodurile sistemului.

În această lucrare este analizată situaţia în care se doreşte replicarea unor date pe mai multe noduri destinaţie din sistem. Deşi pare o problemă banală la prima privire, ceea ce face această sarcină dificilă este faptul că numărul de noduri dintr-un sistem distribuit este de ordinul zecilor şi chiar sutelor. De asemenea, volumul datelor de transferat poate fi foarte mare, de ordinul GB sau chiar TB, făcând din această problemă o adevărată provocare.

Aplicaţia propusă în lucrarea de faţă reuşeşte să combată aceste probleme prin folosirea unor topologii ce implică divizarea nodurilor în grupuri mici şi autonome de replicare şi prin implementarea unor algoritmi eficienţi pentru procesarea datelor ce împiedică apariţia erorilor de memorie cauzate de volumele mari de date.

## 1.2 Contextul proiectului

Distribuirea rapidă şi sigură a datelor de la o singură sursă la un număr mare de destinaţii localizate într-un sistem de mari dimensiuni a reprezentat un subiect intensiv cercetat pe parcursul ultimului deceniu. Această cercetare are aplicabilitate în contextul reţelelor cu distribuţie de conţinut (*en.* Content Delivery Networks), dar poate fi utilizată şi în alte domenii. O reţea CDN reprezintă un sistem distribuit de servere de dimensiuni mari desfăşurate în mai multe centre de date dispersate geografic. Scopul unui CDN este de a asigura un grad înalt de disponibilitate şi de performanţă a serviciilor oferite. Se foloseşte un set dedicat de servere pentru a distribui conţinut clienţilor din partea serverului original pe care se afla datele cerute. În acest scop, conţinutul este replicat pe aceste servere, mai

apropiate de utilizator, îmbunătăţind astfel latenţa şi reducând în acelaşi timp traficul din reţea.

Din punctul de vedere al utilizatorului, este irelevant de pe ce server îi sunt aduse datele, cu condiţia ca acestea să fie prezentate în timp util. Rolul unui CDN devine astfel cel de a găsi cea mai bună soluţie pentru fiecare utilizator, în funcţie de locaţia acestuia. Pentru a putea oferi aceeaşi calitate a serviciilor tuturor utilizatorilor, este necesar ca datele să fie replicate pe cât mai multe servere dispersate geografic. În această lucrare vom vedea diferite abordări ale acestei probleme şi soluţiile propuse pentru o replicare cât mai eficientă şi adaptivă. Deşi punctul de plecare al lucrării sunt reţelele cu distribuţie de conţinut, algoritmii propuşi aici pot fi utilizaţi în orice tip de sistem distribuit ce necesită distribuirea unor volume mari de date la mai multe destinaţii.

## 1.3 Metoda abordată

Pentru scalarea aplicaţiei la o multitudine de noduri destinaţie, abordarea folosită aici este de a le împărţi în grupuri relativ mici, independente unul de celălalt. Comunicaţia se face doar în interiorul unui grup de replicare, pentru a menţine numărul de conxiuni la un nivel acceptabil. Fiecare grup este responsabil pentru nodurile sale, iar protocoalele de comunicare asigură replicarea fiabilă a datelor.

Pentru a putea obţine transferuri de dimensiuni mari, un algoritm de management al memoriei este responsabil pentru procesarea datelor la primirea lor. Deoarece una dintre cerinţe este eficientizarea timpilor de transfer, aplicaţia se adaptează la specificaţiile reţelei şi la variaţiile parametrilor.

## 1.4 Motivaţie

Pentru a motiva alegerea acestei teme, vom analiza un exemplu concret de utilizare. Să luăm în considerare existenţa unui sistem distribuit cu sute sau chiar mii de servere distribuite geografic. Presupunem că un utilizator doreşte transferul unui fişier de dimensiuni mari de pe un server sursă pe mai multe servere destinaţie. Transferul acestui fişier folosind conexiuni punct-la-punct (*en.* P2P = Peer to Peer) între sursă şi fiecare dintre destinaţii reprezintă o abordare naivă şi prezintă limitări de performanţă. După cum se precizează şi în [1], o primă problemă este risipirea consumului de lăţime de bandă. O altă problemă generată de această abordare este faptul că fiecare rata de transfer pentru fiecare destinaţie este limitată de caracteristicile conexiunii P2P.

Mai precis, dacă între serverul sursă S şi una dintre destinaţii A există o cale cu viteza de transfer $v_{SA}$, este posibil să existe un alt server destinaţie B, astfel încât calea la serverul A ce trece prin B sa aibă o viteză de transfer mai mică decât cea directă: $v_{SA} > (v_{SB} + v_{BA})$. În acest caz, este mai eficient ca transferul pe serverul A să se facă prin intermediul serverului B.

Chiar dacă am considera căile de la serverul sursă la destinaţii ca fiind cele mai eficiente, avem două noi probleme:

- În cazul în care sursa deschide conexiuni la toate destinaţiile, poate să apară o problemă de suprasarcină, deoarece numărul de conexiuni suportate de un server este limitat. Aceasta poate rezulta în închiderea unor conexiuni, pentru a putea păstra altele deschise. Lăţimea de bandă este împărţită între conexiuni, crescând timpul total de transfer.
- În cazul în care sursa deschide conexiuniunile la destinaţii în mod serial, adică fiecare nouă conexiune este deschisă după închiderea precedentei, fiecare server destinaţie trebuie să aştepte ca transferul la serverele care sunt înaintea lui în lista de transfer sa primească fişierul. Lăţimea de bandă poate să nu fie folosită în totalitate. Şi în acest caz, timpul de transfer este mare, deoarece reprezintă suma tuturor timpilor de transfer către fiecare destinaţie.

## 1.5 Structura lucrării

Structura următoarelor capitole este:

- *Capitol 2: Obiective şi specificaţii* | în acest capitol sunt descrise obiectivele propuse şi sunt detaliate spcificaţiile funcţionale ale problemei prezentate în lucrarea de faţă.
- *Capitol 3: Studiu bibliografic* | conţine prezentarea şi analiza altor lucrări ce tratează această problemă. Fiecare abordare este detaliată, pentru a se putea observa avantajele şi dezavantajele fiecăreia.
- *Capitol 4: Fundamentare teoretică* | în acest capitol sunt descrise cunoştinţele necesare pentru ca cititorul să înţeleagă această lucrare din punct de vedere teoretic.
- *Capitol 5: Analiză şi design* | această parte prezintă schema propusă ca şi soluţie, descriind funcţionalitatea fiecărui modul al acesteia.
- *Capitol 6: Proiectare de detaliu şi implementare* | conţine informaţii detaliate despre implementare.
- *Capitol 7: Testare şi validare* | prezintă testele efectuate pentru a demonstra funcţionalitatea aplicaţiei şi cele pentru a determina eficienţa acesteia.
- *Capitol 8: Manual de utilizare* | conţine detaliile legate de instalarea şi utilizarea aplicaţiei.
- *Capitol 9: Concluzii* | această secţiune conţine concluziile lucrării, precum şi sugestii pentru dezvoltări ulterioare.

# Capitolul 2

# Obiective şi specificaţii

## 2.1 Obiectivele proiectului

Înainte de a defini specificaţiile funcţionale ale acestei lucrări, trebuie stabilit un set concret de obiective. Scopul este ca în final aplicaţia să îndeplinească toate obiectivele descrise în acest capitol. Obiectivele propuse sunt descrise în ordine cronologică, după cum urmează:

**Studierea domeniului**

Acest pas implică un studiu bibliografic intensiv în domeniul sistemelor distribuite, şi mai precis al transferului volumelor mari de date. Principalele etape ale acestui pas sunt: studiul reţelelor cu distribuţie de conţinut, studiul transferurilor volumelor mari de date în sisteme distribuite, studiul diferitelor topologii şi reţele, studierea şi întelegerea unei reţele de distribuţie de tip multicast, studiul implementărilor existente şi înţelegerea acestora.

**Analiza şi compararea implementărilor existente**

După familiarizarea cu domeniul temei şi înţelegerea diferitelor implementări existente, urmează o analiză a acestora, pentru a determina direcţia de urmat. Fiecare lucrare existentă are avantaje şi dezavantaje, iar scopul acestui pas este de a pune în balanţă caracteristicile fiecăreia pentru a le alege pe cele mai bune în propria implementare.

**Dezvoltarea de topologii**

Analiza topologiilor existente rezultă în generarea unor noi structuri ce combină şi balansează avantajele descrise mai sus. Aceste topologii sunt realizate între un nod sursă şi minim două noduri destinaţie. Scopul este de a reuşi generarea unor topologii eficiente pentru un număr mare de servere destinaţie, foarte dispersate geografic, şi pentru transferul datelor de dimensiuni foarte mari. Pentru a diminua timpul total de transfer trebuie limitată comunicarea dintre noduri, dar în acelaşi timp trebuie ţinut cont de faptul că adaptabilitatea se poate face doar prin comunicare.

**Dezvoltarea algoritmilor de transfer**
Generarea topologiilor este primul pas în obţinerea unui transfer eficient, dar este foarte important şi cum sunt transferate datele. Nodul sursă este foarte important în această etapă, deoarece reprezintă primul pas de distribuţie.

**Implementarea diferiţilor algoritmi**
Acest pas este doar de implementare a algoritmilor găsiţi anterior, urmărind topologiile studiate. Algoritmii pot fi modificaţi în cazul apariţiei unor limitări de implementare. De asemenea, în acest pas se pot folosi cunoştinţe legate de mediul de implementare pentru a optimiza procesele de transfer.

**Testare şi comparare**
În timpul analizei se fac diverse calcule pentru a determina timpii de transfer. În acest pas se face validarea acestor calcule, şi se compară diferitele implementări, atât din punct de vedere al funcţionalităţii, cât şi al eficienţei. Tot în acest pas trebuie determinate limitările pentru fiecare soluţie: numărul maxim de noduri destinaţie, dimensiunea maximă a datelor ce pot fi transferate (fără a se lua în considerare capacitatea maximă a serverelor, care se presupune a fi infinită).

**Analiza rezultatelor**
Acest pas presupune o retrospectivă asupra muncii efectuate. Se analizează implementările făcute, cu avantajele şi dezavantajele lor şi se ajunge la un set de concluzii.

## 2.2 Specificaţia temei

Date de intrare:

**Fişier** Notaţia generică de fişier se referă la un volum mare de date existent pe un sistem de calcul, numit sursă.

**Nod sursă** Noţiunea de nod se referă la o componentă a unui sistem distribuit, iar sursă la existenţa fişierului pe acesta.

**Noduri destinaţie** Tot componente ale sistemului distribuit, dar care nu posedă fişierul.

**Client** Componentă pasivă a sistemului distribuit, deoarece face parte din acesta doar pentru un scurt timp.

Cerinţe funcţionale:

- proiectarea unui sistem distribuit care conţine toate elementele descrise mai sus.
- proiectarea unei arhitecturi de reţea pentru conectarea componentelor.
- dezvoltarea de protocoale de comunicare între componente.

- dezvoltarea de protocoale de transfer între componentele active ale sistemului.
- dezvoltarea de protocoale de replicare între componentele active ale sistemului.

Cerinţe non-funcţionale:

- scalabilitate – din punct de vedere al mărimii fişierului şi al numărului de noduri destinaţie.
- eficienţă – din punct de vedere al timpului de replicare.
- adaptabilitate – din punct de vedere al lăţimii de bandă de pe conexiuni şi al dimensiunii pachetelor de date.
- fiabilitate – din punct de vedere al nodurilor pe care se efectuează replicarea cu succes.

# Capitolul 3

# Studiu bibliografic

În acest capitol sunt analizate în detaliu alte lucrări, articole sau proiecte care implementează conceptele prezentate în lucrarea de faţă. Acestea au atât rolul de a informa în legătură cu alte soluţii existente, cât şi de a crea o bază teoretică şi comparativă pentru implementarea proprie.

## 3.1 Algoritmii Fast Replica

### 3.1.1 Fast Replica

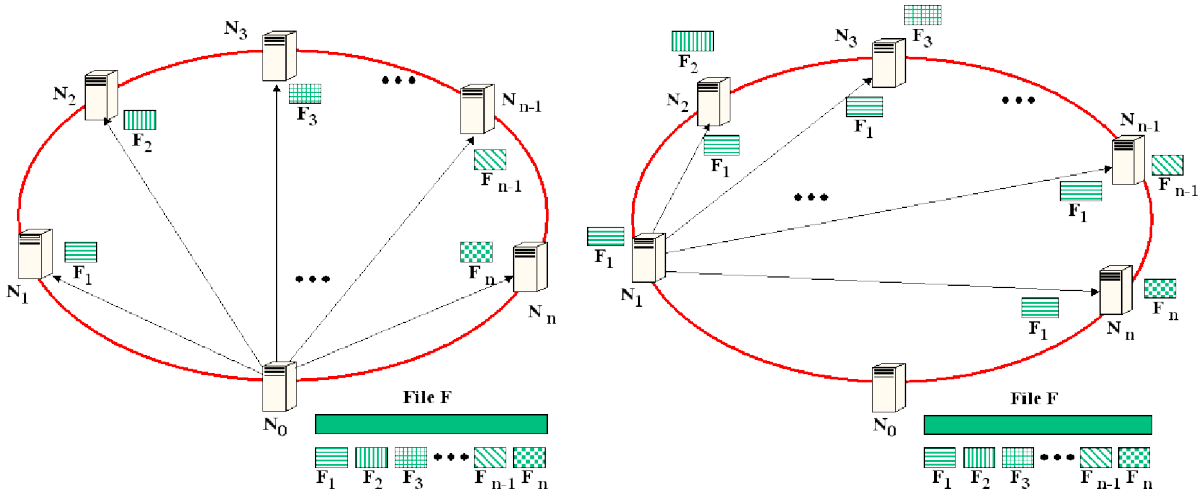
În [2] este prezentat algoritmul *FastReplica*, un algoritm de distribuire prin tehnica de împingere a datelor (*en. push*). Aici se consideră, ca şi în lucrarea de faţă, o reţea distribuită de mari dimensiuni şi problema distribuirii de conţinut în această reţea. În aceasta se găsesc:

- motivele pentru necesitatea unui asemenea algoritm – este vorba despre reţelele de distribuire de conţinut CDN în care datele sunt replicate pentru a se evita supraîncărcarea unui server şi blocajele în reţea.
- primele 3 cele mai populare metode de distribuire a conţinutului în Internet: distribuire prin satelit, distribuire multicast, distribuire multicast la nivel de aplicaţie.
- descrierea algoritmului pentru două tipuri de reţele: mici (10-30 noduri) - *Fast Replica in the small* şi mari (sute de noduri) - *Fast Replica in the large*
- extinderea algoritmului pentru a asigura transferuri de încredere (*en. reliable*).
- evaluarea performanţelor şi compararea cu calculele precedente.

Există câteva idei de bază exploatate în [2]. Pentru a replica un fişier mare pe n noduri (n este în intervalul de 10-30 noduri), fişierul original este împărţit în n sub-fişiere

de dimensiuni egale şi fiecare sub-fişier este transferat la un nod diferit din grup. După aceea, fiecare nod îşi propagă sub-fişierul celorlalte noduri din grup. Astfel, în loc de replicarea obişnuită a unui fişier întreg pe n noduri, cu ajutorul a $n$ căi, conectând nodul original la întregul grup de replicare, FastReplica exploatează $n * n$ căi din cadrul grupului de replicare unde fiecare cale este utilizată pentru transferul a $1/n$ din fişier. Scopul este de a proiecta un algoritm scalabil şi fiabil, care poate fi folosit pentru replicarea fişierelor de dimensiuni mari la un grup mare de noduri.

În figura 3.1 se pot observa cei doi paşi ai algoritmului *FastReplica*: distribuţie – 3.1(a) şi colecţie – 3.1(b).




(a) FastReplica - pasul de distribuţie

(b) FastReplica - pasul de colecţie

Figura 3.1: Ilustrare algoritm FastReplica

După cum se poate vedea, algoritmul *FastReplica* prezintă o multitudine de avantaje:

- timpul de replicare al fişierului este minimizat prin folosirea tuturor căilor disponibile în reţea: $Timp_{mediu} = \frac{1}{n} \sum_{i=1}^{n} Timp^i(F)$ unde $Timp^i(F)$ reprezintă timpul de transfer al fişierului $F$ de la nodul sursă la nodul destinaţie $N_i$, iar $n$ reprezintă numărul total de servere destinaţie.
- replicarea se face în doar doi paşi: distribuţie şi colecţie.
- modelul este simplu şi uşor de înţeles, ceea ce îl face şi uşor de implementat.
- algoritmul poate fi generalizat la un număr mult mai mare de noduri prin formarea aşa-ziselor grupuri de replicare, utlizând o metodă iterativă.
- algoritmul este extins pentru a detecta şi a corecta eventualele defecte ale nodurilor apărute atât înaintea începerii, cât şi în timpul transferului.

Deşi aceste avantaje fac *FastReplica* un model foarte folosit, studiul acestuia a scos la iveală şi anumite dezavantaje:

- în timpul pasului de distribuţie, nodul sursă $N_0$ are $n-1$ conexiuni de ieşire, una către fiecare nod din grupul de replicare. Pentru valori mari ale lui $n$, numărul total de conexiuni poate cauza probleme.

- în timpul pasului de colecţie, fiecare nod $N_i$ are următoarele conexiuni: $n-1$ conexiuni de intrare şi $n-1$ conexiuni de ieşire, totalizând $2n-2$ conexiuni pentru fiecare nod destinaţie. Pentru valori mari ale lui $n$, numărul total de conexiuni poate cauza probleme.

- conexiunile din pasul de distribuţie nu sunt folosite la capacitatea lor maximă, deoarece după terminarea acestui pas, pe aceste conexiuni nu se mai transferă date utile către serverele destinaţie.

- nodul sursă trebuie să trimită fiecărui nod destinaţie adresele celorlalte noduri, pentru ca acesta să se poată conecta la ele. Astfel, comunicarea dintre noduri creşte exponenţial cu numărul total de destinaţii.

### 3.1.2 ALM-FR

În [3] ideile *FastReplica* sunt combinate cu tehnica nivelului de aplicaţie multicast (*en Application Level Multicast*) pentru optimizarea replicării fişierelor mari în cadrul CDN. Această metodă este denumită *ALM-FR*. Un transfer de fişiere în algoritmul clasic *FastReplica* urmează un mod de distribuire de tip stocare şi înaintere, adică doar după ce un sub-fişier este primit de către un nod, acest nod începe să transmită sub-fişierul corespunzător nodurile ulterioare. în acest model, atunci când apare un nod defect, impactul se propagă doar la nodurile din grupul local de replicare. Algoritmul de recuperare este oarecum natural şi simplu: aceasta implică doar nodurile din grupul replicare corespunzător.

Deoarece tehnica de multicast IP nu beneficiază de o implementare larg răspândită, multe eforturi industriale şi de cercetare şi-au redirecţionat atenţia către investigarea şi implementarea multicastului la nivelul aplicaţiei, în care nodurile distribuite pe internet acţionează ca routere intermediare pentru a distribui eficient conţinutul de-a lungul unui arbore predefinit. Mai detaliat, această lucrare propune împarţirea setului original de noduri în grupurile de replicare $G_0, G_1, ...G_{k1}$, fiecare format din $k$ noduri. Aceste grupuri sunt ordonate în arborii speciali de multicast $\hat{M}, M^1, M^2, ...M^{m1}$. Fiecare dintre aceşti arbori este format din maxim $m$ grupuri, unde $\hat{M}$ reprezintă arborele multicast primar, iar $M^1, M^2, ...M^{m1}$ reprezintă arborii multicast secundari. Schema rezultată poate fi analizată în figura 3.2.

Fişierul este divizat, ca şi în algoritmul original, în $k$ sub-fişiere egale ca dimensiuni $F_0, F_1, ...F_k$. Diferenţa apare în paşii de execuţie:

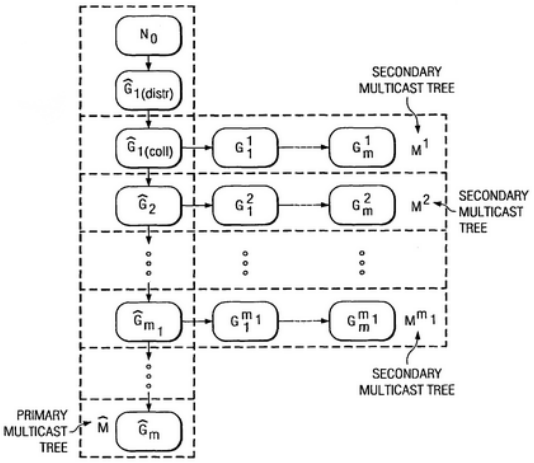



Figura 3.2: ALM-FastReplica: schema generală

- pasul de distribuţie: bucăţile de fişier $F_0, F_1, ...F_k$ sunt trimise de către nodul iniţiator $N_0$ către nodurile din grupul $\hat{G_1}$, ce aparţine arborelui primar de multicast, fiecare nod primind câte un sub-fişier.

- pasul de colecţie: în cadrul grupului $\hat{G_1}$, nodurile încep să deschidă conexiuni pentru a transfera bucăţile de fişier primite de la nodul iniţiator. La finalul acestui pas, toate nodurile din grupul $\hat{G_1}$ vor avea fişierul în totalitate.

- pasul de comunicare de grup: comunicarea între grupul $\hat{G_1}$ şi $\hat{G_2}$ se desfăşoară urmând un protocol diferit: fiecare nod $N_i^1$ din grupul $\hat{G_1}$ trimite sub-fişierul $F_i$ către nodul $N_i^2$ din grupul $\hat{G_2}$ imediat ce îl primeşte. După aceea, grupul $\hat{G_2}$ începe pasul de colecţie, reluând algoritmul până când toate grupurile au primit întreg fişierul.

## 3.2 GridFTP

### 3.2.1 Protocolul GridFTP

Una dintre marile probleme din HPC (*en. High Power Computing*) este aceea de a muta fişiere de mari dimensiuni (GB, chiar şi TB) între diverse locaţii. Mecanismele simple de transfer precum FTP(*en. File Transfer Protocol*) şi SCP(*en. Secure Copy* ) nu sunt suficiente atât din perspectiva fiabilităţii cât nici din cea a performanţei. [4] GridFTP este un protocol de transfer de date bazat pe FTP, optimizat pentru folosirea în reţele WAN cu lăţime mare de bandă. Este un protocol de înaltă performanţă, sigur şi fiabil, şi în ultimii ani a devenit foarte popular pentru transferurile volumelor mari de date. Printre caracteristicile cheie ale acestui protocol se numără:

- performanţa – acesta are suport inclus pentru utilizarea conexiunilor TCP paralele şi a transferurilor multiple pentru a atinge rate de transfer mari.

- puncte de control – aceştea sunt utile pentru a relua un transfer eşuat doar de la un anumit punct.

- transferuri de la terţi – protocolul FTP, pe care se bazează GridFTP, are canale separate de control şi de date, permiţând transferuri de la terţi, adică transferuri între două noduri, mediate de un al treilea ce acţionează ca şi client.

- securitate – oferă securitate pe ambele canale. Canalul de control este criptat în mod implicit, deoarece este canalul pe care se face prima dată comunicarea, iar canalul de date foloseşte autentificare, cu posibilitatea de a avea şi protecţia integrităţii şi criptare.

### 3.2.2 Globus GridFTP

Globus Toolkit® este un *framework open source* pentru construirea de reţele de calcul, care oferă o implementare vastă şi actuală a protocolului GridFTP. Acest *framework* oferă:

- o implementare pentru server (*globus-gridftp-server*).

- o implementare pentru client bazată pe scripturi ( *globus-url-copy*).

- un set de biblioteci pentru dezvoltarea de clienţi noi.

În [5] şi [6] este prezentată munca cercetătorilor de la Argonne National Laboratory (ANL) şi de la Ohio State Univerity (OSU) legată de această temă şi pusă în aplicare folosind Globus GridFTP. În [5] sunt prezentaţi algoritmi dinamici ce efectuează transferul simultan al bucăţilor de fişier din multiple replici ale acestuia, localizate în diverse surse. Algoritmul este adaptiv, atât în selectarea surselor, cât şi în schimbarea dinamică a lăţimii de bandă din reţea. Articolul [6] propune o altă modalitate de implementare a conceptelor GrifFTP, prin generarea de reţele de acoperire (*en. overlay*). Beneficiile oferite de aceste tipuri de reţele sunt evidenţiate printr-o arhitectură ce include două optimizări cheie:

- *multi-hop path splitting* – înlocuirea unei conexiuni directe între sursă şi destinaţie cu o cale multi-hop, înlănţuită prin noduri intermediare.

- *multi-pathing* – împarţirea datelor la sursă şi trimiterea acestora pe multiple căi obţinute folosind pasul anterior. Altfel spus, mai multe rute independente pot fi folosite pentru a transfera simultan bucăţile de fişier la destinaţie.

### GridFTP Multicast

Implementarea Globus GridFTP oferă posibilitatea de a trimite pachete de date la mai multe destinaţii. Pentru aceasta, este necesar să se includă în stiva de drivere modulul de multicast. Atunci când un pachet ajunge la un server configurat în acest fel, el trece prima dată prin modulul de multicast, care îl transmite pe alte căi, la următoarele servere din structură( 3.3(b), 3.3(c), 3.3(d)). Arhitectura GridFTP generează o topologie de tip arbore binar, după cum se poate vedea şi în figura 3.3(a). Nodul sursă reprezintă rădăcina acestui arbore, iar nodurile destinaţii fii acestuia. [7]

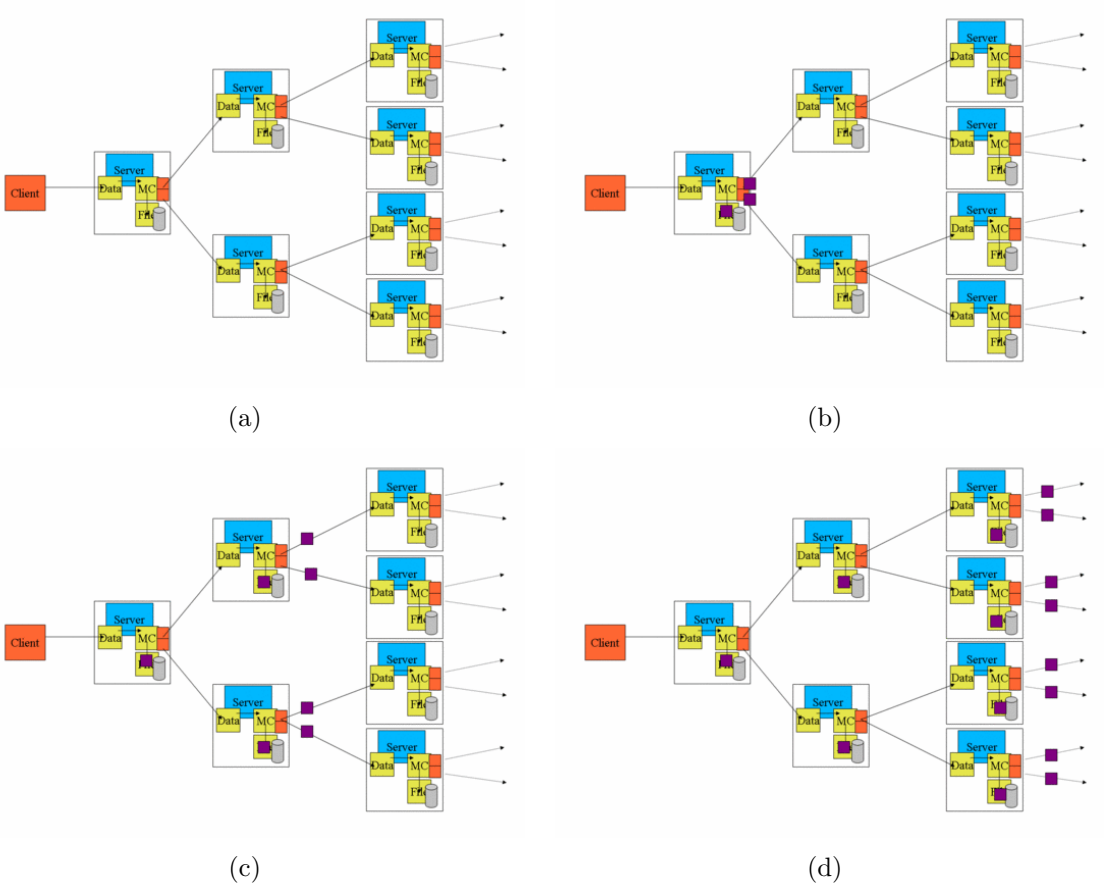



Figura 3.3: GridFTP multicast - schema generală pe paşi [7]

Avantajele acestei abordări:

- Are o structură simplă, uşor de implementat şi de utilizat. Structura de arbore binar se scalează bine pentru un număr mare de noduri destinaţie, deoarece numărul de conexiuni pentru fiecare nod rămâne maxim $N$.

- Are o arhitectură modularizată, ce permite atât utilizarea în combinaţie cu alte drivere existente, cât şi extinderea prin implementarea de noi drivere.
- Timpul maxim de transfer este eficientizat prin faptul că fiecare pachet este trimis mai departe imediat ce este primit.
- Oferă posibilitatea de a creşte numărul de fii $N$ pentru a obţine performanţe ridicate.

Dezavantaje:

- Pentru valori ale lui $N$ egale cu numărul de noduri destinaţie, nu există nicio diferenţă între această topologie şi o transmitere în serie a datelor.
- Volumul de date de control (adresele nodurilor, dimensiuni fişier) trimise pe legături este mare. De exemplu, primele două noduri din ierarhie trebuie să primească date despre $N/2$ servere fiecare.
- Serverele aflate pe poziţiile frunze vor primi date care trec prin toţi strămoşii acestora. În această trecere, există posibilitatea deteriorării sau a pierderii complete a datelor.
- Soluţia nu este fiabilă: în cazul în care unul dintre serverele din ierarhie eşuează în timpul transferului, nici unul dintre nodurile din subarborele acestuia nu va mai primi pachetele necesare.

### GridFTP Striped

În cadrul proiectului Globus GridFTP a apărut şi un *framework* numit GridFTP Striped. Ideile de bază ce au stat la apariţia acestuia sunt descrise în [8]. Globus GridFTP Striped reprezintă un set de biblioteci pentru client şi server, concepute pentru construirea de aplicaţii de date. Acesta foloseşte ideea că sursele de date sunt de multe ori clustere şi organizează datele într-o structură adaptată. Astfel, datele sunt ţinute pe mai multe servere care au un sistem comun de fişiere. Această organizare este comandată de faptul că uneori un server poate fi format din mai multe dispozitive fizice, care pot comunica între ele. Există două tipuri de noduri:

- *front end node* – acesta este nodul cu care se realizează conexiuni, cel care reprezintă celelalte noduri. Doar prin acesta se pot accesa datele, dar pe maşina fizică ce conţine acest nod nu se află date, ci doar informaţii despre nodurile de date.
- *back end node* – pe aceste noduri de găsesc datele reale. Doar nodul front end se poate conecta la acestea, în cazul unui transfer atât de intrare, cât şi de ieşire.

Arhitectura detaliată se poate vedea în figura 3.4.

Acest model are ca şi caracteristici:

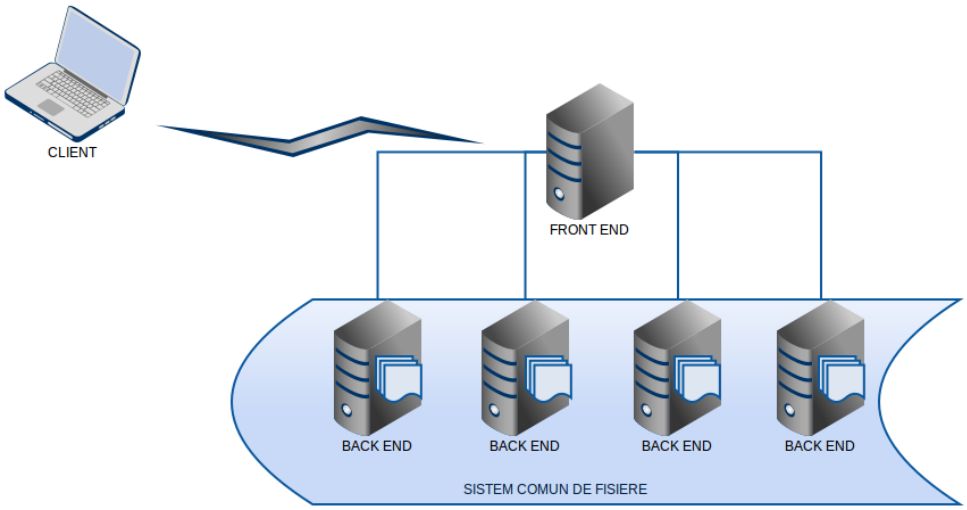



Figura 3.4: Arhitectura GridFTP striped

- modularitate: pentru utilizarea în diferite medii şi cu diverse configuraţii fără nevoia de a modifica implementarea.
- eficienţă: se preferă evitarea păstrării unor copii inutile de date, iar pentru că acestea sunt divizate pe mai multe locaţii fizice, creşte performanţa, cât şi limitările existente.
- siguranţă şi securitate: deoarece doar nodul principal are acces la datele de pe nodurile secundare, sunt limitate accesurile neautorizate la posibile date confidenţiale sau de importanţă ridicată.

## 3.3 Fast Parallel File Replication

Replicarea paralelă a fişierelor mari simultan la mai multe locaţii este o parte integrantă a mediilor de date de tip grid, după cum se precizează şi în [9]. În acest articol este analizat mediul de transfer GridFTP punct-la-punct şi este conceput un mod de transfer punct-la-multipunct paralel pe baza acestuia. Această lucrare prezintă un motor de replicare rapidă paralelă a fişierelor, care creează mai mulţi arbori de distribuţie prin centralizarea sesiunilor GridFTP şi organizarea acestora în structuri logice. În continuare sunt prezentate principalele arhitecturi analizate în [9]. Pentru fiecare dintre aceste arhitecturi se va ilustra arborele obţinut prin aplicarea algoritmului pe figura 3.5.

### 3.3.1 Abordare ierarhică

În această abordare, arborii sunt construiţi prin plasarea nodurilor în manieră ierarhică, iar nodurile de pe nivelele superioare sunt utilizate pentru a transmite sub-fişiere

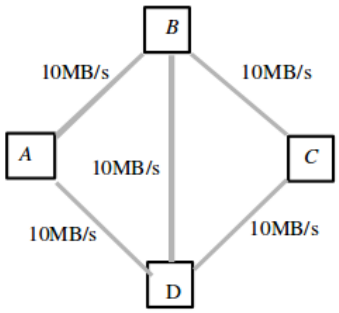



Figura 3.5: Exemplu de noduri şi legături într-o reţea de replicare

nodurilor de pe nivelele inferioare. Această arhitectură se aseamănă foarte mult cu algoritmul *FastReplica*, după cum se specifică şi în lucrare. Construcţia rezultată formează două nivele ale ierarhiei; nu este informată de capacitatea legăturii sau a serverului şi este independentă de topologia reţelei care stă la baza sa. Reţelele obţinute sunt ilustrate în figurile 3.6(a), 3.6(b), 3.6(c). Sunt obţinuţi 3 arbori, dar performanţa nu este cea aşteptată.

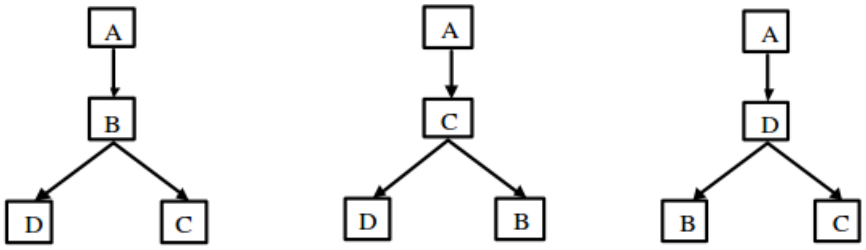



Figura 3.6: Arborii obţinuti cu abordarea ierarhică

### 3.3.2 I-BFS

În această abordare, primul arbore este rezultatul unei căutari în lăţime (*en. Breadth First Search*). Arborele rezultat are nodul sursă ca rădăcină, şi se propagă la toate nodurile destinaţie, după cum se poate vedea în figura 3.3.2. Pentru arborele rezultant $T_i$ se obţine lăţimea minimă de bandă $B_i$. Aceasta reprezintă un blocaj în reţea şi este deci eliminată din setul de lăţimi de bandă ale legăturilor din $T_i$. Algoritmul BFS este apelat din nou pentru a crea al doilea arbore, iar procesul este repetat până când nu se mai pot genera alţi arbori valizi. Avantajul I-BFS, pe lângă timpul mic de convergenţă, este faptul că numărul de conexiuni concurente pentru un singur nod rămâne în limite acceptabile.

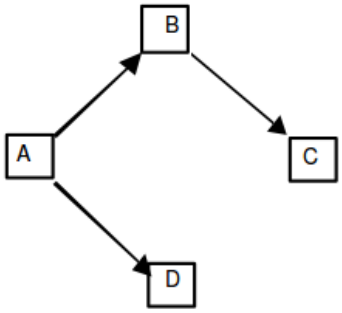



Figura 3.7: Arborele obţinut cu algoritmul I-BFS

### 3.3.3 I-DFS

Această soluţie este la fel ca I-BFS, cu diferenţa că algoritmul de căutare în adâncime (*en. Depth First Search*) este folosit pentru construirea arborilor. Rezultatul poate fi văzut în figurile 3.8(a) şi 3.8(b). Trebuie remarcat faptul că deşi numărul de arbori generaţi este mai mic decât numărul obţinut cu abordarea ierarhică, acest algoritm are cel mai mic timp de convergenţă şi se scalează cel mai bine pe acest exemplu.

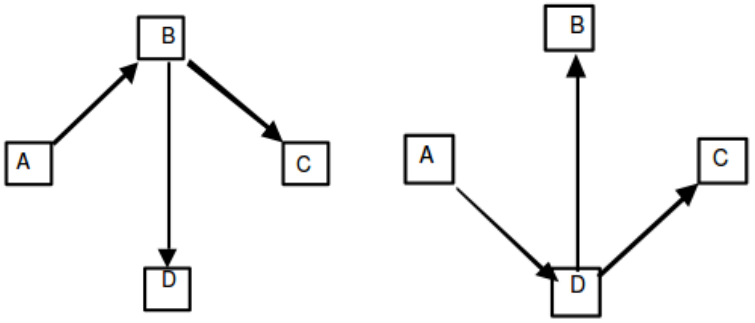



Figura 3.8: Arborii obţinuti cu algoritmul I-DFS

### 3.3.4 I-SW

În această ultimă abordare, primul arbore este creat bazat pe găsirea celei mai scurte şi mai late căi din reţea (*en. Shortest Widest*). Aceasta este obţinută prin utilizarea algoritmului lui Dijkstra, în care metrica folosită pe muchii este lăţimea de bandă. După crearea primului arbore, ceilalţi sunt generaţi folosind aceeaşi manieră iterativă ca şi la I-DFS şi I-BFS. Figura 3.9 ilustrează structura generată cu acest algoritm.

În aceste ultime abordări (cele iterative) descrise în [9], topologia reţelei şi lăţimea de bandă disponibilă a legăturilor trebuie să fie cunoscute înaintea aplicării algoritmilor. Aceste informaţii pot fi achiziţionate folosind un serviciu numit NWS (*en. Network Weather Service*) şi detaliat în [10].

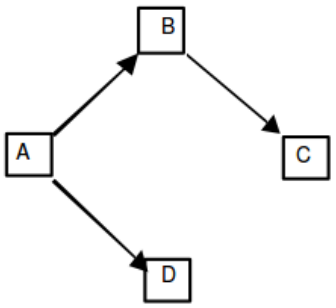


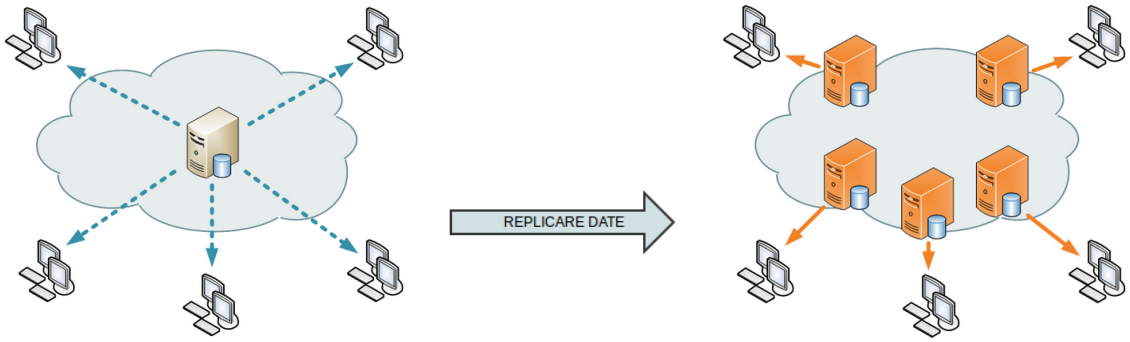

Figura 3.9: Arborele obţinut cu algoritmul I-SW

# Capitolul 4

# Fundamentare teoretică

## 4.1 Reţele cu distribuţie de conţinut

O reţea cu distribuţie de conţinut este un sistem distribuit de mari dimensiuni, în care serverele sunt instalate în diverse centre de date dispersate pe Internet [11]. Scopul unei asemenea reţele este acela de a oferi conţinutul cerut de către clienţi în timp real, adică să aibă o performaţă ridicată. Pentru a atinge această performaţă, reţelele cu distribuţie de conţinut folosesc replicarea pe un grup de servere pentru a putea deservi cu un grad mare de disponibilitate toţi clienţii, indiferent de locaţia fizică a acestora.




(a) Reţea obişnuită

(c) Reţea cu distribuţie de conţinut

Figura 4.1: Crearea unei reţele de distribuţie de conţinut. Sursa [12]

Modalitatea de funcţionare a unui CDN este descrisă în figura 4.1. În locul unui singur nod ce oferă servicii clienţilor, ca în figura 4.1(a), reţelele cu distribuţie de conţinut păstrează copii ale acestor servicii sau fişiere pe mai multe noduri, după cum se poate vedea în 4.1(c). Astfel, atunci când un număr mare de clienţi vor să acceseze aceleaşi date sau servicii, va exista o balansare a cererilor şi ofertelor. Se evită în acest mod supra-încărcarea unui singur server, în timp ce altele nu sunt utilizate. Deşi au fost proiectate

iniţial pentru fluxurile de date continue (*live streaming*), acum acestea sunt folosite pentru orice fel de date: media, obiecte descărcabile, aplicaţii, reţele sociale. Obiectivul este de a oferi un set de servicii stabile şi fiabile, cu performaţă ridicată.

## 4.2 Arhitectura client-server

În domeniul sistemelor distribuite, o arhitectură client-server este o structură formată din două parţi ce funcţionează după următoarele principii:

**Clientul**
Este partea care cere de la server servicii sau date; clientul este cel care iniţiază sesiunea de comunicare. Clientul este cel care are nevoie de resurse, el nu participă cu propriile resurse la această arhitectură.

**Serverul**
Este partea care oferă serviciile şi datele; serverul ascultă cererile clienţilor şi le răspunde acestora. Serverele sunt deseori folosite pentru a da spre folosinţă resurse atât clienţilor, cât şi altor servere. Ele pot restricţiona sau limita accesul la aceste resurse, în funcţie de cereri.

Pentru a putea comunica, clientul şi serverul trebuie să vorbească acelaşi limbaj, deci să urmeze acelasi protocol. După stabilirea unei conexiuni, clientul şi serverul schimbă mesaje folosind acest protocol, după modelul cerere-răspuns. Acest protocol rulează la nivelul de aplicaţie. Comunicarea se realizează de obicei în cadrul unei reţele, dar este posibil ca atât clientul cât şi serverul să se afle pe acelaşi dispozitiv fizic. Un client se poate conecta la mai multe servere, iar un server poate deservi mai mulţi clienţi în acelaşi timp. Clienţii nu pot comunica între ei, dar două servere au posibilitatea de a comunica, dacă este necesar.

## 4.3 Transferuri third-party

După cum am precizat şi în paragraful anterior, două servere pot comunica între ele. Dar chiar şi pentru această comunicaţie este necesar un client care să o iniţieze şi să transmită unuia dintre servere datele celuilalt, pentru ca acesta să se poată conecta la cel ulterior. Primul server devine astfel un client pentru cel de-al doilea, dar clientul iniţial rămâne totuşi informat. Acest tip de comunicaţie poartă numele de *third party*, sau a treia parte, deoarece clientul este aici o a treia parte a comunicaţiei. În cazul unui transfer de acest fel, clientul trebuie să se conecteze la servere pentru a le transmite cererea sa, iar apoi nodul care conţine datele ce trebuie transferate se conectează la nodul care trebuie sa primească datele. Această structură poate fi analizată în figura 4.2.

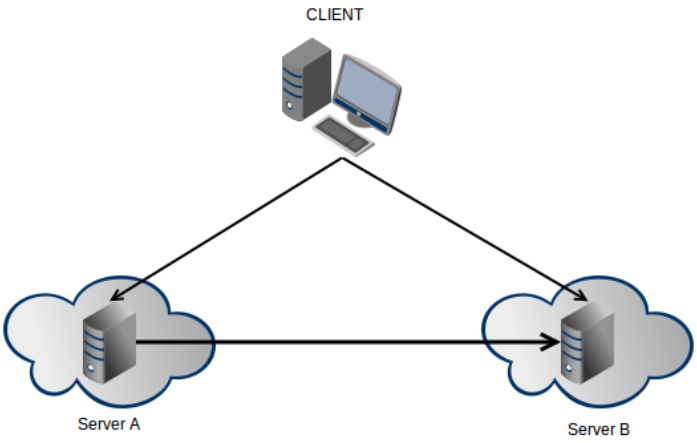



Figura 4.2: Transfer third-party

Canalele pe care clientul se conectează la serverele $A$ şi $B$ se numesc canale de control, deoarece clientul este cel care controlează întreaga operaţiune şi care iniţiază transferul. Între serverul $A$ şi serverul $B$ se află canalul de date, deoarece pe această legătură se transferă datele. Protocolul de comunicaţie dintre client şi server diferă de cel dintre servere, iar serverele trebuie să asculte pe două porturi: unul pentru clienţi şi unul pentru servere. Motivele unei asemenea abordări sunt variate:

- permite mutarea datelor între două locaţii aflate la distanţă de client (prin distanţă a se înţelege că serverele şi clientul nu se află pe aceeaşi maşină fizică, ceea ce înseamnă că clientul nu are acces la datele de pe serverul $A$ decât prin reţea).

- clientul are controlul asupra desfăşurării transferului, el fiind cel care oferă serverelor informaţiile necesare.

- la nivelul transferului de date există transparenţă pentru client, acesta nefiind implicat direct.

- este diminuat traficul pe o singură legătură, deoarece există legături diferite pentru date şi pentru comenzi. Aceasta poate însemna şi o scădere în traficul din reţea.

- serverul sursă preia responsabilitatea pentru efectuarea transferului de la client, devenind chiar el un client în momentul conectării la serverul destinaţie.

- abordarea third-party are aplicabilitate şi în securitatea transferului.

## 4.4 TCP

*Transmission Control Protocol* este unul dintre cele mai importante protocoale de comunicare din suita IP. Acesta este descris în [13]. TCP a fost creat pentru a fi folosit ca protocol de transmisie între două noduri din reţelele bazate pe schimbarea de pachete şi în sisteme interconectate. Ca şi caracteristici, protocolul TCP este un protocol de mare încredere, fiabil şi care oferă funcţia de verificare şi de corectare a erorilor. TCP aparţine nivelul transport, deci este situat deasupra nivelului IP, căruia îi transmite segmente variabile de informaţie sub forma de datagrame. Această ierarhie se vede în figura 4.3, unde este simbolizată o parte din stiva de protocoale TCP-IP.

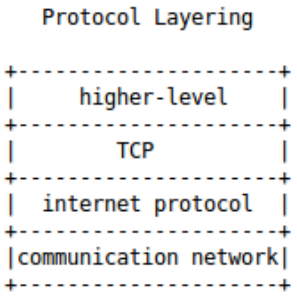



Figura 4.3: O parte din stiva de protocoale TCP-IP

Protocolul TCP este folosit în transferul de fişiere, deoarece oferă încrederea că datele vor ajunge la destinaţie, chiar dacă există erori în reţea. Este un protocol bazat pe flux de date (*en. data streaming*) care garantează că toţi biţii trimişi vor fi la fel cu cei primiţi, în ordinea corectă. O alternativă la protocolul TCP este UDP (*User Datagram Protocol*), care nu oferă retransmisia datelor pentru a asigura corectitudinea, dar care este mult mai rapid. Pentru acesta este necesară o implementare proprie a anumitor caracteristici, în funcţie de necesităţile cerute. Totuşi, pentru transferul de fişiere în reţele cu comutare de pachete, cea mai bună alegere este TCP.

## 4.5 Tehnici de transfer

Pentru realizarea transferului de la un nod sursă la un nod destinaţie, se disting două tehnici de transfer diferite. Acestea sunt descrise în secţiunile următoare. Cele două tehnici sunt identificate din perspectiva nodului care iniţiază distribuirea datelor.

### 4.5.1 Tehnica PUSH

Prima strategie de distribuire este cea de împingere a datelor de la sursă la destinaţie, după cum se poate vedea în figura 4.4. Această strategie este cel mai des folosită

în transferurile de fişiere, deoarece presupune distribuirea datelor cu iniţiativa sursei (*en. sender-initiative*). Avantajul unei asemenea strategii este faptul că sursa poate controla transferul, deoarece ştie ce date posedă şi nu există problema ca datele să nu fie găsite sau să fie corupte.

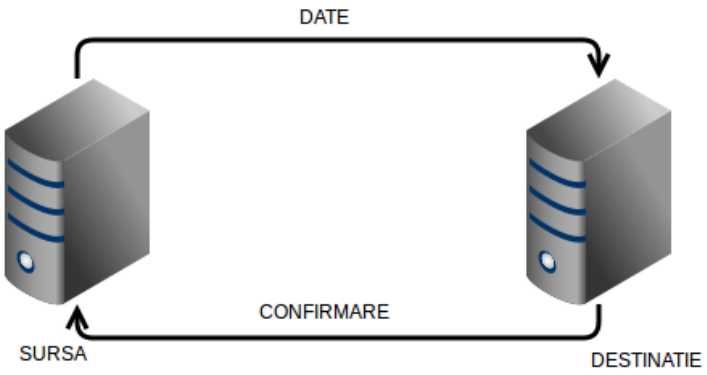



Figura 4.4: Transfer iniţiat de nodul sursă

Comunicarea este realizată în doi paşi:

- în primul pas, sursa se conectează la destinaţie şi începe să îi trimită acestuia datele pe care le deţine. Modalitatea de transmisie poate fi de două feluri: sincronă sau asincronă. În cazul unui transfer sincron, sursa trebuie să aştepte realizarea următorului pas înainte de a începe să trimită din nou date. În cazul unui transfer asincron, sursa poate să ignore confirmarea şi să trimită datele în continuare, până la terminarea acestora. Ambele modalităţi au avantaje şi dezavantaje.

- cel de-al doilea pas constă din răspunsul serverului destinaţie, care trebuie să confirme primirea datelor, în cazul în care transferul este acceptat. În caz contrar, destinaţia poate să refuze conexiunea sau chiar datele, trimiţând înapoi la sursă un mesaj de eroare.

### 4.5.2 Tehnica PULL

Strategia de tragere a datelor de la sursă este opusă celei descrise anterior. În această strategie, cererea de date vine de la destinaţie, după cum este ilustrat şi în figura 4.5. Deşi nu aşa des utilizată pentru transferul de fişiere, această tehnică oferă o mai mare flexibilitate, deoarece destinaţia poate să ceară exact datele de care are nevoie, şi pe care nu le va refuza. Avantajul acestei tehnici este faptul că serverul destinaţie joacă rolul de client, el fiind capabil să ceară date de la sursă.

Şi aici, comunicaţia se desfăşoară în doi paşi:

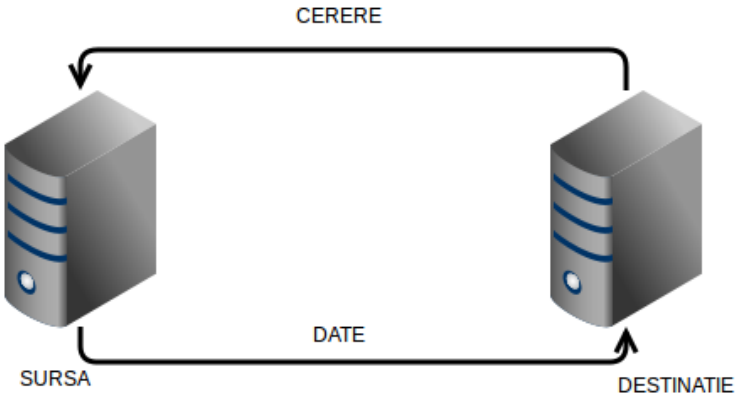



Figura 4.5: Transfer iniţiat de nodul destinaţie

- iniţiatorul transferului este destinaţia (*en. receiver-initiative*), care are nevoie de un anumit set de date. El trimite o cerere către serverul sursă prin care precizează datele de care are nevoie.
- al doilea pas constă din răspunsul serverului sursă, care va începe să trimită datele cerute către serverul destinaţie. În cazul în care sursa nu deţine datele cerute, va răspunde cu un mesaj de eroare.

# Capitolul 5

# Analiză şi design

În acest capitol sunt utilizate informaţiile obţinute în studiul bibliografic pentru a analiza problema ce trebuie rezolvată şi pentru a putea propune diferite soluţii la aceasta.

## 5.1 Analiza problemei

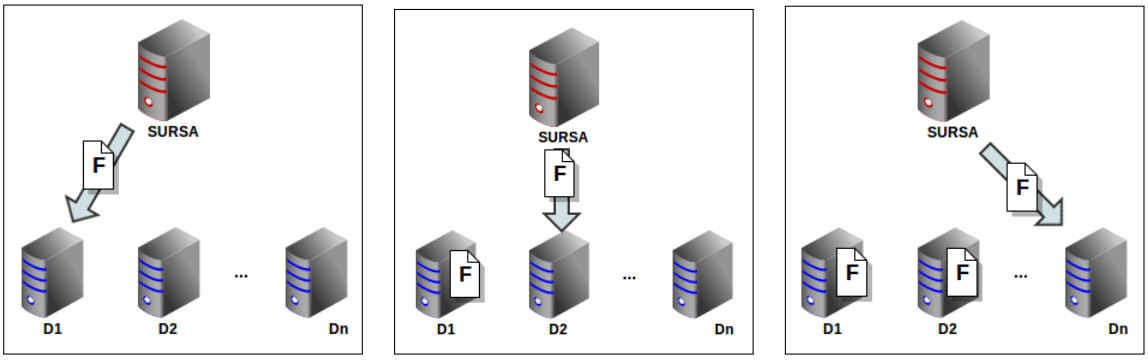
După cum am precizat şi în primul capitol, problema abordată în lucrarea de faţă este a replicării unui set de date sub forma unui fişier pe un număr mare de noduri destinaţie în cadrul unui sistem distribuit. Scopul este acela de a determina numărul maxim de noduri pe care replicarea se poate face, şi, de asemenea, dimensiunea maximă a datelor ce pot fi transferate. Desigur, algoritmul prezentat în această lucrare trebuie să îndeplinească următoarele cerinţe:

- să permită transferul unui volum mare de date.
- să aibă o arhitectură scalabilă (*en. scalability*), ce permite replicarea pe un număr foarte mare de noduri destinaţie.
- să fie adaptiv (*en. adaptability*), deoarece nodurile pe care se face replicarea pot fi dispersate geografic, având lăţimi de bandă diferite, de care trebuie ţinut cont.
- să fie fiabil (*en. reliability*), adică să permită terminarea transferului chiar şi în cazul apariţiei unor erori.
- să aibă un grad mare de disponibilitate (*en. availability*), ceea ce înseamnă că poate deservi un număr mare de clienţi.
- să prezinte rezistenţă la erori şi eşecuri (*en. fault and failure tolerance*); această caracteristică este opusul uneia dintre cele mai mari probleme din domeniul sistemelor distribuite: SPOF (*Single Point Of Failure*).

Până acum am prezentat mai mulţi algoritmi rezultaţi din cercetările efectuate în acest domeniu, alături de avantajele şi dezavantajele acestora. În continuare, vom prezenta trei abordări pentru această problemă: primele două sunt folosite pentru a realiza o comparaţie cu cea de-a treia, care reprezintă aportul personal în acest domeniu.

## 5.2 Distribuţia serială

Prima soluţie propusă pentru comparare este distribuirea serială a datelor. După cum se poate remarca şi din denumire, aceasta presupune trimiterea fişierului la fiecare nod destinaţie de către nodul sursă. Pentru o mai bună înţelegere a acestei abordări, se poate consulta figura 5.1. Aici se poate vedea cum nodul sursă începe prin a se conecta la prima destinaţie, căreia îi trimite fişierul în totalitate (5.1(a)). După acest pas, indiferent dacă a fost realizat cu succes sau nu, nodul sursă începe conexiunea şi respectiv transferul către următorul nod destinaţie (5.1(b)). Acest pas este repetat de $n$ ori, unde $n$ este numărul total de destinaţii. După ce fişierul este trimis şi ultimului nod din setul de destinaţii (5.1(c)), transferul poate fi considerat încheiat. După acest ultim pas, în cazul unei rulări fără erori, fişierul se află pe fiecare dintre destinaţiile vizate.




(a) Primul pas: primul nod destinaţie $D_1$ primeşte fişierul.

(b) Al doilea pas: al doilea nod destinaţie $D_2$ primeşte fişierul.

(c) Ultimul pas: ultimul nod destinaţie $D_n$ primeşte fişierul.

Figura 5.1: Replicarea datelor pe $n$ noduri folosind algoritmul de distribuţie serială.

La o analiză mai atentă a acestei soluţii, se poate vedea că timpul total de transfer este suma timpilor pentru fiecare destinaţie:

$$Timp_{transfer} = T_1(F) + T_1(F) + ... + T_n(F) \tag{5.1}$$

unde $T_i(F)$ reprezintă timpul de transfer al fişierului $F$ pe nodul $D_i$. Dacă notăm cu $BW_i$ lăţimea de bandă a legăturii dintre sursă şi nodul $D_i$ şi cu $Size(F)$ dimensiunea

fişierului $F$, atunci relaţia de mai sus devine:

$$Timp_{transfer} = \frac{Size(F)}{BW_1} + \frac{Size(F)}{BW_2} + ... + \frac{Size(F)}{BW_n} \tag{5.2}$$

Deşi distribuţia serială poate părea şi chiar este o soluţie ineficientă din punct de vedere al timpului, aceasta are totuşi anumite avantaje:

- este scalabilă, deoarece indiferent de numărul de noduri destinaţie, în această arhitectură va exista la orice moment cel mult o conexiune (cea dintre nodul sursă şi cel destinaţie curentă).
- este fiabil, deoarece în cazul în care transferul către unul dintre noduri nu poate fi efectuat, aceasta nu afectează celelalte noduri.
- este de încredere, deoarece sursa nu începe transferul către un alt nod până când nodul curent nu a primit întreg fişierul.

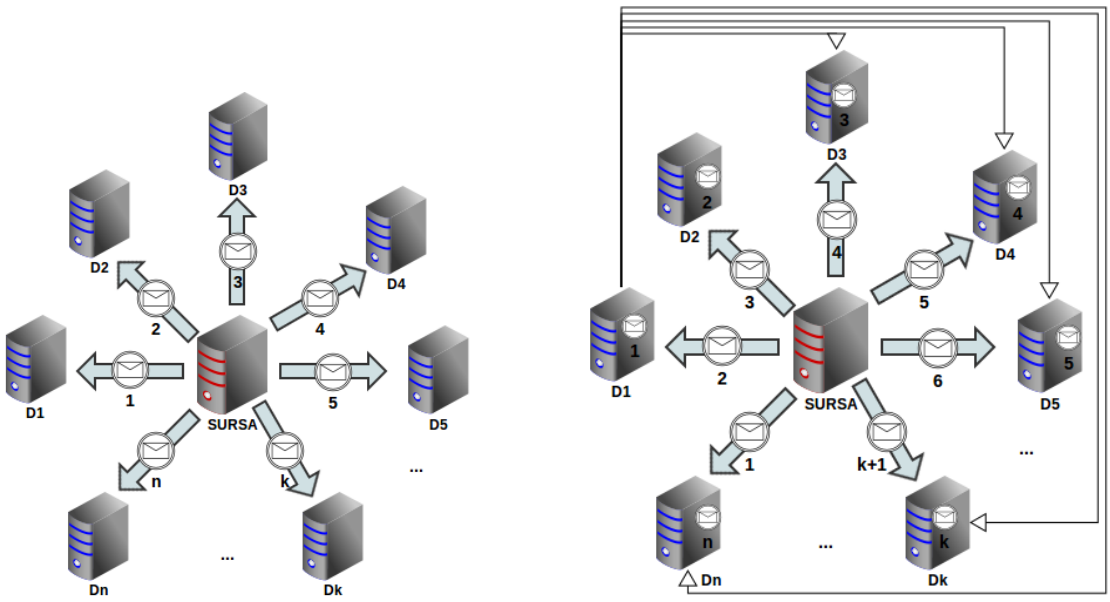
## 5.3 Distribuţia în doi paşi

O a doua abordare folosită pentru testarea algoritmilor ce urmează este o versiune modificată a *FastReplica*, prezentat în capitolul anterior. Aceasta presupune divizarea documentului în $n$ părţi egale ca dimensiune, numite subfişiere, şi executarea a doi paşi:

- în primul pas (numit în *FastReplica* pasul de distribuţie), nodul sursă trimite fiecărei destinaţii $D_k$ subfişierul $k$, după cum se poate observa şi în figura 5.2(a). Acest pas este aici identic cu cel din *FastReplica*.
- în cel de-al doilea pas (numit în *FastReplica* pasul de colecţie), nodul sursă trimite fiecărei destinaţii $D_k$ subfişierul $k+1$ , după cum se poate observa şi în figura 5.2(b). Diferenţa dintre acest algoritm şi *FastReplica* este faptul că în acest pas, fiecare nod destinaţie deschide doar $n-2$ conexiuni concurente la celelalte noduri destinaţie.

Avantajele acestei versiuni faţă de *FastReplica* sunt următoarele:

- deoarece sursa trimite subfişiere şi în al doilea pas, aceste legături sunt folosite la capacitatea lor maximă.
- fiecare nod primeşte două subfişiere de la sursă, ceea ce înseamnă că fiecare nod are $n-2$ legături de intrare în al doilea pas.
- fiecare nod $D_k$ trimite subfişierul primit în primul pas la nodurile $D_{k+2}, D_{k+3}...D_{k-1}$ în cel de-al doilea pas. Deci vor exista $n-2$ legături de ieşire în al doilea pas, totalizând $2n-4$ conexiuni pentru fiecare destinaţie.




(a) Fiecare nod $k$ primeşte pachetul $k$.

(b) Fiecare nod $k$ primeşte pachetul $k+1$.

Figura 5.2: Replicarea datelor pe $n$ noduri folosind algoritmul în doi paşi.

Folosind aceste informaţii se poate calcula timpul de transfer pentru această arhitectură. În ecuaţiile ce urmează, considerăm $n$ numărul de noduri destinaţie, $F$ fişierul ce trebuie replicat, $Size(F)$ dimensiunea acestui fişier şi $BW$ lăţimea de bandă a legăturilor dintre oricare două noduri (se presupune că reţeaua este omogenă). Timpul total de transfer este suma timpilor execuţiei celor doi paşi descrişi mai sus:

$$Timp_{transfer} = Timp_{distributie} + Timp_{colectie} \tag{5.3}$$

Timpul de distribuţie este egal cu timpul de transfer al unui subfişier pe unul dintre nodurile destinaţie:

$$Timp_{distributie} = \frac{Size(F)}{n \times BW} \tag{5.4}$$

Timpul de colecţie este de fapt maximum dintre timpul de distribuţie şi timpul necesar unui nod destinaţie să trimită subfişierul pe care îl posedă pe toate cele $n-2$ conexiuni de ieşire:

$$Timp_{colectie} = max(\frac{Size(F)}{n \times BW}, \frac{(n-2) \times Size(F)}{n \times BW}) = \frac{(n-2) \times Size(F)}{n \times BW} \tag{5.5}$$

Din ecuaţiile 5.3, 5.4 şi 5.5 rezultă că timpul total de replicare a fişierului pe toate destinaţiile este:

$$Timp_{transfer} = \frac{Size(F)}{n \times BW} + \frac{(n-2) \times Size(F)}{n \times BW} = \frac{(n-1) \times Size(F)}{n \times BW} \tag{5.6}$$

## 5.4 MC-5

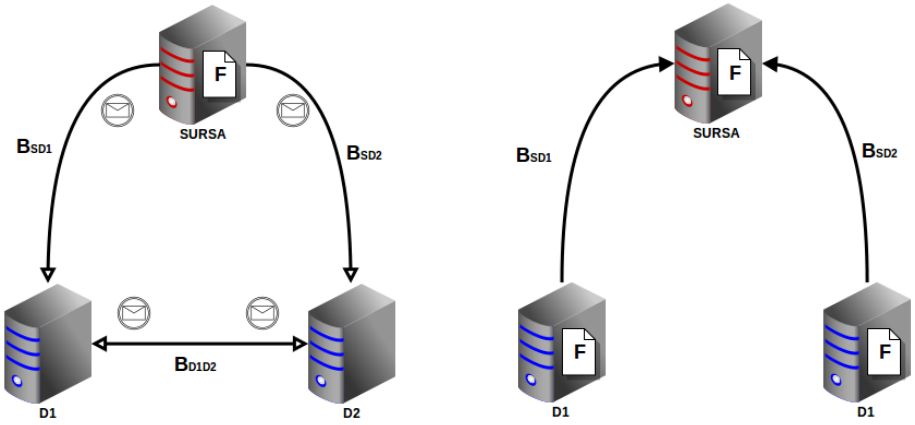
Soluţia propusă pentru problema replicării unui fişier de dimensiuni mari pe multe noduri destinaţie a primit denumirea de **MC-5**. În următoarele paragrafe vom explica de ce a fost ales acest nume şi cum funcţionează această soluţie. Vom vedea de asemenea avantajele şi dezavantajele existente, plus o comparaţie cu ceilalţi algoritmi deja prezentaţi.

### 5.4.1 Replicarea pe două noduri

Pentru a putea genera o arhitectură scalabilă pentru un număr mare de destinaţii, uneori este necesar să se pornească de la un număr mic de destinaţii. În această idee, prima parte a soluţiei prezentate aici este replicarea fişierului pe două noduri. Situaţia poate fi analizată în figura 5.3. Topologia este formată dintr-un nod sursă $S$ şi două noduri destinaţie $D_1, D_2$. Iniţial, fişierul ce va fi replicat $F$ se găseşte doar pe nodul sursă. Acesta se conectează la ambele destinaţii în acelaşi timp şi le transmite acestora informaţiile necesare pentru a se putea realiza transferul:

- numele şi calea fişierului $F$.
- dimensiunea acestuia, care în calculele ce urmează va fi notată cu $Size(F)$.
- dimensiunea unui pachet, deoarece pentru transfer, fişierul va fi trimis în pachete.
- numărul total de pachete.
- adresa IP a celuilalt nod destinaţie (care de aici încolo va fi referit ca fratele nodului curent).
- portul pe care se poate face conectarea la nodul frate (acesta este necesar deoarece se doreşte realizarea unei conexiuni bidirecţionale între cele două destinaţii).

În primă fază, nodul sursă are datoria de a căuta informaţiile corecte despre fişier: folosind calea fişierului, nodul sursă trebuie să verifice existenţa fişierului, drepturile de editare/copiere/modificare ale acestuia, cât şi dimensiunea totală. Apoi, în funcţie de dimensiune şi de alţi parametrii, nodul sursă decide mărimea pachetelor de date şi numărul acestora. Alegerea mărimii pachetelor va fi explicată mai detaliat în secţiunea de implementare. După trimiterea pachetelor de configuraţii către nodurile destinaţie, acestea vor răspunde nodului sursă cu o confirmare ce arată că au primit configuraţiile şi că au relizat conexiunea între ele. Acest pas înseamnă declanşarea transferului propriu-zis, deoarece




(a) Situaţia iniţială (b) Situaţia după replicare

Figura 5.3: Replicarea datelor pe două noduri folosind toate legăturile disponibile.

începând cu acest moment, nodul sursă va trimite fiecărei destinaţii câte un pachet, alternativ, începând din ambele capete ale acestuia, până ce se ajunge la un punct comun. Acest pas poate fi văzut în figura 5.3. La primirea unui pachet, fiecare nod destinaţie aplică o strategie de stocare şi înaintare (*en. store-and-forward*) adică memorează pachetul primit şi îl trimit mai departe nodului frate.

Folosind această strategie, prezentată în algoritmul 1, se observă că nodurile destinaţie vor primi date pe toate legăturile existente. Pentru o înţelegere mai bună a acestei strategii, poate fi consultată figura 5.4. Aici se pot vedea pachetele trimise de nodul sursă, cât şi cele trimise de nodurile destinaţie între ele. Folosind acest algoritm, nodul sursă nu are nevoie de confirmarea primirii fiecărui pachet în parte de către destinaţii. După cum se poate vedea şi în algoritmul 1, el va trimite pachetele de date concomitent către destinaţii începând cu ambele capete ale fişierului. Până nu se ajunge la un punct comun, nodul sursă va tot trimite pachetele, folosind tehnica de împingere a datelor (*en. push*) , prezentată anterior. După ce se ajunge la acest punct comun, nodul sursă a trimis fiecare pachet disponibil o singură dată pe oricare dintre legături. Acest aspect este important deoarece se doreşte evitarea trimiterii mai multor pachete pe aceeaşi legătură, la fel ca şi primirea aceluiaşi pachet pe mai multe legături.

Evitarea cererii unor confirmări de primire pentru pachete este avantajoasă din punct de vedere al timpului, aceasta fiind o caracteristică importantă a tehnicii *push*. Trebuie precizat faptul că trimiterea unui pachet de către nodul sursă se referă la acţiunea de iniţiere a transferului propriu-zis, şi nu la momentul de primire a pachetului. Deci, după ce nodul sursă va termina de trimis primul set de pachete pentru fiecare nod, va exista o diferenţă de timp până când acestea se vor afla pe serverele destinaţie (în acest timp, zicem că pachetele se află în tranzit, adică în coada de mesaje TCP). În timp ce nodul sursă trimite aceste pachete, nodurile destinaţie comunică şi ele folosind o legătură

bidirecţională, prin care trimit pachetele primite de la sursă. Folosind această tehnică, fiecare nod va primi într-un final toate pachetele (jumătate de la sursă, jumătate prin intermediul nodului frate). Dar această abordare ar însemna să nu fie utilizate toate legăturile existente la capacitatea lor maximă. De aceea, după ce sursa a trimis prima serie de pachete fiecărui nod, ea cere informaţii de la destinaţii ce o ajută să determine ce pachete mai sunt necesare pentru fiecare. Aceasta este tehnica de aducere a datelor (*en. pull*) şi este folosită pentru a determina exact ce pachete lipsesc fiecărei destinaţii.

**Algorithm 1** Transferarea unui fişier pe două destinaţii

**Date intrare:**

- $D_1, D_2$ – nodurile destinaţie.
- $F$ – fişierul ce va fi transferat.
- $packsz$ – dimensiunea unui pachet.

**TRIMITE_FISIER**($D_1, D_2$, $F$, $packsz$)

```
deschide_conexiune(D1)
deschide_conexiune(D2)
n ← ⌈size(F)/packsz⌉
for i = 1 → j, j = n → i do
   trimite(D1, F[i])
   trimite(D2, F[j])
   i ← i + 1
   j ← j − 1

end for
while True do
   ind1 ← preia_ultimul_index(D1)
   ind2 ← preia_ultimul_index(D2)
   if ind1 ≠ ind2 then
      trimite(D1, F[i])
      i ← i + 1
      trimite(D2, F[j])
      j ← j − 1
   else
      return
   end if
end while
```

Aceşti paşi sunt repetaţi până când fiecare nod destinaţie trimite nodului sursă confirmarea că a primit toate pachetele şi că fişierul este acum complet ( figura 5.3(b)). Conexiunile sunt oprite, întâi cele dintre destinaţii, iar după trimiterea confirmării, şi cele

dintre sursă şi destinaţii. Fişierul $F$ se află acum pe toate nodurile implicate în transfer.

Algoritmul descris mai sus este simplu, dar puternic. Să analizăm câteva caracteristici ale acestuia, comparativ cu abordarea clasică de transfer serial:

- are un număr mai mare de conexiuni, deoarece există conexiuni şi între destinaţii, dar acestea sunt utilizate la capacitatea lor maximă – sunt folosite mereu pentru a transfera informaţii.
- la fel ca şi în abordarea serială, replicarea datelor pe un nod nu depinde de eşecul sau succesul fratelui acestuia – datorită modului în care se transferă fişierul, există certitudinea că toate pachetele de date vor ajunge la destinaţii, pe una dintre legăturile existente.
- spre deosebire de abordarea serială, aici sunt utilizate mai multe conexiuni, ceea ce se înseamnă fiabilitate ridicată, dar şi o cantitate crescută de informaţii de configurare ce trebuie trimise de la sursă.
- timpul de transfer este evident mai mic decât în cazul unei soluţii în care fişierul este transmis pe rând fiecărei destinaţii.
- comunicarea minimizată dintre noduri este un avantaj pentru minimizarea timpului de replicare.
- în cazul în care nu se poate realiza conexiunea între destinaţii, acest algoritm se va reduce la o schemă de distribuţie serială.

Dacă realizăm o analiză a acestui algoritm, se observă că acesta seamănă cu o versiune cu două noduri a algoritmului în doi paşi descris mai sus. Există totuşi câteva diferenţe importante:

- în această soluţie fişierul nu este divizat în două subfişiere, ci în multiple pachete de dimensiuni considerabil mai mici decât ale subfişierelor. Aceasta permite realizarea transferului în mai mult de doi paşi.
- deşi numărul şi tipul conexiunilor este acelaşi ca şi la algoritmul în doi paşi, aici felul în care sunt trimise datele diferă prin faptul că nodurile destinaţie au rolul de a retrimite orice pachete primesc de la sursă. Astfel, chiar dacă legătura dintre sursă şi una dintre destinaţii nu este funcţională, fişierul poate fi trimis folosindu-se celălalt nod destinaţie ca intermediar.
- numărul relativ mare de pachete trimise în această abordare permite nodurilor să se adapteze la viteza legăturilor din reţea: pe legăturile mai bune vor fi transferate mai multe pachete.
- în situaţia în care nodul sursă decide divizarea fişierului în două pachete, cei doi algoritmi devin identici.

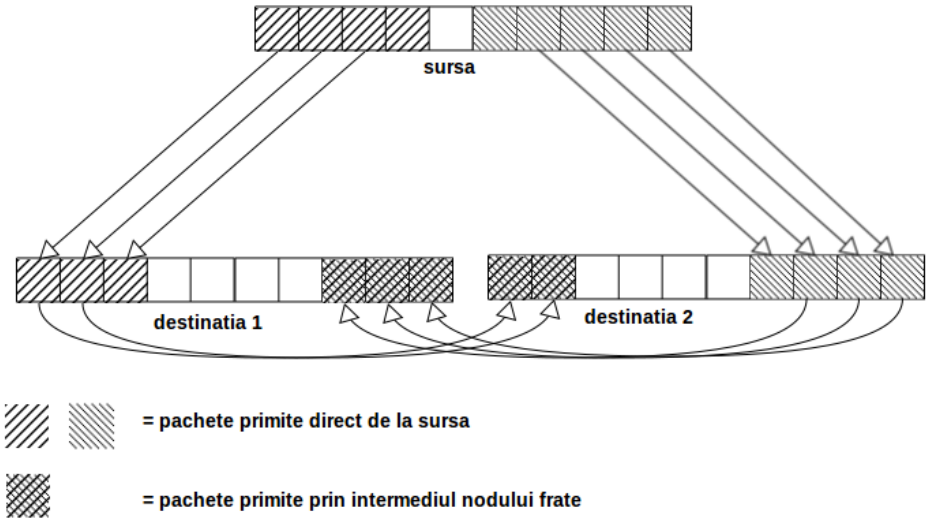



Figura 5.4: Divizarea documentului şi trimiterea datelor pe toate căile posibile în arhitectura cu două noduri destinaţie.

Pentru calcularea timpului total de replicare în această situaţie, vom considera o reţea omogenă, cu lăţimea de bandă $BW$. $T_{D1}$ şi $T_{D2}$ sunt timpii de transfer pentru nodurile $D_1$ respectiv $D_2$. Aceştia pot fi exprimaţi ca produsul dintre timpul de transfer pentru un pachet de date şi numărul total de pachete trimise pe acea legătură. Dimensiunea unui pachet este raportul între dimensiunea totală a fişierului şi numărul de pachete $k$.

$$Timp_{replicare} = max(T_{D1}, T_{D2}) = \frac{k}{2} \times \frac{\frac{Size(F)}{k}}{BW} = \frac{Size(F)}{2 \times BW} \tag{5.7}$$

### 5.4.2 Replicarea pe seturi reduse de noduri

Pornind de la algoritmul descris anterior şi de la o arhitectură minimală, cu doar două noduri, se poate generaliza şi se poate găsi o arhitectură ce reuşeşte să se scaleze pe un număr mare de noduri. Abordarea propusă în această lucrare porneşte de la această idee, iar modalitatea de generare a acesteia este ilustrată în figura 5.5.

Ideea de bază a acestei abordări de genul împarte şi cucereşte (*lat. divide et impera*) este de a nu privi toate nodurile ca un mare set, ci de a le restructura în seturi mai mici, cu minim două şi maxim cinci noduri fiecare. Aceste seturi se numesc grupuri de replicare şi pot fi de patru feluri ($G_1$, $G_2$, $G_3$, $G_4$), după cum se vede şi în figura 5.5. Un grup este format din trei tipuri de noduri:

- nod sursă ($S$) – acesta este unic şi comun tuturor grupurilor de replicare.

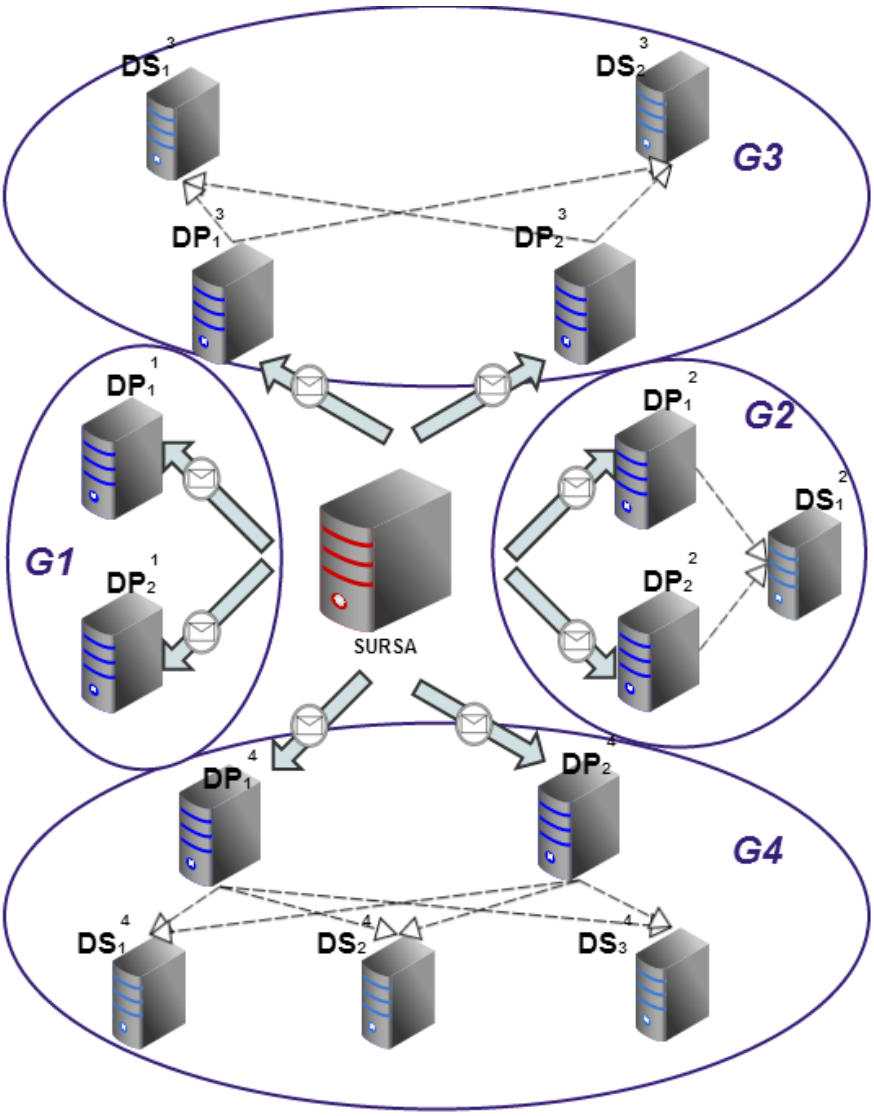



Figura 5.5: Arhitectura pentru replicarea datelor pe grupuri formate din maxim 5 noduri.

- noduri destinaţie principală ($DP_i$) – două la număr, aceste noduri formează, împreună cu sursa, o structură de genul celei prezentate în subcapitolul anterior. Acestea se conectează direct la sursă, cât şi între ele, şi au rolul de a transmite mai departe pachetele primite.
- noduri destinaţie secundară ($DS_i$) – aceste noduri pot fi în număr de zero, unu, doi sau trei şi se conectează la destinaţiile principale pentru a primi de la acestea fişierul. Ele nu comunică direct cu sursa şi nu trimit pachetele primite mai departe.

În funcţie de numărul total de destinaţii, nodul sursă decide schema de distribuţie,

| $n \pmod 5$ | Schema de distribuţie | | Conexiuni sursă |
|---|---|---|---|
| 0 | | $k \times G_4$ | $2 \times k$ |
| 1 | $k = 0$ | unicast | 1 |
| | $k > 0$ | $2 \times G_2 + (k-1) \times G_4$ | $2 \times (k+1)$ |
| 2 | $k = 0$ | $G_1$ | 2 |
| | $k > 0$ | $G_2 + G_3 + (k-1) \times G_4$ | $2 \times (k+1)$ |
| 3 | $k = 0$ | $G_2$ | 2 |
| | $k > 0$ | $G_2 + (k-1) \times G_4$ | $2 \times (k+1)$ |
| 4 | $k = 0$ | $G_3$ | 2 |
| | $k > 0$ | $G_3 + (k-1) \times G_4$ | $2 \times (k+1)$ |

Tabela 5.1: Generarea schemei de distribuţie din numărul total de destinaţii

adică ce fel de grupuri să genereze. În tabelul 5.1 se pot vedea schemele generate şi numărul de conexiuni totale pentru nodul sursă pe baza acestor scheme. Aici, $n$ reprezintă numărul de destinaţii, iar $G_1$, $G_2$, $G_3$, $G_4$ tipurile de grupuri de replicare.

Din acest tabel se poate observa că numărul maxim de conexiuni pentru nodul sursă este $2 \times (k+1)$, unde $k = n \div 5$. De aici rezultă maxim $\frac{2 \times n}{5} + 2$ conexiuni folosind acest sistem. Acest rezultat face din topologia propusă una scalabilă pentru un număr mare de destinaţii. În algoritmul 2 sunt descrise acţiunile luate de un nod destinaţie la primirea unui pachet. Deoarece este posibil şi chiar foarte probabil ca nodurile destinaţie să nu primească pachetele de date în ordinea naturală $(1, 2, ...n)$ acestea nu le pot scrie direct în fişier. Rezultă ca va fi necesară o zonă de memorie pentru pachetele care nu sunt încă în ordine. Cea mai uşoară abordare, în acest caz, este de a memora toate pachetele primite într-o zonă tampon (*en. buffer*) şi de a le scrie în fişier după ce sunt primite toate. Totuşi, din cauza limitărilor de memorie, această abordare nu este mereu posibilă. Din această cauză, o soluţie alternativă trebuie găsită. Pentru aceasta, algoritmul 2 a fost creat, pornind de la ideea de unei ferestre glisante (*en. sliding window*) de memorie. Fiecare nod sursă are un buffer de memorie de dimensiune $r = Size(W)$ – aceasta este denumită dimensiunea ferestrei şi este comunicată de către nodul sursă la începutul fiecărui transfer, deoarece depinde de mărimea fişierului. Deoarece nu se doreşte memorarea tuturor pachetelor primite, ci doar a celor care nu pot fi scrise în fişier chiar la primire, $r$ va fi mai mic decât numărul total de pachete. Să analizăm acum algoritmul pe paşi:

- atunci când un pachet nou este primit, prima sarcină a nodului sursă este să verifice dacă aceasta poate fi scris direct în fişier.

- dacă da, nodul sursă trebuie să parcurgă fereastra şi să găsească şi pachetele următoare ce pot fi scrise.

- dacă nu, se caută un loc liber în fereastră, denumit slot, iar pachetul este memorat acolo până când va putea fi scris în fişier.

- în cazul în care nu mai există locuri libere în fereastră, pachetul cu cel mai mare index (deci cel care va fi scris cel mai târziu în fişier) va fi aruncat, iar noul pachet îi va lua locul.

**Algorithm 2** Procesarea unui pachet pe nodurile destinaţie.

**Date intrare:**

- $P$ – pachetul primit.
- $W$ – fereastra de memorie.
- $F$ – fişierul în care se vor scrie datele.

**PROCESARE_PACHET**($P, W, F$)

```
urm ← INDICE_URM(F)
curent ← INDICE_PACHET(P)
if urm = curent then
  SCRIE(F, P)
  while True do
    urm ← INDICE_URM(F)
    curent ← INDICE_MINIM(W)
    if urm = curent then
      SCRIE(F, W[curent])
    else
      break
    end if
  end while
else
  if ESTE_PLIN(W) then
    slot ← INDICE_MAXIM(W)
  else
    slot ← GASESTE_LIBER(W)
  end if
  W[slot] ← P
end if
```

### 5.4.3 Analiza soluţiei propuse

După detalierea arhitecturii propuse şi a algoritmilor de transfer, vom vedea în continuare o analiză a acestei soluţii din mai multe puncte de vedere:

**Scalabilitate**

Această caracteristică se referă la posibilitatea de a realiza topologii cu număr foarte mare de noduri, fără a cauza probleme de performanţă. Deoarece nodurile destinaţie

sunt divizate în grupuri de replicare după cum a fost descris, numărul total maxim de conexiuni pentru fiecare dintre acestea este şase (pentru destinaţiile principale, două conexiuni de intrare şi patru de ieşire).

Problema se pune la nodul sursă care are două legături cu fiecare grup de replicare. Deci, pentru un număr de $A$ grupuri, sursa va avea $2 \times A$ conexiuni. Numărul de grupuri este dat de numărul de destinaţii împărţit la 5, deci pentru 100 destinaţii sursa va avea 40 conexiuni. Putem zice că soluţia propusă este scalabilă, dar cu o anumită limită.

**Adaptabilitate**

Acest punct se referă la modificarea anumitor parametrii de transfer sau adaptarea anumitor caracteristici pentru a obţine o performanţă ridicată. În cazul de faţă, trebuie remarcat că adaptabilitatea este realizată prin utilizarea tutoror legăturilor disponibile între noduri. Acestea sunt în permanenţă ocupate cu pachete de configurare sau de date, optimizând foarte mult transferul. De asemenea, se observă că pe legăturile mai bune, adică mai rapide, se vor trimite mai multe pachete, deoarece pe acestea vor veni mai multe cereri.

**Fiabilitate**

Spunem despre un sistem că este fiabil sau de încredere dacă acesta reuşeşte să îndeplinească scopul pentru care a fost proiectat de fiecare dată, indiferent de diversele medii în care e executat. Aici, fiabilitatea reprezintă garanţia că întreg fişierul va ajunge pe nodurile destinaţie, chiar dacă anumite pachete sunt pierdute sau deteriorate.

Acest fapt poate fi demonstrat prin algoritmii prezentaţi anterior. Aceştia asigură trimiterea pachetelor de date prin faptul că sursa cere mereu confirmări de la destinaţii pentru primirea tuturor pachetelor. În cazul în care primirea nu e confirmată, se execută o retrimitere a pachetelor pierdute sau deteriorate.

**Toleranţă la erori**

Uneori, înainte sau chiar în timpul execuţiei, pot apărea erori pe serverele destinaţie. Toleranţa la erori înseamnă că defectarea unui server nu ar trebui să afecteze în nici un fel replicarea fişierului pe celelalte servere. Să analizăm această problemă în contextul sistemului prezentat aici:

- în cazul în care serverul pe care apare eroarea este o destinaţie principală, destinaţiile secundare care ar trebui să primească date de la acesta vor primi totuşi pachetele prin intermediul fratelui acestuia, deci problema este rezolvată.
- în cazul în care serverul pe care apare eroarea este o destinaţie secundară, acesta nu este responsabil cu retrimiterea pachetelor către alte servere, deci nici aici nu se poate vorbi de propagarea erorilor.

În concluzie, folosind abordarea propusă, cu grupuri mici de replicare, putem fi siguri că fişierul va fi replicat pe toate serverele funcţionale.

**Eficienţă memorie**

Deoarece dorim să transferăm volume mari de date, managementul memoriei este o caracteristică necesară pentru acest sistem. Pentru nodul sursă, ideal este să nu se folosească zone inutile de memorie pentru a ţine pachetele de date: aceasta se realizează prin citirea directă a datelor din fişier. Pentru nodurile destinaţie, este important să nu se stocheze pachetele primite în memoria internă a proceselor, deoarece aceasta este limitată şi poate ridica excepţii ce duc la terminarea prematură a transferului. De aceea, algoritmul 1 a fost creat, pentru a ţine sub controlul nivelul memoriei folosite la un moment dat.

**Eficienţă timp**

Scopul general al acestei lucrări este de a avea un sistem performant din punct de vedere al timpului de transfer, adică de a încerca minimizarea acestuia. După cum am văzut, pentru fiecare grup de replicare, sursa este considerată ca parte integrantă a acestuia, deoarece ea trimite pachetele concurent tuturor grupurilor. Această caracteristică este o importantă creştere a eficienţei, la fel ca şi folosirea tuturor legăturilor disponibile dintre noduri.

# Capitolul 6

# Proiectare de detaliu şi implementare

## 6.1 Mediul de dezvoltare

Pentru implementarea aplicaţiei din lucrarea de faţă a fost ales ca limbaj de programare Python, iar ca platformă de dezvoltare Twisted. Acestea sunt prezentate în secţiunile ce urmează.

### 6.1.1 Python

**Python** este un limbaj de programare de nivel înalt apărut în anul 1991. Python oferă multiple paradigme de programare, precum programare orientată obiect, programare imperativă şi funcţională. Acesta dispune de un sistem dinamic pentru tipurile de date şi de management automat al memoriei, două puncte forte pentru aplicaţia prezentă. Este disponibil pentru toate marile tipuri de sisteme de operare şi are licenţă *open source*. Permite atât interpretarea codului (rulare direct folosind fişierele sursă) cât şi compilarea acestuia pentru a creşte viteza de execuţie. De asemenea, oferă posibilitatea de a structura codul şi de a-l arhiva în module şi librării simplu de utilizat.

Python a fost conceput cu o sintaxă ce permite exprimarea conceptelor în mai puţine linii de cod decât în alte limbaje şi cu un design ce conferă lizibilitate codului. Accentul este pus pe construirea programelor din punct de vedere funcţional prin evitarea scrierii de cod inutil. Acestea sunt câteva dintre motivele pentru care Python a fost ales pentru implementarea sistemului prezentat în lucrarea de faţă.

### 6.1.2 Twisted

Twisted este o platformă pentru crearea de aplicaţii de reţea scrisă în Python şi licenţiată *open-source*. Aceasta este disponibilă la adresa http://twistedmatrix.com. Twisted este bazat pe paradigma de programare acţionată de evenimente (*en. event-driven*). În figura 6.1 este prezentată structura unei aplicaţii realizate cu Twisted: este

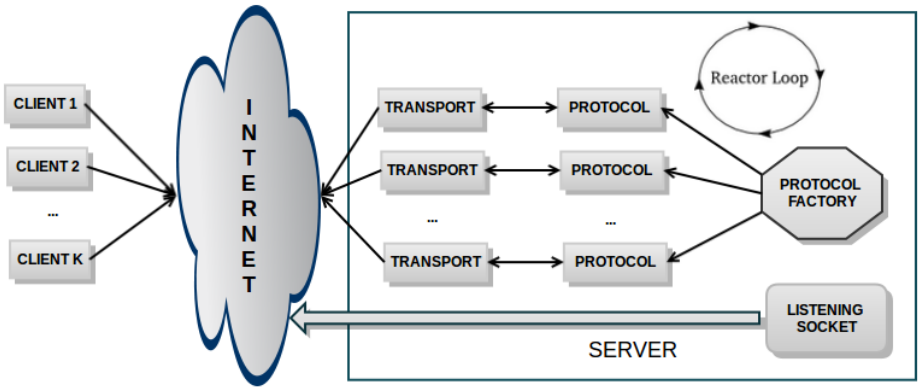



Figura 6.1: Funcţionarea unui server implementat cu Twisted.

vorba despre un server ce aşteaptă conexiuni de la clienţi prin intermediul Internetului. În această imagine sunt câteva dintre cele mai importante elemente Twisted:

**Reactor**
: Această componentă este implementată conform modelului de proiectare reactor (*en. reactor design pattern*): o buclă care aşteaptă apariţia unor evenimente, pe care apoi le gestionează. Acest model este util pentru manevrarea cererilor concurente de la mai mulţi clienţi.

**Transport**
: Această componentă reprezintă o singură conexiune care poate trimite sau primi octeţi. După cum sugerează şi numele, aceasta reprezintă o abstractizare a transportului de date. Implementarea oferită de Twisted este non-blocantă şi este recomandat să se folosească în locul unei implementări proprii.

**Protocol**
: Această componentă poate fi suprascrisă pentru a implementa un protocol specific de reţea. Fiecare instanţă Protocol este utilizată pentru gestionarea unei singure conexiuni, deci aici vor fi memorate starea protocolului cât şi datele parţial primite. Fiecărei instanţe Protocol îi corespunde o instanţă Transport, care reprezintă conexiunea pe care protocolul acţionează. Twisted oferă implementări pentru cele mai comune şi mai des utilizate protocoale.

**Factory**
: Crearea instanţelor unui Protocol este datoria componentei Factory. Aceasta este implementată folosind modelul de proiectare fabrică (*en. factory design pattern*). De fiecare dată când o nouă cerere de conexiune apare, este de datoria acestei componente să instanţieze un nou Protocol.

## 6.2 Implementare

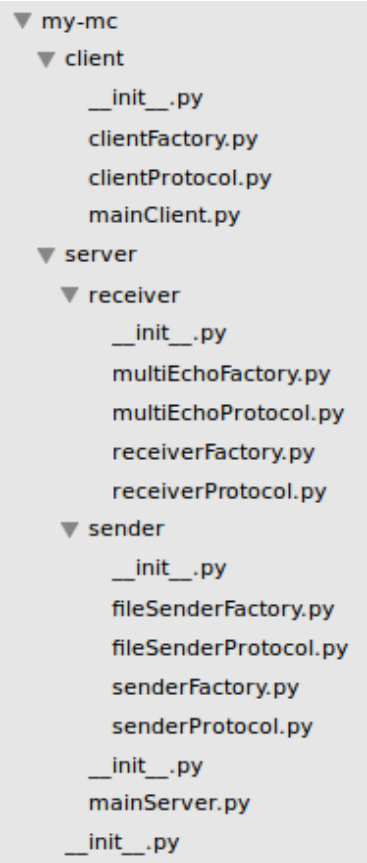



Figura 6.2: Structura aplicaţiei pe module.

După cum a fost precizat şi anterior, implementarea a fost făcută în Python, folosind platforma Twisted. Structura aplicaţiei poate fi analizată în imaginea 6.2, din stânga. Aceasta este modularizată şi structurată în două module ce sunt detaliate mai jos.

Când un modul este importat, interpretorul Python va îl va căuta în căile predefinite, şi când îl va găsi, îl va adăuga în tabela locală de simboluri. Din motive de eficienţă, fiecare module este importat o singură dată per sesiune. Ca o importantă accelerarea a timpului de pornire pentru programele scrise în Python care utilizează multe module, codul pre-compilat este utilizat (*en. byte-compiled code*). Modalităţile de lansare în execuţie a clientului şi a serverului vor fi descrise mai detaliat în manualul de instalare şi utilizare, alături de resursele necesare.

Fiecare modul conţine fişierul __init__.py, care este necesar pentru ca Python să trateze folderele în care se află ca şi module; această abordare a fost aleasă pentru a preveni folderele cu nume comune de la a fi confundate cu adevăratele module Python (*e.g. string*). În cel mai simplu caz, __init__.py poate fi un doar un fişier gol, dar poate fi şi folosit pentru a executa cod de iniţializare pentru pachetul respectiv. În structura aceasta, fişierele __init__.py sunt goale, deci nu vor mai fi prezentate în paragrafele următoare.

### 6.2.1 Implementare client

Primul modul din imaginea 6.2 este denumit **client** şi este format din următoarele fişiere sursă:

**mainClient.py**

Acesta este punctul de începere a execuţiei unui client, fişierul ce conţine metoda principală de lansare (*en. main*). Deoarece lansarea se face din terminal primul pas este de a analiza opţiunile date ca argumente în linia de comandă:

**'-sa'/'–source-address'** Adresa IP a serverului sursă. Dacă nu este precizată, se va folosi adresa de buclă locală 127.0.0.1.

**'-sf'/'–source-filepath'*** Calea fişierului pentru serverul sursă. Acest argument este necesar.

**'-mc'/'–multicast'*** Calea fişierului de multicast în care sunt scrise, câte una pe linie, adresele IP ale serverlor destinaţie. Acestea trebuie să fie de forma $X.X.X.X : [P]$, unde $X \in [0, 255]$ şi $P \in [20000, 25000]$. Acest argument este necesar.

**'-sp'/'–source-port'** Portul pe care se realizează conectarea la serverul sursă. Valoarea implicită este 22020.

**'-df'/'–dest-filepath'*** Calea fişierului pentru serverul sursă. Acest argument este necesar.

**'-dp'/'–dest-port'** Portul pe care se va realiza conectarea la serverele destinaţie în cazul în care acesta nu este precizat în fişierul de multicast. Valoarea implicită este 22000.

**'-ps'/'–packet-size'** Dimensiunea unui pachet exprimată în octeţi. Valoarea implicită este de 67108864 octeţi(64MB).

**'-ws'/'–window-size'** Dimensiunea ferestrei glisante exprimată în număr de sloturi. Valoarea implicită este de 10 sloturi.

**'-lf'/'–logfile'** Calea fişierului jurnal (*en. logfile*) în care se scriu diverse informaţii din timpul transferului şi eventuale erori. Valoarea implicită este /tmp/log/-client.log

După parsarea argumentelor din linia de comandă, se deschide fişierul de multicast pentru a se citi adresele serverlor destinaţie. Următorul pas este iniţializarea protocolului de comunicare între client şi sursă, prin pornirea reactorului şi deschiderea canalelor de comunicaţie.

**clientFactory.py**

MyClientFactory este o clasă ce extinde ClientFactory, implementat în Twisted, şi care are rolul de a instanţia protocolul de comunicare.

**clientProtocol.py**

MyClientProtocol este clasa ce extinde LineReceiver şi care conţine protocolul de comunicare între client şi serverul sursă. Metodele principale din această clasă sunt:

**connectionMade** apelată atunci când conexiunea s-a realizat.

**lineReceived** apelată la primirea unei linii de la celălalt capăt al comunicaţiei.

### 6.2.2 Implementare server

Pentru servere am ales o implementare separată pentru sursă, respectiv destinaţie. Fiecare din aceste instanţe ascultă pe un alt port, deci în cazul unui transfer nu mai este

necesară pornirea unei noi instanţe, făcând aplicaţia mai eficientă din punct de vedere al timpului. Modulul **server** este format din două submodule şi fişierul de lansare în execuţie a unui server – mainServer.py. Şi aici, primul pas este parsarea argumentelor din linia de comandă:

**'-sp'/'–source-port'** Portul pe care va asculta serverul sursă. Valoarea implicită este 22020.

**'-dp'/'–dest-port'** Portul pe care va asculta serverul destinaţie. Valoarea implicită este 22000.

**'-lf'/'–logfile'** Calea fişierului jurnal (*en. logfile*) în care se scriu diverse informaţii din timpul transferului şi eventuale erori. Valoarea implicită este /tmp/log/server.log

După parsarea argumentelor, sunt pornite cele două instanţe de protocol şi este pornit reactorul. Din acest moment, serverul aşteaptă conexiuni de la clienţi.

### Sursă

Componentele pentru implementarea serverului sursă pot fi văzute în modulul **server.sender**. Acestea sunt:

**senderFactory.py**
MySenderFactory extinde ServerFactory, şi are rolul de a crea instanţe MySenderProtocol.

**senderProtocol.py**
MySenderProtocol extinde LineReceiver şi conţine protocolul de comunicare între client şi serverul sursă. Metodele importante din această clasă, pe lângă cele moştenite de la Protocol, sunt:

**_incFileDone** apelată de instanţa de trimitere a fişierului atunci când destinaţia cu care comunica a primit toate pachetele. Utilizată pentru a contoriza numărul de destinaţii care au primit întreg fişierul, pentru a putea notifica clientul la sfârşitul transferului.

**_getPacketById** apelată de instanţa de trimitere a fişierului pentru a primi un nou pachet. În această metodă se citesc packetSize octeţi din fişier şi se creează un pachet de date, folosind formatul din figura 6.3. Aici, antetul este format din două câmpuri: zero, care reprezintă identificatorul unui pachet de date şi este format din patru octeţi cu valoarea 0; şi seqNr, care reprezintă numărul de secvenţă al pachetului, adică pe ce poziţie va fi scris în fişier.

**_getConfigurations** apelată de ambele destinaţii principale; doar prima dintre ele va primi configuraţiile, pe care le va trimite mai apoi nodului frate şi nodurilor fii.

**fileSenderFactory.py**

MyFileSenderFactory extinde ServerFactory, şi are rolul de a crea instanţe MyFileSenderProtocol.

**fileSenderProtocol.py**

MyFileSenderProtocol extinde LineReceiver şi conţine protocolul de comunicare între serverul sursă şi un server destinaţie (acesta este descris în secţiunea de protocoale).

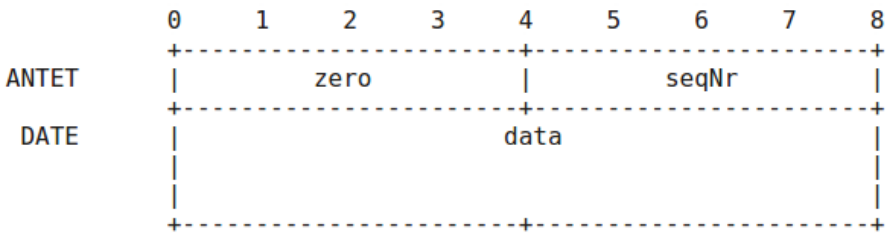



Figura 6.3: Structura unui pachet de date.

### Destinaţie

Fişierele sursă ce formează implementarea serverului destinaţie se află în modulul **server.receiver**. Acestea sunt:

**receiverFactory.py**

MyReceiverFactory extinde ServerFactory, şi are rolul de a crea instanţe MyReceiverProtocol.

**receiverProtocol.py**

MyReceiverProtocol extinde LineReceiver şi conţine protocolul de comunicare între serverul sursă şi serverul destinaţie principală. În această clasă este implementată procesarea pachetelor de date primite folosind algoritmul 2. Metode importante:

**_newConnection** metodă apelată la realizarea unei conexiuni şi folosită pentru a contoriza toate conexiunile existente la un moment dat. Aceasta este necesară deoarece destinaţia principală are rolul de a trimite mai departe pachetele de date pe care le primeşte de la sursă, deci conexiunile trebuie realizate înainte de primirea datelor.

**_processData** această metodă este apelată când se detectează primirea unui pachet de date. Aici se realizează despachetarea acestuia şi se decide dacă pachetul va fi scris direct în fişier sau va fi stocat în fereastra de memorie. Tot aici se hotărăşte ce pachete sunt trimise nodurilor fii.

**multiEchoFactory.py**
MultiEchoFactory extinde Factory, şi este clasa ce creează instanţe MultiEchoProtocol, câte una pentru fiecare destinaţie secundară pe care se doreşte retransmiterea pachetelor primite de la sursă.

**multiEchoProtocol.py**
MultiEchoProtocol extinde LineReceiver şi este folosit pentru a realiza conexiunile cu destinaţiile secundare. Aici, o mare importanţă o au metodele moştenite de la LineReceiver:

**connectionMade** la stabilirea unei conexiuni, aceasta este adăugată în lista de noduri fii.

**connectionLost** la pierderea conexiunii, aceasta este scoasă din listă.

## 6.3 Protocoale de comunicare

În figura 6.4 se poate vedea diagrama de clase a aplicaţiei în raport cu clasele utilizate din Twisted. După cum se poate observa, toate protocoalele de comunicare moştenesc clasa LineReceiver. Acesta oferă posibilitatea de a trimite linii sau date neprelucrate, care sunt terminate de un delimitator. Lungimea maximă a unei linii ce poate fi trimisă este de 16384 octeţi, dar aceasta poate fi modificată pentru a se mapa necesităţilor aplicaţiei. Principalele metode implementate de această clasă:

**lineReceived** apelată la primirea unei linii întregi.

**rawDataReceived** apelată la primirea unui set de date neprelucrate.

**sendLine** apelată pentru a trimite o linie la celălalt capăt al conexiunii.

### 6.3.1 Client - Server

Deoarece în această aplicaţie este vorba de un transfer *third-party*, în primul rând vom avea comunicare între client şi servere. Datorită numărului mare de servere destinaţie, clientul va comunica doar cu sursa, trimiţându-i acesteia informaţiile necesare pentru transfer. Deoarece nu există transfer de date între client şi sursă, acestea vor comunica doar prin comenzi, după cum este descris mai jos. Trebuie remarcat faptul că nu este exclus ca clientul şi sursa să se afle pe aceeaşi maşină fizică. Mai jos sunt descrise comenzile care stau la baza acestei comunicaţii. Comenzile ce au săgeata spre dreapta ($\rightarrow$) sunt trimise de la client la sursă, iar cele ce au săgeata spre stânga ($\leftarrow$) de la sursă la client. Descrierea este făcută din perspectiva clientului, deoarece el este iniţiatorul conversaţiei:

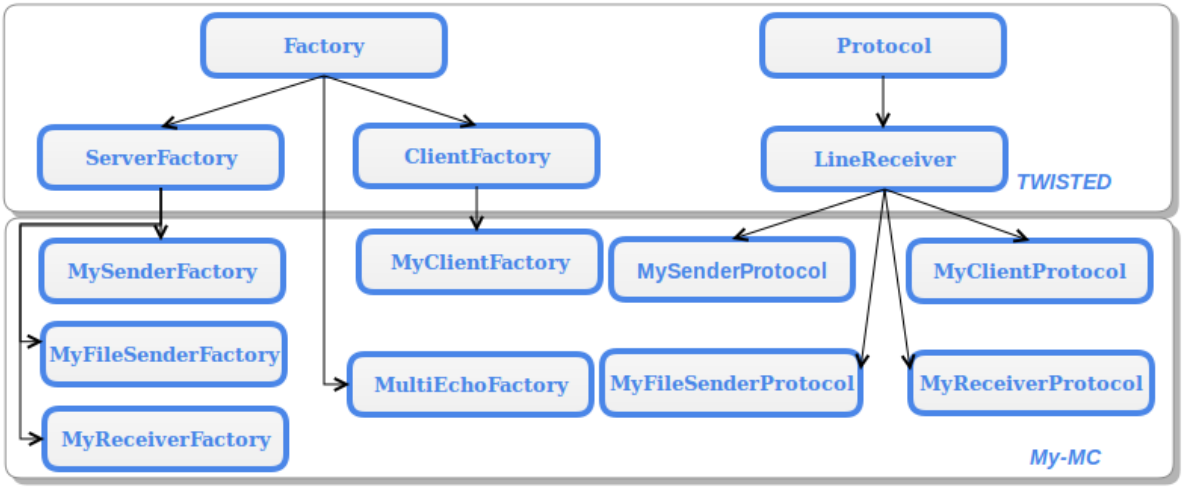



Figura 6.4: Diagrama de clase a aplicaţiei.

**SRFI cale_fisier_sursa** → trimite calea fişierului de pe serverul sursă. La primire, sursa trebuie să verifice existenţa fişierului şi drepturile acestuia de citire, copiere şi modificare.

**DSFI cale_fisier_destinatie** → trimite calea fişierului pentru serverele destinaţie. Acestea vor trebui să verifice drepturile de scriere pentru calea respectivă.

← **FISZ dimensiune_fisier** primeşte dimensiunea fişierului în octeţi, ceea ce înseamnă că fişierul a fost găsit pe serverul sursă, iar drepturile acestuia permit citirea datelor.

**NRDS numar_destinatii** → trimite numărul total de destinaţii către sursă.

**PKSZ dimensiune_pachet** → trimite dimensiunea unui pachet în octeţi.

**WDSZ dimensiune_fereastra** → trimite dimensiunea ferestrei în număr de sloturi.

← **DSOK** primeşte confirmarea de la sursă că s-a alocat suficient spaţiu pentru a memora toate destinaţiile, pentru o ulterioară conectare la acestea.

**DSIP adresa_IP port** → trimite adresa IP şi portul unei destinaţii. Această comandă este trimisă pentru fiecare destinaţie în parte.

← **TRST** primeşte confirmarea începerii transferului. În acest moment, sursa a reuşit să se conecteze la destinaţii şi urmează să trimită fişierul.

← **STAT** primeşte confirmarea terminării transferului. După acest moment, clientul se va deconecta de la serverul sursă.

← **ERRR** această linie este primită doar în cazul apariţiei unei erori pe server: fişierul nu este găsit, comanda primită de server a fost alterată sau eronată, fişierul nu a putut fi deschis pentru citire sau a apărut o eroare internă.

### 6.3.2 Server - Server

După primirea instrucţiunilor de transfer de la client, serverul sursă devine el însuşi client, conectându-se la serverele destinaţie principală, pentru a începe transferul. Protocolul folosit în această sesiune de comunicare este diferit de cel cu clientul şi este descris în paragraful ce urmează. Aici, comenzile cu săgeata orientată spre dreapta (→) sunt trimise de la sursă la destinaţia principală, iar cele cu săgeata orientată spre stânga (←) de la destinaţie la sursă. Naraţiunea este făcută din perspectiva sursei, deoarece ea este iniţiatoarea conversaţiei. De remarcat că sursa comunică doar cu destinaţiile principale:

**SCHM numar_schema** → trimite numărul schemei de distribuţie către destinaţie. Această informaţie este utilă pentru ca destinaţia să ştie câte noduri fii va avea.

**SIBL adresa_IP port** → trimite adresa IP şi portul destinaţiei frate.

**CHLD adresa_IP port** → trimite adresa IP şi portul unei destinaţii cu rol de nod fiu; această comandă va fi trimisă pentru fiecare nod fiu în parte, în funcţie de schema de distribuţie.

← **CONN** primeşte confirmarea de conectare la nodul sursă şi la nodurile fii. Acest pas înseamnă că serverul sursă poate să înceapă să trimită comenzile de configurare a transferului.

**FPTH cale_fisier** → trimite calea fişierului ce va fi trimis. Această informaţie este trimisă mai departe către nodurile fii şi către nodul frate.

**PACK nr_pachete dimens_pachet** → trimite numărul de pachete şi dimensiunea unui pachet de date. Această informaţie face parte din setul de configuraţii, deci va fi trimisă mai departe.

**WDSZ dimensiune_fereastra** → trimite dimensiunea ferestrei de memorie. Şi această comandă este trimisă nodului frate şi nodurilor fii, aceasta fiind ultima configuraţie necesară pentru începerea transferului.

← **SEND** primeşte confirmarea procesării setărilor de configuraţie, ceea ce înseamnă ca transferul poate începe. Acum nodul sursă începe să trimită pachete de date folosind tehnicile descrise în algoritmi.

**REQC** → trimite o cerere de confirmare de primire a tuturor pachetelor de date. Destinaţia poate să răspundă acestei cereri cu una dintre următoarele comenzi.

← **NREC id_pachet** primeşte de la destinaţie o cerere pentru pachetul cu numărul de secvenţă id_pachet, indexul primului pachet lipsă de pe acest server.

← **RECA** primeşte de la destinaţie confirmarea primirii tuturor pachetelor de date. Fişierul este complet.

## 6.4 Calibrare parametrii

Pentru a putea vorbi de eficienţă, trebuie stabilit nivelul la care se face retransmiterea datelor şi care va determina gradul de paralelism în comunicarea dintre diferite noduri. De această granularitate depinde adaptabilitatea protocolului prezentat aici. Cu cât e mai mare dimensiunea unei unităţi de transfer, cu atât mai mică este şi adaptabilitatea. Totuşi, cu cât e mai mică unitatea respectivă, cu atât mai mult se complică mecanismul de evidenţă a pachetelor şi operaţia de transfer. Ideea de bază pentru a rezolva această problemă este că dimensiunea unui pachet de date poate fi crescută/redusă în funcţie de anumiţi factori.

În implementarea propusă există posibilitatea de a lansa un transfer personalizat, prin setarea parametrilor din linia de comandă. Este vorba despre dimensiunea pachetelor şi dimensiunea ferestrei de memorie. Aceşti doi parametrii influenţează timpul de transfer total, iar alegerea unor valori potrivite pentru aceştia este esenţială atunci când se doreşte performanţă.

În acest scop, o serie de transferuri de test au fost realizate. Acestea sunt detaliate în tabelele de mai jos.

În tabelul 6.1 sunt prezentate transferurile realizate pentru determinarea dimensiunii unui pachet. Pentru acestea s-a folosit un fişier de 1GB şi două noduri destinaţie. După cum se poate vedea în tabel, cel mai bun rezultat a fost obţinut pentru valoarea de 1.75MB. Totuşi, trebuie ţinut cont de faptul că sunt mai mulţi factori ce influenţează evoluţia timpului de transfer.

Următorul pas pentru găsirea unei valori corecte a fost varierea dimensiunii fişierului (aceasta a luat valori între 50MB şi 1.6GB) şi a numărului de noduri destinaţie (cu valori între două şi opt destinaţii). După realizarea acestor teste s-a ajuns la concluzia că cei mai buni timpi de transfer sunt dependenţi de dimensiunea pachetelor raportat la dimensiunea totală a fişierului, adică la numărul de pachete. S-a constatat că rezultatele optime se

| Nr. rulare | Dimensiune pachet | Număr pachete | Timp de transfer |
|---|---|---|---|
| 1 | 10.0 MB | 100 | 4m 26s |
| 2 | 5.0 MB | 200 | 2m 56s |
| 3 | 2.5 MB | 400 | 2m 16s |
| 4 | 1.75 MB | 800 | 2m 9s |
| 5 | 1.0 M | 1000 | 2m 18s |
| 6 | 1.5 M | 666 | 2m 18s |
| 7 | 2.0 M | 500 | 2m 16s |

Tabela 6.1: Determinarea dimensiunii optime a unui pachet de date pentru un fişier de 1000 MB.

| Nr. rulare | Dimensiune fereastră | Număr pachete | Timp de transfer |
|---|---|---|---|
| 1 | 200 | 800 | 2m 9s |
| 2 | 300 | 800 | 2m 29s |
| 3 | 100 | 800 | 2m 26s |
| 4 | 150 | 800 | 2m 7s |
| 5 | 250 | 800 | 2m 20s |
| 6 | 220 | 800 | 2m 12s |
| 7 | 210 | 800 | 2m 10s |

Tabela 6.2: Determinarea dimensiunii optime a ferestrei de memorie pentru pachete de 1.75 MB şi fişier de 1000 MB.

obţin atunci când fişierul este divizat în aproximativ 350-450 pachete de date, indiferent de dimensiunea acestora.

Desigur, pentru fişiere foarte mici, nu este recomandat să se folosească această tactică, deoarece se poate ajunge la pachete de dimensiuni mult prea mici. Deşi timpul de transfer în cazul acestor fişiere este oricum destul de scăzut, este recomandabil ca pentru fişierele care au mai puţin de 50MB să nu se coboare dimensiunea unui pachet sub 102400 octeţi (100 KB).

După alegerea unei valori potrivite pentru mărimea pachetelor de date, urmează setarea dimensiunii ferestrei de memorie. În cazul în care aceasta este setată să fie egală cu numărul de pachete, algoritmul de procesare a datelor îşi pierde utilitatea. Pentru dimensiuni prea mari ale acestei ferestre, apare aceeaşi problemă ca şi în cazul lipsei ei: posibilitatea unei erori de memorie cauzată de volumul mare de date. Pentru valori prea mici ale acestui parametru, fereastra se va umple foarte repede, şi foarte multe pachete vor fi aruncate, pentru ca apoi ele să fie retrimise de către sursă. Consecinţa acestui comportament este creşterea timpului de transfer direct proporţional cu scăderea ferestrei.

Din nou, ne confruntăm cu problema găsirii unei situaţii de mijloc, una care să rezolve ambele probleme descrise mai sus. În acest scop, tabelul 6.2 prezintă încercările

efectuate pentru a găsi o valoare optimă pentru fereastra de memorie. Testele au fost rulate pentru un fişier de 1GB şi pachete de 1.75MB. Prin varierea numărului de destinaţii şi a dimensiunii fişierului, s-a ajuns la concluzia că dimensiunea ferestrei depinde de numărul total de pachete. Mai precis, aceasta trebuie să fie aproximativ un sfert din numărul de pachete, pentru a nu cădea în nici una dintre extreme.

Desigur, aceste valori sunt general valabile, dar depind foarte mult de caracteristicile maşinii fizice pe care rulează serverul şi de memoria pe care aceasta o are la dispoziţie. De aceea parametruii pentru dimensiunea ferestrei şi pentru dimensiunea pachetelor sunt încă modificabili, pentru a permite utilizatorului să aleagă cea mai bună soluţie pentru fiecare transfer în parte.

# Capitolul 7

# Testare şi validare

Secţiunile anterioare au detaliat proiectarea şi implementarea unui sistem de replicare a volumelor mari de date în sisteme distribuite. Totuşi, pentru utilizarea acestui sistem, trebuie demonstrată eficienţa acestuia. Demonstraţia se face printr două serii de teste.

## 7.1 Testare funcţională

Primul set de teste are rolul de a demonstra funcţionalitatea sistemului şi de a afla limitele acestuia în funcţie de:

- numărul de destinaţii.
- volumul de date transferate.

Este uşor de observat că prin creşterea acestor parametrii creşte şi timpul de transfer. De aceea, trebuie determinate limitele superioare. Testele efectuate în vederea găsirii volumului maxim de date au funcţionat pentru fişiere de până la 1.7GB, replicate pe trei noduri. Trebuie remarcat faptul că între numărul de destinaţii şi volumul datelor există o relaţie de inversă proporţionalitate: creşterea valorii volumului datelor determină scăderea numărului maxim de destinaţii. Astfel, fişierele relativ mici pot fi replicate pe un număr mare de noduri. Din teste s-a obţinut că replicarea se poate face pe 10 noduri cu fişiere de până la 500MB.

## 7.2 Testare eficienţă

După ce am aflat limitele operaţionale ale sistemului, al doilea set de teste are rolul de a vedea eficienţă acestuia.

Pentru a realiza testarea eficienţei, este necesară o privire comparativă asupra celor trei metode descrise în această lucrare. Prima dată vom vedea cum se comportă soluţia

intitulată ‘distribuţie serială‘. Rezultatele testelor rulate folosind această metodă pot fi văzute în tabelul 7.1 şi în figura 7.1.

| | | Număr destinaţii | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Dimens. fişier | 1MB | 1.243 | 1.125 | 0.944 | 0.957 | 2.145 | 0.963 | 1.849 |
| | 10MB | 2.646 | 3.222 | 3.2 | 3.58 | 6.367 | 5.025 | 9.194 |
| | 50MB | 10.22 | 15.27 | 17.29 | 21.001 | 33.379 | 34.65 | 39.787 |
| | 100MB | 18.183 | 28.365 | 32.252 | 39.618 | 55.078 | 40.584 | 56.662 |
| | 250MB | 89.251 | 114.928 | 151.751 | 190.531 | 206.137 | 235.763 | 276.821 |
| | 500MB | 89.251 | 114.928 | 151.751 | 190.531 | 206.137 | 175.763 | 276.821 |
| | 1GB | 177.007 | 225.607 | 292.612 | 374.827 | 340.272 | 416.349 | 475.435 |

Tabela 7.1: Rezultate algoritm distribuţie serială

Tabelul prezintă timpii de transfer (până ce fişierul e replicat pe toate nodurile) pentru două până la opt destinaţii şi pentru fişiere de 1, 10, 50, 100, 250, 500 respectiv 1000 MB. Graficul 7.1 a fost generat folosind acest tabel. De aici se poate observa că timpul de transfer are o dependenţă liniară faţă de numărul de noduri.

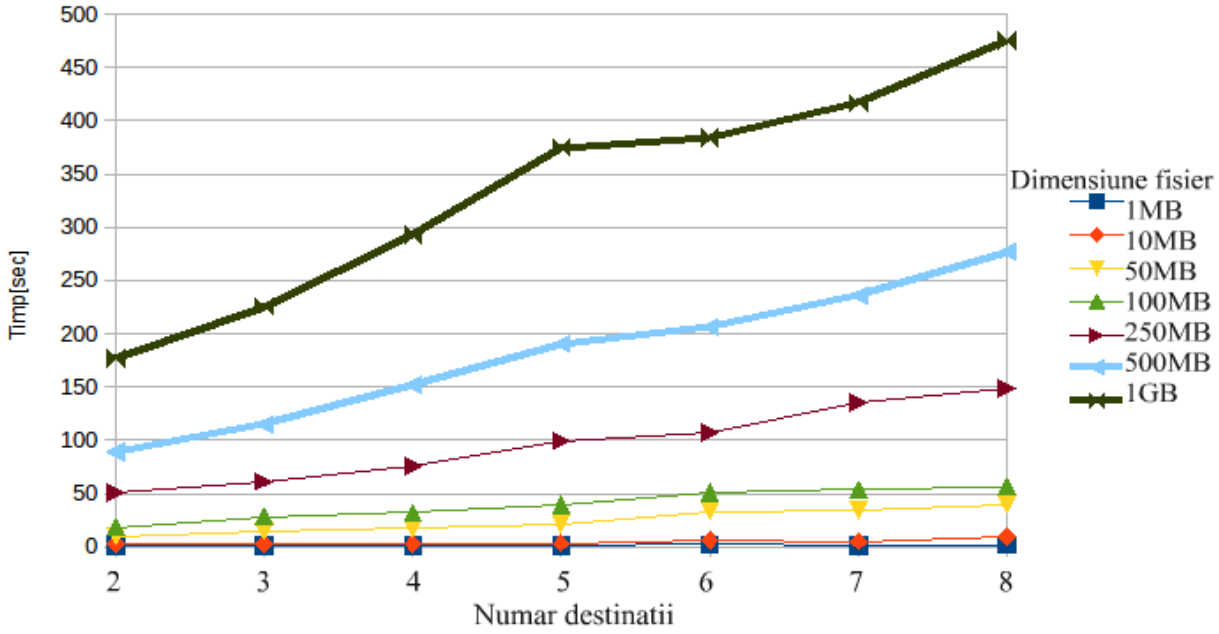



Figura 7.1: Rezultate grafice algoritm distribuţie serială

Cel de-al doilea set de teste este aplicat metodei ‘în doi paşi‘, care este o versiune uşor modificată a *FastReplica*. Pentru aceste teste au fost folosite între trei şi zece destinaţii, iar fişierele transferate au avut dimensiunile de 1, 10, 50, 100, 250 respectiv 500 MB.

| | | Număr destinaţii | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Dimens. fişier | 1MB | 1.54 | 1.41 | 1.31 | 1.21 | 1.13 | 1.10 | 1.22 | 1.48 |
| | 10MB | 5.34 | 4.25 | 4.27 | 3.88 | 3.68 | 3.49 | 3.36 | 3.56 |
| | 50MB | 22.94 | 17.92 | 18.2 | 17.66 | 16.749 | 15.74 | 17.22 | 18.27 |
| | 100MB | 72.17 | 55.94 | 47.61 | 42.26 | 39.37 | 36.82 | 47.23 | 56.012 |
| | 250MB | 442.82 | 326.67 | 265.22 | 209.9 | 177.8 | 162.99 | 292.88 | 382.69 |
| | 500MB | 1467.248 | 1326.25 | 1140.05 | 898.77 | 762.46 | 669.8 | 702.6 | 956.78 |

Tabela 7.2: Rezultate algoritm în doi paşi

Rezultatele acestor teste pot fi studiate în tabelul 7.2, respectiv în figura 7.2. Timpii de transfer sunt prezentaţi în tabel, iar pentru o mai bună înţelegere a eficienţei se poate consulta graficul generat de aceştia.

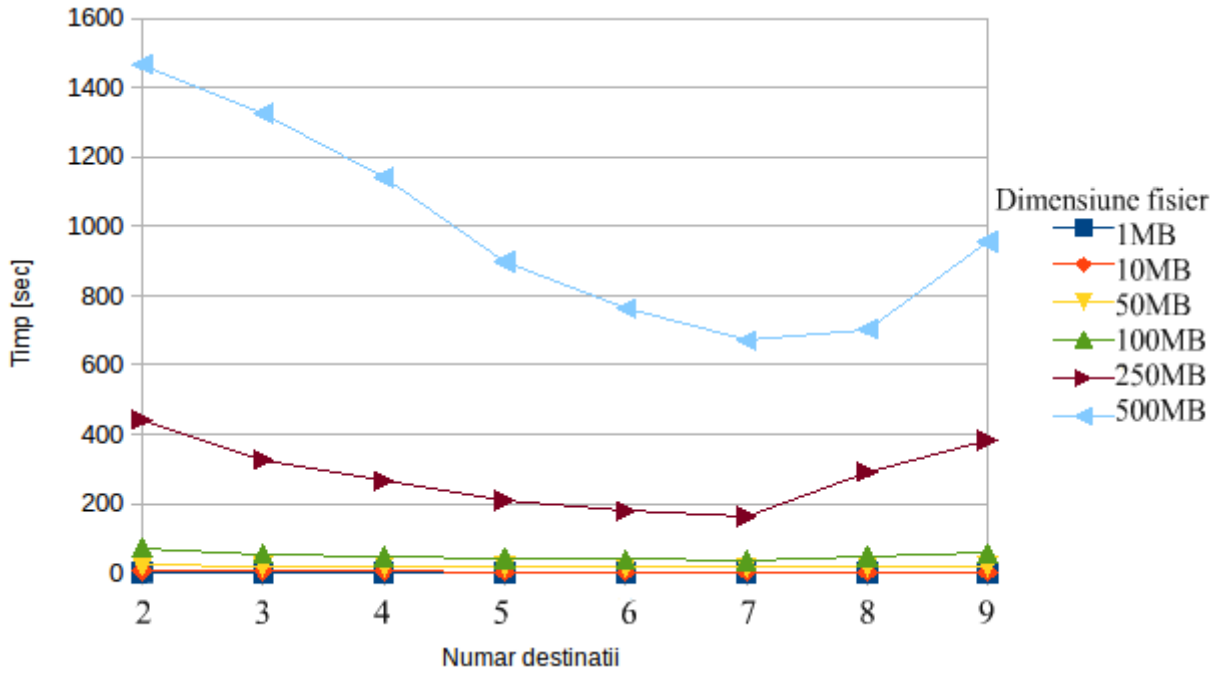



Figura 7.2: Rezultate grafice algoritm în doi paşi

În grafic se poate vedea că timpii de transfer sunt relativi mari pentru un număr mic de noduri, scăzând până la un anumit punct, pentru ca apoi să crească din nou. Acest comportament se explică prin faptul că în această metodă fişierul este împărţit într-un număr de subfişiere egal cu numărul de destinaţii. De aceea, pachetel de date, care aici se numesc subfişiere, au dimensiuni mari, şi necesită un timp de transfer individual foarte mare. Totuşi, pentru valori potrivite ale numărului de noduri şi ale volumului de date, această metodă este eficientă.

| | | Număr destinaţii | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Dimens. fişier | 50MB | 5.958 | 8.919 | 9.38 | 9.284 | 12.055 | 11.635 | 11.935 | 34.473 |
| | 100MB | 12.095 | 20.443 | 18.524 | 21.777 | 26.799 | 29.875 | 38.17 | 79.463 |
| | 250MB | 32.051 | 38.427 | 38.816 | 38.796 | 85.775 | 58.06 | 71.024 | 203.35 |
| | 500MB | 62.994 | 80.404 | 91.775 | 101.202 | 118.335 | 116.21 | 188.571 | 341.92 |
| | 1GB | 129.542 | 174.297 | 188.29 | 215.158 | 338.338 | 320.54 | 336.817 | 666.54 |

Tabela 7.3: Rezultate algoritm MC-5.

Ultima soluţie testată este MC-5. Pentru aceasta s-au folosit între două şi nouă noduri destinaţie, iar pentru volumul de date 50, 100, 250, 500 respectiv 1000 MB. Rezultatele pentru această metodă sunt ilustrate în tabelul 7.3, respectiv în figura 7.3.

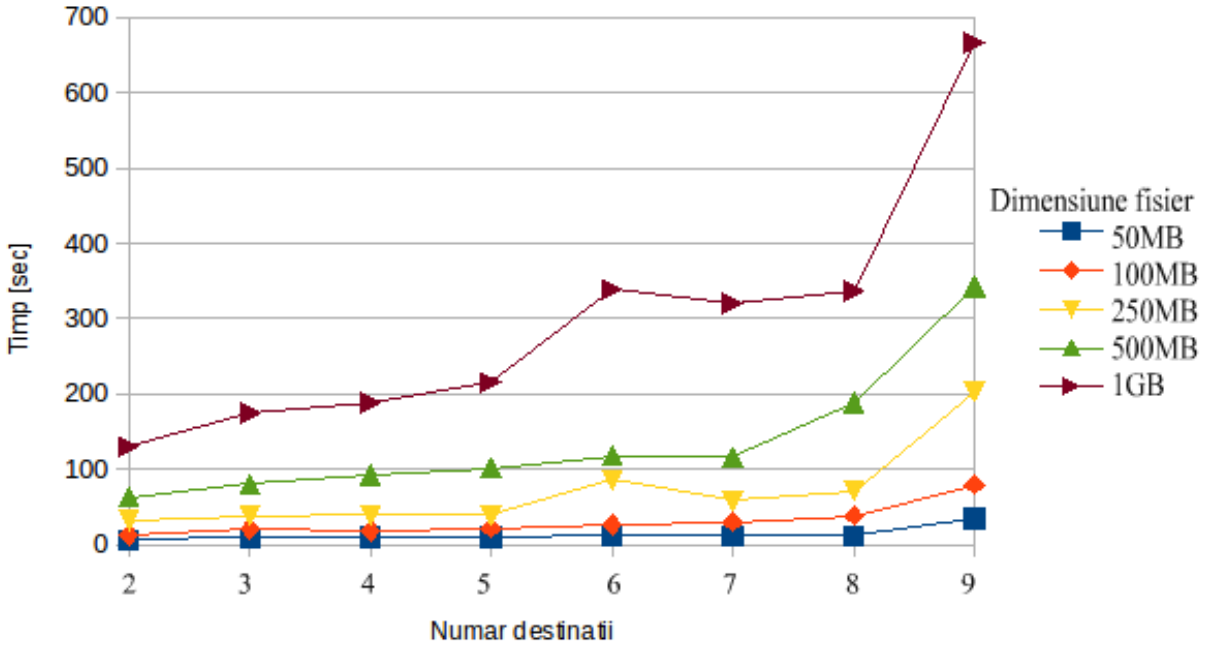
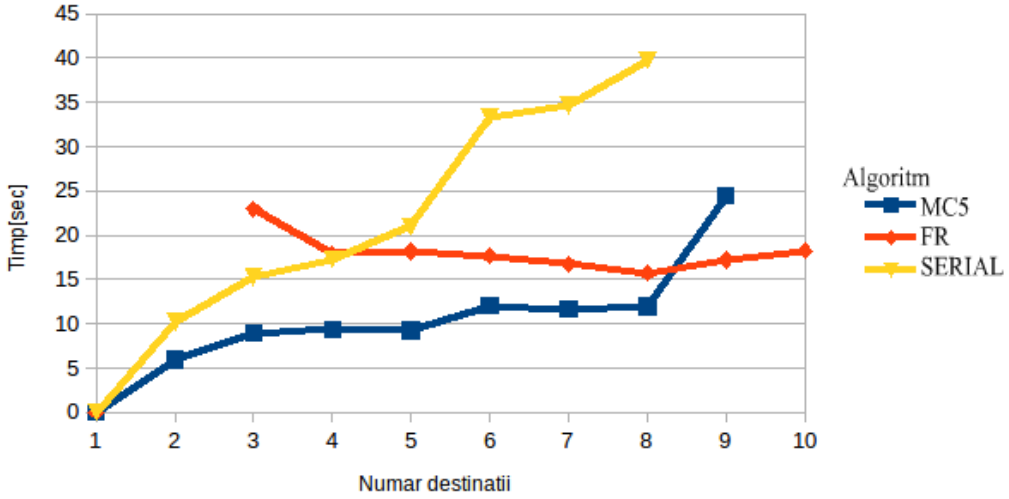



Figura 7.3: Rezultate grafice algoritm MC-5.

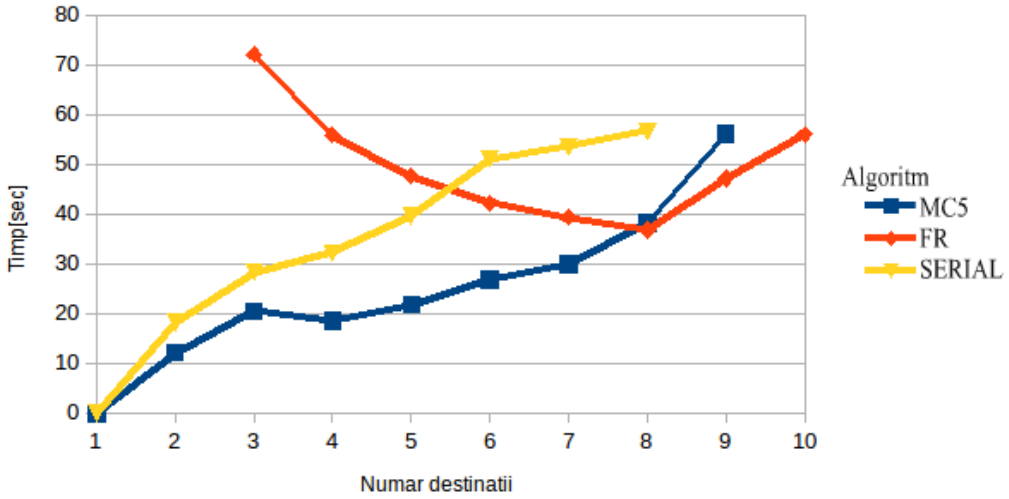
Tabelul conţine, ca şi în celelalte cazuri, timpii de transfer pentru valorile specificate. Analiza graficului arată că soluţia aceasta înregistrează o creştere foarte mică a timpilor de transfer până ce se ajunge la şase destinaţii. Acest comportament apare din cauza structurii grupurilor de replicare pe baza cărora funcţionează MC-5.

## 7.3 Comparare implementări

În figurile 7.4(a) şi 7.4(b) se pot vedea comparativ performanţele fiecărei soluţii descrise.




(a) Comparaţie pentru fişier de 50MB.




(b) Comparaţie pentru fişier de 100MB.

Figura 7.4: Compararea algoritmilor MC-5, FR şi SERIAL pentru diferite dimensiuni de fişiere.

Soluţia propusă în această lucrare este denumită MC-5, cea în doi paşi este denumită FR, iar cea de distribuţie serială SERIAL. Este uşor de observat din aceste grafice că MC-5 reprezintă o soluţie bună din punct de vedere al timpilor de transfer. Comparativ cu SERIAL, această abordare are timpi de transfer cu până la 70% mai buni, iar în medie, transferurile sunt mai rapide folosind MC-5. Faţă de FR, soluţia MC-5 nu este mereu mai bună, dar totuşi obţine timpi de transfer mai mici în aproximativ 80% din cazuri.

# Capitolul 8

# Manual de instalare şi utilizare

Acest capitol conţine instrucţiunile de instalare şi utilizare ale aplicaţiei prezentate. De asemenea, pentru un caz de utilizare, sunt prezentate şi rezultatele aşteptate în cazul unei execuţii fără erori.

## 8.1 Instalare

### 8.1.1 Resurse hardware

Ca şi resurse hardware, este necesar un sistem de calcul care are minim următoarele configuraţii:

- procesor dual-core
- 2GB de memorie RAM
- 50MB spaţiu de stocare pe disc
- placă de reţea

### 8.1.2 Resurse software

Pentru a putea rula aplicaţia prezentată în lucrarea de faţă, următoarele resurse software sunt necesare:

- sistem de operare Ubuntu 12.04 LTS pe 32 biţi
- Python 2.7.4

```
adelaneacsu@ubuntu:~$ sudo apt-get update
adelaneacsu@ubuntu:~$ sudo apt-get install -y python2.7
```

- Twisted 13.0.0

```
adelaneacsu@ubuntu:~$ sudo apt-get install python-twisted
```

### 8.1.3 Instrucţiuni instalare

Codul sursă al algoritmilor prezentaţi aici se află în arhiva 'my-protocols.tar.gz', ce poate fi dezarhivată folosind comanda următoare:

```
adelaneacsu@ubuntu:~$ tar -xzf my-protocols.tar.gz
```

În interiorul acestei arhive se află alte trei arhive, câte una pentru fiecare soluţie descrisă aici. Acestea pot fi dezarhivate în acelaşi mod ca mai sus. Punctul de interes este 'my-mc.tar.gz', în care se află soluţia propusă în acest document.

## 8.2 Manual de utilizare

În primul rând trebuie precizat că pentru realizarea unui transfer folosind această aplicaţie sunt necesare cel puţin trei maşini virtuale: una pentru lansarea clientului şi a sursei şi două pentru serverele destinaţie. Nu este obligatoriu ca acestea să fie pe maşini fizice diferite, dar acest lucru este de preferat ţinând cont că se doreşte ca nodurile să fie dispersate.

Pentru pornirea unui server, indiferent dacă acesta va juca rol de sursă sau de destinaţie, se foloseşte:

```
adelaneacsu@ubuntu:~$ python my-mc/server/mainServer.py
```

Înainte de a lansa un client în execuţie, trebuie creat fişierul de multicast, ce conţine adresele IP ale serverelor destinaţie. Pentru pornirea clientului, se foloseşte următoarea comandă minimală (argumentele ce apar aici sunt necesare):

```
adelaneacsu@ubuntu:~$ python my-mc/client/mainClient.py
  -sf \textit{path_to_source_file}
  -df \textit{path_to_destination_file}
  -mc \textit{path_to_multicast_file}
```

Dacă se doreşte măsurarea timpului de rulare a procesului client, se poate folosi comanda *time* din Linux:

```
adelaneacsu@ubuntu:~$ time python my-mc/client/mainClient.py .
```

Comenzile de mai sus pot primi argumente suplimentare pentru a eficientiza şi pentru a personaliza transferul bazat pe cunoştinţele utilizatorului despre reţea. De exemplu, la realizarea fişierului de multicast, adresele IP ale nodurilor destinaţie care se află în

aceeaşi reţea locală (LAN) pot fi scrise succesiv, pentru a fi conectate în acelaşi grup de replicare.

De asemenea, alegerea dimensiunii pachetelor joacă un rol important în eficientizarea transferului, după cum a fost demonstrat în capitolele anterioare. Există şi alte argumente ce pot fi folosite după placul utilizatorului; important este să se înţeleagă conceptele care stau la baza acestuia.

## 8.3 Exemplu de utilizare

Pentru acest exemplu, avem la dispoziţie maşinile virtuale din tabelul 8.1.

| ID | Adresă IP | Nume | Locaţie |
|---|---|---|---|
| ADE | 193.226.6.229 | - | Cluj, România |
| CJ | 193.226.5.102 | gt-ige.utcluj.ro | Cluj, România |
| MA | 147.96.25.28 | gridway.fdi.ucm.es | Madrid, Spania |
| LY | 134.158.75.91 | vm-91.lal.stratuslab.eu | Lyon, Franţa |
| AT | 62.217.122.13 | vm013.grnet.stratuslab.eu | Athena, Grecia |

Tabela 8.1: Maşini virtuale disponibile pentru teste.

Considerăm maşina locală ADE, de pe care va fi lansat clientul. Tot aceasta va constitui şi sursa, deci aici se află fişierul ce dorim să-l replicăm, numit 'replica.file'. Primul pas este crearea fişierului de multicast numit aici 'mc.file'. Acesta poate avea forma:

```
193.226.5.102
147.96.25.28:22000
134.158.75.91
62.217.122.13:22000
```

Pe maşinile virtuale CJ, MA, LY şi AT se pornesc serverele, folosind instrucţiunile de mai sus. Comanda pentru lansarea clientului:

```
adelaneacsu@ubuntu:~$ time python my-mc/client/mainClient.py
  -sf replica.file -df copy.file -mc mc.file
```

Răspunsul aşteptat este în stilul acesta:

```
Connection made: 127.0.0.1 on port 22020
The file has 524288000 Bytes of data. Starting transfer ...
Transfer fully completed. Connection ended.

real    7m2.950s
user    0m0.140s
sys     0m0.036s
```

# Capitolul 9

# Concluzii

## 9.1 Analiza rezultatelor obţinute

Un sistem distribuit a fost dezvoltat conform specificaţiilor iniţiale ce permite replicarea volumelor mari de date de pe un nod sursă pe un set de noduri destinaţie. Acesta se bazează pe o abordare de tip *divide et impera* şi presupune împărţirea setului de destinaţii în seturi mai mici, care sunt mai uşor de analizat şi de manevrat.

Sistemul respectă atât cerinţele funcţionale, cât şi pe cele legate de eficienţă, scalabilitate, adaptabilitate şi fiabilitate. Prin experimente repetate, s-a demonstrat că abordarea folosită aici produce rezultate bune comparativ cu alte soluţii existente. Performanţa este atinsă prin folosirea unor algoritmi ce permit replicarea chiar şi în cazul volumelor foarte mari de date.

În concluzie, obiectivele descrise la începutul lucrării au fost îndeplinite cu succes. Analiza rezultatelor testării dovedeşte că abordarea propusă are un potenţial mare de dezvoltare şi de extindere, ceea ce face din soluţia aceasta o bază solidă pentru crearea unui sistem mai complex, îmbunătăţit.

## 9.2 Dezvoltări ulterioare

Deşi soluţia propusă şi-a îndeplinit scopul pentru care a fost creată, în acest domeniu există mereu loc de îmbunătăţiri. De aceea, în această secţiune sunt propuse posibilităţi de dezvoltări ulterioare bazate pe sistemul prezentat în lucrarea de faţă:

- extinderea protocoalelor de comunicare pentru a permite transferul mai multor fişiere simultan.
- implementarea unui sistem complementar ce permite aducerea unui fişier de la mai multe surse concurent.
- protejarea sistemului împotriva atacurilor maliţioase prin metode de autentificare şi prin securizarea conexiunilor.

- folosirea mai multor conexiuni TCP paralele între două noduri pentru a creşte rata de date trimise.
- dezvoltarea unor algoritmi ce permit calcularea valorilor optime pentru pachetele de date şi pentru fereastra de memorie.

Structura modulară a sistemului prezentat aici permite realizarea unora dintre aceste propuneri fără modificarea codului existent. Totuşi, pentru anumite puncte, este necesară o nouă analiză a problemelor şi a soluţiilor posibile.

# Bibliografie


[1] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: acm, 2002, pp. 455–465.

[2] J. Lee and L. Cherkasova, "Fastreplica: Efficient large file distribution within content delivery networks," in *In 4th USENIX Symposium on Internet Technologies and Systems*. USENIX Association, 2003, p. 7.

[3] L. Cherkasova, "Optimizing the reliable distribution of large files within cdns," in *ISCC '05: Proceedings of the 10th IEEE Symposium on Computers and Communications*. IEEE Computer Society, 2005, p. 7.

[4] W. Allcock, "Gridftp protocol specification (global grid forum recommendation gfd.20)," *International Journal of High Performance Computing Applications: Special issue on Grid Computing: Infrastructure and Applications*, Mar. 2003. [Online]. Available: http://www.globus.org/alliance/publications/papers/GFD-R.0201.pdf

[5] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, and J. Saltz, "A dynamic scheduling approach for coordinated wide-area data transfers using gridftp," in *Proceedings of the 2008 IEEE international parallel and distributed processing symposium*. IEEE Computer Society, 14-18 April 2008, pp. 1 – 12.

[6] T. Kurc, R. Kettimuthu, P. Sadayappan, J. Saltz, G. Khanna, and U. Catalyurek, "Using overlays for efficient data transfer over shared wide-area networks," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Computer Society, 15-21 Nov. 2008, pp. 1 – 12.

[7] A. N. Laboratory. (2013) Gridftp multicast. Internet page last accesed on June 2013. [Online]. Available: http://www.mcs.anl.gov/ bresnaha/mc/

[8] W. Allcock, J. B. R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *In SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[9] R. Izmailov, S. Ganguly, and N. Tu, "Fast parallel file replication in data grid," in *In Future of Grid Data Environments workshop (GGF-10)*, 2004.

[10] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: A distributed resource performance forecasting service for metacomputing," *Journal of Future Generation Computing Systems*, vol. 15, pp. 757–768, 1999.

[11] A. Vakali and G. Pallis, "Content delivery networks: status and trends," *Internet Computing, IEEE*, pp. 68 – 74, Nov.-Dec. 2003.

[12] W. F. Inc. (2013) Content delivery network. Internet page last modified on 22 June 2013. [Online]. Available: http://en.wikipedia.org/wiki/Content_delivery_network

[13] I. S. I. . U. of Southern California, "TRANSMISSION CONTROL PROTOCOL," RFC 793, Internet Engineering Task Force, Sep. 1981. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

# Capitolul 10

# Anexe

```
import logging

from twisted.protocols.basic import LineReceiver

class MyClientProtocol(LineReceiver):

    def __init__(self, factory):
        self.factory = factory

    def connectionMade(self):
        msg = 'Connection made: %s' % self.transport.getPeer()
        logging.info(msg)
        print 'Connection made: %s on port %d' % (self.transport.
   getPeer().host, self.transport.getPeer().port)

        line = 'SRFI ' + self.factory.sourceFile
        self.sendLine(line)
        line = 'DSFI ' + self.factory.destFile
        self.sendLine(line)


    def lineReceived(self, line):
        print line
        data = line.strip().split(' ')
        if data[0] == 'FISZ':
            # file size
            logging.info('File ' + self.factory.sourceFile + ' has '
   + data[1] + ' bytes.')
            size = int(data[1])
            if size == 0:
                logging.info('Transfer stopped.')
```

```
                print 'The file does NOT exist on the source server
    or it does not have reading rights.'
                print 'Communication with the source server will now
    stop.'
                self.factory.callback._stop()
            else:
                print 'The file has %d Bytes of data.' % size
                sLine = 'NRDS ' + str(len(self.factory.destinations))
                self.sendLine(sLine)
                sLine = 'PKSZ ' + str(self.factory.packetSize)
                self.sendLine(sLine)
                sLine = 'WDSZ ' + str(self.factory.windowSize)
                self.sendLine(sLine)

        elif data[0] == 'DSOK':
            # received number of destinations
            for dst in self.factory.destinations:
                sLine = 'DSIP %s %d' % (dst[0], dst[1])
                self.sendLine(sLine)

        elif data[0] == 'TRST':
            # transfer started
            msg = 'Transfer started ...'
            logging.info(msg)
            print msg

        elif data[0] == 'STAT':
            # statistics
            msg = 'Transfer fully completed. Connection ended.'
            print msg
            logging.info(msg)
            self.factory.callback._stop()

        elif data[0] == 'ERRR':
            msg = 'An error ocurred. Connection ended.'
            print msg
            logging.info(msg)
            self.factory.callback._stop()
```

clientProtocol.py

```
import logging
import struct
import time
```

```
from twisted.protocols.basic import LineReceiver
from twisted.internet.protocol import ServerFactory
from twisted.internet.endpoints import TCP4ClientEndpoint
from twisted.internet import reactor, defer

from multiEchoFactory import *


class MyReceiverProtocol(LineReceiver):

    def __init__(self, factory):
        self.factory    = factory
        self.MAX_LENGTH = 524288010

    def connectionMade(self):
        if self.factory.source is None:
            #print 'source is %d' % self.transport.getPeer().port
            self.factory.source = self
        logging.info('Connection made: %s' % self.transport.getPeer()
    )

    def connectionLost(self, reason):
        if self.factory.source == self:
            self.factory.source = None
            for echoer in self.factory.echoFactory.echoers:
                echoer.transport.loseConnection()
        logging.info('Connection lost: %s' % self.transport.getPeer()
    )

    def lineReceived(self, line):
        if line[0:4] == '0000':
            # this is a packet - store and forward
            if self.factory.source == self:
                for echoer in self.factory.echoFactory.echoers:
                    echoer.sendLine(line)
            return self._processData(line[4:])
        else:
            print 'rc = %s %s' % (line ,self.transport.getPeer().host
    )
            data = line.strip().split(' ')
            if data[0] == 'SCHM':
                # SCHM number of schema (0, 1, 2 or 3)
                try:
                    self.factory.nrConnections = int(data[1]) + 1
```

```
                except IndexError:
                    self.sendLine('ERRR')
                    return

            elif data[0] == 'SIBL' or data[0] == 'CHLD':
                # SIBL/CHLD IP port
                try:
                    IP = data[1]
                    port = int(data[2])
                    endpoint = TCP4ClientEndpoint(reactor, IP, port)
                    endpoint.connect(self.factory.echoFactory).
addCallback(self._newConnection)
                except IndexError:
                    self.sendLine('ERRR')
                    return

            elif data[0] == 'FPTH':
                # FPTH filepath
                try:
                    self.factory.fileObj = open(data[1], 'wb')
                    self.factory.nextToWrite = 0
                    logging.info('File %s opened for writing. Raw
mode set.' % data[1])
                except IndexError:
                    self.sendLine('ERRR')
                    return
                # start statistics
                self.factory.startTime  = time.time()
                self.factory.minTime    = 1000000
                self.factory.maxTime    = 0
                self.factory.avgTime    = 0
                self.factory.nrPacksRec = 0
                # forward configurations
                if self.factory.source == self:
                    for echoer in self.factory.echoFactory.echoers:
                        echoer.sendLine(line)

            elif data[0] == 'PACK':
                # PACK nrPacks sizePack
                try:
                    self.factory.nrPackets = int(data[1])
                    self.factory.packetSize = int(data[2])
                    self.factory.packetsReceived = [False] * self.
factory.nrPackets
```

```
                except IndexError:
                    self.sendLine('ERRR')
                    return
                if self.factory.source == self:
                    for echoer in self.factory.echoFactory.echoers:
                        echoer.sendLine(line)

            elif data[0] == 'WDSZ':
                # WDSZ windSz
                try:
                    self.factory.windowSize = int(data[1])
                    self.factory.window = [""] * self.factory.
windowSize
                    # -1 if the slot is free, packet index otherwise
                    self.factory.slotBusy = [-1] * self.factory.
windowSize
                except IndexError:
                    self.sendLine('ERRR')
                    return
                if self.factory.source == self:
                    for echoer in self.factory.echoFactory.echoers:
                        echoer.sendLine(line)
                self.factory.source.sendLine('SEND')

            elif data[0] == 'REQC':
                # REQC -- check of all packets arrived
                allSent = True
                for bitIndex in range(len(self.factory.
packetsReceived)):
                    if not self.factory.packetsReceived[bitIndex]:
                        allSent = False
                        self.sendLine('NREC ' + str(bitIndex))
                        break
                if allSent:
                    self.sendLine('RECA')


    def _processData(self, rawData):
        logging.info('Received %d bytes.' % (len(rawData) - 4))
        # split : header + data
        currIndex = int(rawData[0:4])
        print 'PACK %d' % currIndex
        # statistics
        self.factory.nrPacksRec += 1
```

```
        currTime = time.time()
        delta = currTime - self.factory.startTime
        if delta < self.factory.minTime:
            self.factory.minTime = delta
        elif delta > self.factory.maxTime:
            self.factory.maxTime = delta
        self.factory.avgTime += delta
        self.factory.startTime = currTime

        if self.factory.packetsReceived[currIndex] == False:
            # only keep first copy of each packet, since they might
arrive on more than one links
            self.factory.packetsReceived[currIndex] = True
            if self.factory.nextToWrite == currIndex:
                # current packet must be written directly to file
                self.factory.fileObj.write(rawData[4:])
                self.factory.nextToWrite += 1
                countdown = self.factory.windowSize
                done = False
                # also write all packets that come after
                while not done:
                    for slotIndex in range(self.factory.windowSize):
                        countdown -= 1
                        if self.factory.slotBusy[slotIndex] == self.
factory.nextToWrite:
                            self.factory.fileObj.write(self.factory.
window[slotIndex])
                            self.factory.slotBusy[slotIndex] = -1
                            self.factory.nextToWrite += 1
                            countdown = self.factory.windowSize
                        if countdown == 0:
                            done = True
            else:
                # find an empty slot to store data
                for slotIndex in range(self.factory.windowSize):
                    if self.factory.slotBusy[slotIndex] == -1:
                        self.factory.window[slotIndex] = rawData[4:]
                        self.factory.slotBusy[slotIndex] = currIndex
                        break

            if self.factory.nextToWrite == self.factory.nrPackets:
                # file was completely written
                self.factory.fileObj.close()
                if len(self.factory.echoFactory.echoers) == 0:
```

```
                    self.sendLine('CHRA')

    def _newConnection(self, deffered):
        if len(self.factory.echoFactory.echoers) == self.factory.
    nrConnections:
            # all nodes are connected
            self.sendLine('CONN')
```

receiverProtocol.py

```
import os
import logging

from multiprocessing.connection import Client
from twisted.internet.protocol import ServerFactory
from twisted.protocols.basic import LineReceiver


class MyFileSenderProtocol(LineReceiver):

    def __init__(self, factory):
        self.factory = factory

    def connectionMade(self):
        self.sendLine('SCHM ' + str(len(self.factory.children)))
        self.sendLine('SIBL ' + self.factory.sibling[0] + ' ' + str(
    self.factory.sibling[1]))
        for child in self.factory.children:
            self.sendLine('CHLD ' + child[0] + ' ' + str(child[1]))
        logging.info('Connection made: %s' % self.transport.getPeer()
    )

    def connectionLost(self, reason):
        logging.info('Connection lost: %s' % self.transport.getPeer()
    )

    def lineReceived(self, line):
        print 'fs = %s %s' % (line ,self.transport.getPeer().host)
        data = line.strip().split(' ')
        if data[0] == 'CONN':
            # get configuration data
            if self.factory.parent._getConfigurations(self.factory.
    groupNr) == 1:
                # first connected
                self.sendLine('FPTH ' + self.factory.parent.dstpath)
```

```
                nrPackets = self.factory.parent.fileSize / self.
    factory.parent.packetSize
                if self.factory.parent.fileSize % self.factory.parent
    .packetSize != 0:
                    nrPackets += 1
                self.sendLine('PACK ' + str(nrPackets) + ' ' + str(
    self.factory.parent.packetSize))
                self.sendLine('WDSZ ' + str(self.factory.parent.
    windowSize))

            elif data[0] == 'SEND':
                # server received configurations, now is asking for data
                packet = self.factory.parent._getPacketById(self.factory.
    parity)
                k = 2
                while packet != -1:
                    self.sendLine(packet)
                    packet = self.factory.parent._getPacketById(self.
    factory.parity + k)
                    k += 2
                self.sendLine('REQC')

            elif data[0] == 'NREC':
                # at least one packet did not arrive to the destination,
    resend it
                packet = self.factory.parent._getPacketById(int(data[1]))
                self.sendLine(packet)
                self.sendLine('REQC')

            elif data[0] == 'RECA':
                # all file was received
                print '%s %s' % (line ,self.transport.getPeer().host)
                self.factory.parent._incFileDone(self)
```

fileSenderProtocol.py

```
import os
import logging

from twisted.protocols.basic import LineReceiver
from twisted.internet.endpoints import TCP4ClientEndpoint,
    TCP4ServerEndpoint
from twisted.internet import reactor

from fileSenderFactory import *
```

```
class MySenderProtocol(LineReceiver):

    def __init__(self):
        self.directPeers = []
        logging.info('Server started.')

    def connectionMade(self):
        self.fileDoneCnt = 0
        self.destinations = []
        self.client = self.transport.getPeer()
        logging.info('Connection made: %s' % self.client)

    def connectionLost(self, reason):
        if hasattr(self, 'client'):
            logging.info('Connection ended from: %s' % self.client)
        else:
            logging.error('Connection ended unexpectedly. Reason: %s'
    % reason)

    def lineReceived(self, line):
        #print 'sd = %s %d' % (line ,self.transport.getPeer().port)
        data = line.strip().split(' ')
        if data[0] == 'SRFI':
            try:
                self.filepath = data[1]
            except IndexError:
                self.sendLine('ERRR')
                return
            if not os.path.exists(self.filepath):
                self.sendLine('FISZ 0')
                return
            self.fileSize = os.stat(self.filepath).st_size
            self.sendLine('FISZ ' + str(self.fileSize))
            try:
                self.fileObj = open(self.filepath, 'rb')
            except IOError:
                self.sendLine('ERRR')
                return

        elif data[0] == 'DSFI':
            try:
                self.dstpath = data[1]
            except IndexError:
```

```
            self.sendLine('ERRR')
            return

        elif data[0] == 'NRDS':
            try:
                self.nrDestinations = int(data[1])
            except IndexError:
                self.sendLine('ERRR')
                return
            # decide distribution schema
            self.schemaIndices = [0, 0, 0, 0] # 2, 3, 4, 5
            if self.nrDestinations % 4 == 0:
                self.schemaIndices[2] = self.nrDestinations / 4
            elif self.nrDestinations % 4 == 1:
                self.schemaIndices[2] = self.nrDestinations / 4 - 1
                self.schemaIndices[3] = 1
            elif self.nrDestinations % 4 == 2:
                k = self.nrDestinations / 4
                if k == 0:
                    self.schemaIndices[0] = 1
                elif k == 1:
                    self.schemaIndices[1] = 2
                else:
                    self.schemaIndices[2] = k - 2
                    self.schemaIndices[3] = 2
            elif self.nrDestinations % 4 == 3:
                k = self.nrDestinations / 4
                if k == 0:
                    self.schemaIndices[1] = 1
                elif k == 1:
                    self.schemaIndices[1] = 1
                    self.schemaIndices[2] = 1
                elif k == 2:
                    self.schemaIndices[1] = 1
                    self.schemaIndices[2] = 2
                else:
                    self.schemaIndices[2] = k - 3
                    self.schemaIndices[3] = 3
            self.sendLine('DSOK')

        elif data[0] == 'PKSZ':
            try:
                self.packetSize = int(data[1])
            except IndexError:
```

```
                self.sendLine('ERRR')
                return

        elif data[0] == 'WDSZ':
            try:
                self.windowSize = int(data[1])
            except IndexError:
                self.sendLine('ERRR')
                return

        elif data[0] == 'DSIP':
            try:
                destIP = data[1]
                destPort = int(data[2])
            except IndexError:
                self.sendLine('ERRR')
                return
            dst = [destIP, destPort]
            index = len(self.destinations)
            self.destinations.append(dst)

            if index == self.nrDestinations - 1:
                start = 0
                group = 0
                for i in range(len(self.schemaIndices)):
                    for k in range(0, self.schemaIndices[i]):
                        lc = TCP4ClientEndpoint(reactor, self.
destinations[start][0], self.destinations[start][1])
                        rc = TCP4ClientEndpoint(reactor, self.
destinations[start+1][0], self.destinations[start+1][1])
                        lc.connect(MyFileSenderFactory(self, self.
destinations[start+1], self.destinations[start+2:start+2+i], group
, 0))
                        rc.connect(MyFileSenderFactory(self, self.
destinations[start], self.destinations[start+2:start+2+i], group,
1))
                        start += (2 + i)
                        group += 1
                self.configs = [0] * group

    def _incFileDone(self, peer):
        self.fileDoneCnt += 1
        logging.info('Transfer to %s completed.' % peer)
        if self.fileDoneCnt == self.nrDestinations:
```

```
            finishMsg = 'All destinations received the file. Transfer
    fully completed.'
            logging.info(finishMsg)
            print finishMsg
            for cPeer in self.directPeers:
                cPeer.transport.loseConnection()
            self.sendLine('STAT')
        else:
            self.directPeers.append(peer)

    def _getPacketById(self, packetId):
        offset = packetId * self.packetSize
        chunk = ''
        if self.fileSize > offset:
            # there is still data to read from file
            currOffset = self.fileObj.tell()
            if currOffset != offset:
                self.fileObj.seek(offset)
            chunk = '0000' + str(packetId).zfill(4) + self.fileObj.
    read(self.packetSize)
            return chunk
        else:
            return -1

    def _getConfigurations(self, groupNumber):
        if self.configs[groupNumber] == 0:
            self.configs[groupNumber] = 1
            return 1
        else:
            return -1
```

senderProtocol.py