ALGORITMI PARALELI ȘI DISTRIBUIȚI
Tema #2 Protocolul CHORD în MPI

Responsabili: Radu-Ioan CIOBANU, George Alexandru TUDOR,
Alexandra Ana-Maria ION, Ion-Dorinel FILIP

Termen de predare: 15 Ianuarie 2025 - 23:59 (soft), 18 Ianuarie 2026 - 23:59 (hard)

Cuprins

1 Introducere și scopul temei

În această temă, veți implementa o versiune simplificată a protocolului **CHORD** folosind MPI, în care fiecare proces MPI joacă rolul unui nod dintr-un inel distribuit. Scopul este să înțelegeți cum poate fi realizat un mecanism de căutare distribuit, fără nod central, construit în jurul unei structuri logaritmice numite **finger table**.

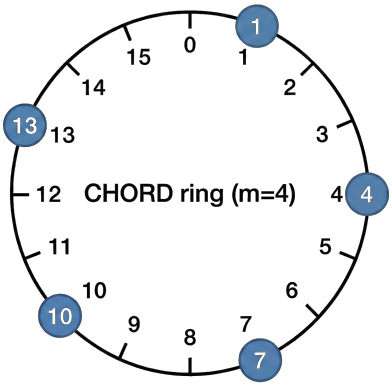
Nu veți implementa un sistem complet tolerant la erori sau dinamic, ci un scenariu static: nodurile există de la început, nu apar noduri noi pe parcurs și nu dispar noduri din inel. Chiar și în această variantă simplificată, veți lucra cu ideile esențiale ale CHORD: identificatori pe un inel, succesor și predecesor, finger table, rutare în $O(\log N)$, mesaje între noduri și un mecanism de terminare distribuită.

La final, programul vostru va simula un mic sistem CHORD, în care procesele MPI își construiesc structurile locale și colaborează prin mesaje pentru a rezolva cereri de tip „găsește nodul responsabil pentru cheia K”.

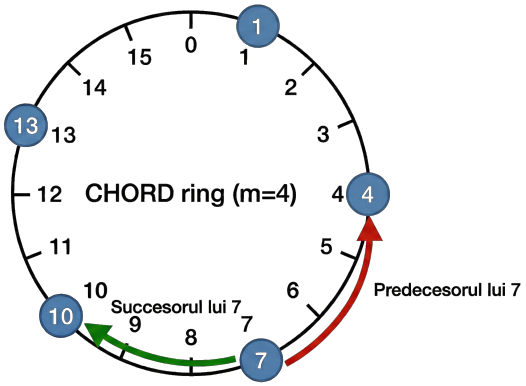
2 Protocolul CHORD

CHORD este un protocol care implementează un **Distributed Hash Table (DHT)** peer-to-peer. Un DHT stochează perechi de tip cheie-valoare prin asignarea cheilor la diferite noduri. Un nod va stoca valorile pentru toate cheile de care este responsabil. Protocolul CHORD specifică modul în care cheile sunt asignate la noduri, precum și modul în care un nod descoperă valoarea pentru o cheie dată, localizând întâi nodul responsabil de acea cheie.

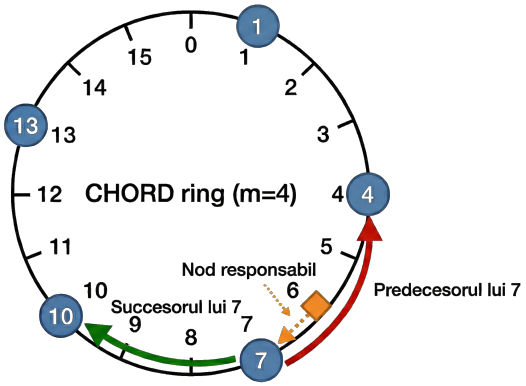
CHORD aranjează nodurile într-un cerc de identificatori. Se alege un număr m, iar toate ID-urile, atât ale nodurilor, cât și ale cheilor, sunt valori întregi între 0 și $2^m - 1$. În această temă, se va folosi $m = 4$, astfel încât spațiul de identificatori este $0 \dots 15$. Fiecare nod are un ID distinct în acest interval, iar nodurile sunt considerate plasate pe circumferința unui cerc, în ordine crescătoare. Imaginea de mai jos arată un inel CHORD cu $m = 4$, pe care există nodurile cu ID-urile 1, 4, 7, 10 și 13.



Pentru orice nod n, succesorul lui n este primul nod, în sens crescător, care are ID mai mare decât al lui n. Dacă nu există un astfel de nod (adică n este ultimul pe cerc), succesorul este nodul cu ID-ul minim. Predecesorul lui n este nodul care îl precede în ordinea circulară. În figura de mai jos, nodul 4 este predecesorul nodului 7, iar nodul 10 este succesorul acestuia.



Regula de responsabilitate pentru chei este simplă: nodul responsabil pentru o cheie K este succesorul lui K pe inel. Dacă există un nod cu ID exact K, acela este responsabil. Dacă nu, se ia următorul ID mai mare, iar dacă cheia este mai mare decât toate ID-urile, se revine la nodul cu ID-ul minim. În imaginea de mai jos, se arată că nodul 7 este responsabil de cheia 6.



Dacă am folosi doar succesorii, o căutare a unei chei (adică un lookup) ar putea vizita toate nodurile pe cazul cel mai defavorabil, deci ar dura $O(N)$. CHORD introduce însă un **finger table** pentru a evita acest lucru.

Ce este un finger?

Un **finger** este o intrare în finger table-ul unui nod CHORD și reprezintă un „salt înainte" pe inel. Fiecare nod n păstrează un finger table de m astfel de salturi, la distanțe exponențiale 2^i.

Construcția unui finger se face în doi pași:

1. Calculăm o poziție teoretică $start_i = (n.id + 2^i) \mod 2^m$. Aceasta este o poziție pe cerc, aflată la distanța 2^i de nod.
2. Finger-ul propriu-zis este succesorul acestei poziții: $finger[i] = succ(start_i)$.

Finger table-ul unui nod nu este doar o listă arbitrară, ci o structură logaritmică. Ea permite nodului să „sară" peste regiuni mari ale inelului în mod controlat, ceea ce reduce complexitatea rutării.

Exemplu de alcătuire a unui finger table

Considerăm un inel CHORD cu $m = 4$ ($2^m = 16$) și nodurile $\{1, 3, 5, 9\}$. Vrem să construim finger table-ul pentru nodul cu ID-ul 3:

1. **Pentru** $i = 0$, $start_0 = (3 + 1) \mod 16 = 4$. Succesorul lui 4 pe inel este 5, deci $finger[0] = 5$.
2. **Pentru** $i = 1$, $start_1 = (3 + 2) \mod 16 = 5$. Succesorul lui 5 este chiar nodul 5, deci $finger[1] = 5$.
3. **Pentru** $i = 2$, $start_2 = (3 + 4) \mod 16 = 7$. Succesorul lui 7 pe inel este 9, deci $finger[2] = 9$.
4. **Pentru** $i = 3$, $start_3 = (3 + 8) \mod 16 = 11$. Succesorul lui 11 este 1 (deoarece 11 depășește 9 și se revine la minim), deci $finger[3] = 1$.

Rezultatul final este:

i	$start_i$	$finger[i]$
0	4	5
1	5	5
2	7	9
3	11	1

Așadar, finger table-ul nodului 3 este [5, 5, 9, 1].

Lookup eficient folosind finger table

În același exemplu ca mai sus, dacă nodul 3 primește un lookup pentru cheia $K = 12$, el va realiza următorii pași:

- Adaugă ID-ul său în traseu.
- Verifică dacă succesorul său (5) este responsabil de acea cheie[1]. Intervalul circular $(3, 5]$ nu conține 12, deci nu este cazul.
- În această situație, nodul apelează o funcție `closest_preceding_finger(12)`, care caută cel mai mare finger din $\{5, 5, 9, 1\}$ care este strict între 3 (ID-ul nodului) și 12 (cheia căutată).
- 1 nu este în intervalul $(3, 12)$.
- 9 și 5 sunt în interval. Cel mai mare finger valid este 9.

Lookup-ul sare direct la nodul 9, care se adaugă pe traseu și continuă rutarea. Nodul 9 analizează cheia 12:

- Succesorul lui 9 este 1 (intervalul circular $(9, 1]$ conține valorile 10,11,12,...,15 și 0 și 1).
- Cum 12 se află în acest interval, succesorul lui 9, adică nodul 1, este nodul responsabil pentru cheia 12.
- Nodul 9 trimite un răspuns înapoi inițiatorului, în prealabil adăugând ID-ul 1 în traseu.

Astfel, căutarea ajunge să parcurgă nodurile $3 \to 9 \to 1$, în loc să parcurgă $3 \to 5 \to 9 \to 1$ (lucru care s-ar fi întâmplat dacă nu s-ar fi folosit finger table). Această idee duce la rutare în $O(\log N)$ în loc de $O(N)$.

Pentru ca rutarea să fie corectă și pentru a evita blocajele în cazul inelelor mici, o implementare reală a funcției `closest_preceding_finger` trebuie să respecte două reguli esențiale:

1. finger-ul ales trebuie să fie *strict* în intervalul circular $(n.id, K)$
2. finger-ele care nu produc progres (de exemplu, indică spre $n.id$ sau spre K direct) trebuie ignorate.

Dacă aceste condiții nu sunt respectate, lookup-ul poate intra în cicluri de forma $3 \to 5 \to 3 \to 5 \to \ldots$. Prin aplicarea regulilor corecte, rutarea devine progresivă și sigură, exact cum prevede protocolul CHORD.

[1] În mod normal, ar trebui ca întâi un nod să verifice dacă el însuși este responsabil de o cheie, dar, în cadrul acestei teme, nu se va lua în considerare acest caz.

3 Comparație CHORD real cu varianta implementată în temă

Pentru a înțelege mai bine scopul temei și limitările implementării cerute, este important să distingem între protocolul CHORD în forma sa completă și varianta simplificată, statică, pe care o veți implementa în temă.

CHORD real

În varianta reală, CHORD este un sistem distribuit dinamic, conceput pentru a funcționa în medii larg distribuite, cu mii sau milioane de noduri, care pot intra și ieși din sistem în orice moment. Fiecare nod cunoaște doar informație locală limitată: succesorul său, predecesorul său, și câteva intrări din finger table. Nodul nu are o vedere globală asupra întregului sistem.

Când un nod nou se alătură sistemului (operația *join*), acesta contactează un nod deja existent. Printr-o succesiune de mesaje de tip lookup, noul nod își determină succesorul, iar apoi își construiește treptat finger table-ul. În același timp, și celelalte noduri din sistem trebuie să își ajusteze structurile interne pentru a-l integra pe noul venit.

Pentru a menține corectitudinea structurii în timp, CHORD real rulează periodic proceduri de *stabilizare*. Acestea verifică dacă succesorii și predecesorii sunt corecți, repară intrări incorecte din finger table și permit sistemului să se auto-corecteze în cazul în care noduri pică, se deconectează sau reapar. Toate aceste operații se realizează exclusiv prin mesaje între noduri, fără un coordonator central.

Astfel, în CHORD real:

- inelul este dinamic
- finger table-ul se construiește și se repară prin mesaje
- succesorii și predecesorii se pot schimba în timp
- sistemul este tolerant la căderi și reconectări.

Varianta din temă

În temă, nu se urmărește implementarea unui CHORD complet din punct de vedere al dinamismului și al toleranței la erori. În schimb, se va implementa o variantă **statică** și controlată, care izolează exact partea de rutare distribuită a protocolului.

Concret, în această temă:

- toate nodurile pornesc simultan, ca procese MPI
- fiecare nod își cunoaște ID-ul de la început, dintr-un fișier de intrare
- toate ID-urile sunt colectate la fiecare nod prin `MPI_Allgather`
- inelul CHORD (succesorii și predecesorii) se construiește local, o singură dată
- finger table-ul se calculează o singură dată, la început, folosind formula teoretică și lista sortată de ID-uri
- după inițializare, structura nu se mai modifică.

Nu există operații de *join* sau *leave*, nu există proceduri de stabilizare, nu există căderi de noduri. Inelul este fix pe toată durata rulării programului. Această simplificare este intenționată și permite concentrarea exclusiv pe:

- implementarea finger table-ului

- mecanismul de rutare prin `closest_preceding_finger`
- comunicarea distribuită prin mesaje MPI
- gestionarea terminării corecte a execuției.

Construirea finger table

O diferență tehnică majoră între varianta reală și cea din temă este modul în care se determină succesorii pozițiilor $start_i$.

În CHORD real, succesorul fiecărei poziții se determină printr-un lookup distribuit, deoarece nodul nu cunoaște toate ID-urile din sistem. Finger table-ul este, astfel, rezultatul colaborării dintre noduri.

În această temă, fiecare nod cunoaște **toate ID-urile** încă de la început. Din acest motiv, succesorul unei poziții $start_i = (n.id + 2^i) \mod 2^m$ se caută direct în lista sortată a nodurilor existente, folosind funcția `find_successor_simple`.

Această regulă este importantă mai ales pentru inele mici, unde folosirea spațiului complet $0 \dots 2^m - 1$ ar putea genera succesorii greșiți și ar duce la cicluri de rutare.

Concluzii

Deși varianta este statică, logica de bază a rutării rămâne aceeași ca în CHORD real:

- regula de responsabilitate pentru chei este aceeași
- utilizarea finger table-ului pentru salturi exponențiale este identică
- funcția `closest_preceding_finger` aplică aceeași idee de progres controlat
- complexitatea teoretică a rutării rămâne $O(\log N)$.

Astfel, deși nu implementați un sistem complet din punct de vedere al toleranței la erori, implementați exact mecanismul central care face CHORD eficient: rutarea distribuită logaritmică.

4 Fișiere de intrare și scenariu de execuție

Tema se va rula utilizând `mpirun`, cu un număr de procese MPI egal cu numărul de noduri din inelul CHORD. Fiecare proces MPI reprezintă exact un nod din sistemul distribuit. Legătura dintre un proces MPI și nodul CHORD corespunzător se face prin **fișierul de intrare asociat rank-ului MPI**.

Pentru un proces MPI cu rank-ul R, fișierul de intrare va avea numele `inR.txt`. Astfel, procesul cu rank 0 va citi fișierul `in0.txt`, procesul cu rank 1 va citi `in1.txt`, etc. Fiecare fișier descrie local doar comportamentul nodului corespunzător acelui proces: ce ID are în inel și ce lookup-uri trebuie să inițieze.

Structura fișierului de intrare

Fișierul `inR.txt` are următoarea structură generală:

```
<node_id>
<num_lookups>
<key_1>
<key_2>
...
<key_L>
```

Semnificația fiecărei linii este următoarea:

- **Prima linie** conține ID-ul nodului CHORD, un întreg între 0 și $2^m - 1$. Acest ID stabilește poziția nodului în inel.
- **A doua linie** conține un număr întreg L, care reprezintă numărul de lookup-uri pe care acest nod trebuie să le inițieze.
- **Următoarele L linii** conțin cheile pentru care nodul trebuie să pornească operații de tip lookup distribuit.

Exemplu de fișier de intrare

Un fișier `in0.txt` poate avea forma:

```
8
2
7
10
```

Acest fișier descrie următorul scenariu:

- nodul corespunzător procesului cu rank 0 are ID-ul CHORD 8
- nodul trebuie să inițieze două operații de lookup
- prima căutare este pentru cheia 7
- a doua căutare este pentru cheia 10.

Scenariul complet de execuție

Scenariul de lucru al programului este următorul:

1. Toate procesele MPI pornesc simultan.
2. Fiecare proces își citește fișierul `inR.txt` și își extrage:
   - propriul ID CHORD
   - numărul de lookup-uri de inițiat
   - lista de chei pentru care trebuie lansate căutări.
3. Printr-un apel colectiv `MPI_Allgather`, fiecare proces află ID-urile tuturor nodurilor din sistem. Astfel, fiecare nod dispune de o vedere globală asupra inelului.
4. Pe baza acestei liste globale de ID-uri:
   - se construiește inelul CHORD static
   - se determină succesorul și predecesorul fiecărui nod
   - se calculează finger table-ul, folosind `build_finger_table()`.
5. După finalizarea inițializării, fiecare nod își inițiază propriile lookup-uri locale. Pentru fiecare cheie citită din fișier:
   - se construiește un mesaj de tip `LookupMsg`
   - câmpul `initiator_id` este setat la ID-ul propriu;
   - câmpul `current_id` este setat tot la ID-ul propriu
   - câmpul `key` este setat la cheia căutată

- `path_len` este inițializat cu 0
- mesajul este trimis, folosind `MPI_Send`, către propriul rank MPI, cu `TAG_LOOKUP_REQ`.

6. După inițierea lookup-urilor, fiecare nod intră în bucla de serviciu (*service loop*), în care:
   - primește cereri de tip `TAG_LOOKUP_REQ` de la alte noduri
   - primește răspunsuri finale de tip `TAG_LOOKUP_REP`
   - primește notificări de terminare de tip `TAG_DONE`.
7. Nodul continuă să proceseze mesaje până când:
   - toate lookup-urile inițiate local au primit răspuns
   - a primit mesaje `TAG_DONE` de la toate celelalte noduri.
8. Nodul afișează traseele parcurse pentru fiecare lookup inițiat.

Observații importante

- Numărul de procese MPI lansate trebuie să corespundă exact numărului de fișiere `inR.txt`.
- ID-urile nodurilor trebuie să fie distincte în cadrul aceleiași rulări.
- Cheile pot fi orice valori din intervalul $0 \dots 2^m - 1$, indiferent dacă există sau nu un nod cu acel ID.
- Ordinea în care sunt inițiate lookup-urile de la noduri diferite poate fi intercalată arbitrar, în funcție de planificarea MPI, dar rezultatele afișate trebuie să fie corecte din punct de vedere al traseului.
- Un proces poate avea de făcut 0 lookup-uri. În acest caz, el poate să anunțe celelalte noduri că a terminat lookup-urile, dar trebuie să rămână activ pentru a putea răspunde la rândul său la lookup-uri.

Această structură permite testarea clară a mecanismului de rutare distribuită CHORD și a cooperării dintre procesele MPI în absența unui coordonator central.

5 Comportament așteptat și output

După finalizarea fazei de inițializare (citirea inputului, construirea inelului CHORD și calcularea finger table-ului), programul intră în faza de rutare distribuită a cererilor de tip lookup. În această etapă, nodurile cooperează exclusiv prin mesaje MPI pentru a ajunge la nodul responsabil pentru fiecare cheie.

Fiecare operație de lookup este inițiată de un nod și este propagată prin inel conform regulilor protocolului CHORD. La fiecare pas al rutării, nodul care primește cererea:

- adaugă propriul său ID în traseul cererii
- verifică dacă succesorul său este nodul responsabil pentru cheia căutată
- dacă succesorul nu este responsabil de acea cheie, redirecționează cererea către nodul determinat de către funcția `closest_preceding_finger`.

Rutarea continuă până când se află nodul a cărui responsabilitate include cheia K. Nodul al cărui succesor este responsabil de cheie nu afișează nimic, ci trimite un mesaj de răspuns către nodul inițiator, conținând traseul complet parcurs de cerere și ID-ul nodului responsabil.

Afișarea rezultatelor

Doar nodul inițiator al unui lookup are dreptul să afișeze rezultatul. Un nod care doar intermediază o cerere de tip lookup nu trebuie să afișeze nimic legat de acea cerere.

În momentul în care nodul inițiator primește mesajul de răspuns final, el va afișa o singură linie pentru acel lookup, în formatul:

```
Lookup K: n0 -> n1 -> n2 -> ... -> target
```

unde:

- K este cheia căutată
- $n0$ este ID-ul nodului inițiator
- $n1, n2, \ldots$ sunt ID-urile nodurilor intermediare prin care a fost redirecționată cererea
- *target* este nodul responsabil final pentru cheia K.

Ordinea nodurilor afișate trebuie să reflecte **exact traseul logic parcurs de mesaj**, în ordinea în care nodurile au procesat cererea.

Reguli stricte pentru output

Pentru ca testarea automată să fie corectă, trebuie respectate următoarele reguli:

- fiecare lookup inițiat generează **exact o linie de output**
- ordinea liniilor pentru lookup-urile inițiate de același nod trebuie să fie aceeași cu ordinea cheilor din fișierul de intrare
- formatul general `Lookup K: ...` trebuie respectat
- separarea între noduri se face prin simbolul `->`
- nu se afișează mesaje auxiliare sau de debug.

Corectitudinea unui lookup

Un lookup este considerat **corect** dacă:

- ultimul nod din traseu este succesorul corect al cheii K
- traseul nu conține cicluri
- fiecare salt între două noduri consecutive din traseu este justificat fie prin regula succesorului, fie prin regula `closest_preceding_finger`.

O implementare care ajunge corect la nodul responsabil, dar parcurge aproape toate nodurile prin succesor (fără a folosi eficient finger table-ul), este considerată funcțională, dar va primi un punctaj redus.

Observație privind intercalarea mesajelor

Lookup-urile inițiate din noduri diferite pot fi procesate intercalat, în funcție de planificarea execuției MPI. Atfel:

- ordinea globală a liniilor de output nu este garantată
- testarea automată se va face prin asocierea fiecărei linii cu cheia K corespunzătoare
- nu se impune o sincronizare globală pentru afișare.

Singura constrângere este că fiecare linie de output trebuie să descrie corect traseul unui lookup finalizat.

Exemplu

Pentru un lookup inițiat la nodul 3 pentru cheia 12, o afișare validă este:

```
Lookup 12: 3 -> 9 -> 1
```

Aceasta indică faptul că cererea a fost procesată de nodurile 3 și 9, iar nodul responsabil final pentru cheia 12 este nodul 1.

6 Implementare

Fișierul `tema2.c` este oferit sub forma unui **schelet de cod** în directorul **src** din repository-ul temei, care conține infrastructura generală a aplicației distribuite, împreună cu o parte din logica deja implementată. Rolul vostru este să completați acest schelet cu implementarea efectivă a mecanismelor specifice protocolului CHORD.

Pentru a fi clar, mai jos se specifică în detaliu **ce este deja implementat în schelet** și **ce trebuie implementat de voi**.

Ce este deja implementat în schelet

Următoarele componente sunt deja implementate și funcționale în fișierul `tema2.c`:

- definirea constantelor (`M`, `RING_SIZE`, `MAX_NODES`, etc.)
- definirea structurilor de date:
  - structura `NodeState` (care conține ID-ul nodului, succesorul, predecesorul și finger table-ul)
  - structura `Finger`
  - structura `LookupMsg`, utilizată pentru cererile și răspunsurile de tip lookup
- definirea tag-urilor MPI:
  - `TAG_LOOKUP_REQ` pentru cereri
  - `TAG_LOOKUP_REP` pentru răspunsuri
  - `TAG_DONE` pentru terminare
- citirea fișierelor de intrare `inR.txt`, în funcție de rank-ul MPI
- apelul colectiv `MPI_Allgather` pentru colectarea ID-urilor tuturor nodurilor
- construirea inelului CHORD static:
  - sortarea ID-urilor
  - determinarea succesorului și predecesorului fiecărui nod
- funcțiile utilitare pentru:
  - lucrul cu intervale circulare
  - determinarea succesorului unei valori folosind lista sortată de noduri (`find_successor_simple`)
  - maparea dintre ID-urile CHORD și rank-urile MPI
- inițializarea mediului MPI (`MPI_Init`) și închiderea acestuia (`MPI_Finalize`).

Toate aceste componente oferă suportul necesar astfel încât voi să vă concentrați exclusiv pe logica protocolului CHORD și pe rutarea distribuită propriu-zisă.

Ce trebuie să implementați voi în schelet

Următoarele părți din schelet sunt marcate ca **TODO** și trebuie implementate integral de către voi:

1. **Funcția `build_finger_table()`**. Această funcție trebuie să construiască finger table-ul nodului curent, format din câmpurile *start* și $finger$, descrise în secțiunea 2.
2. **Funcția `closest_preceding_finger(K)`**. Această funcție primește o cheie K și trebuie să caute în finger table, de la cea mai mare poziție la cea mai mică, un nod care se află strict în intervalul circular $(self.id, K)$. Dacă un astfel de nod este găsit, funcția îl returnează. Dacă nu, se revine la succesorul direct al nodului curent. Această funcție este esențială pentru ca rutarea să fie logaritmică și pentru a evita ciclurile de rutare.
3. **Funcția `handle_lookup_request()`**. Aceasta este funcția centrală a rutării distribuite. La primirea unei cereri de tip lookup, nodul curent trebuie:
   - să își adauge propriul ID în traseul mesajului
   - să verifice dacă succesorul său este nodul responsabil pentru cheia căutată
   - dacă da, să adauge succesorul în traseu și să trimită un mesaj `TAG_LOOKUP_REP` către nodul inițiator
   - dacă nu, să determine următorul nod folosind `closest_preceding_finger` și să trimită cererea mai departe cu `TAG_LOOKUP_REQ`.
4. **Inițierea lookup-urilor în `main`**. Pentru fiecare cheie citită din fișierul de intrare, trebuie să creați un mesaj de tip `LookupMsg`, să inițializați corect câmpurile acestuia și să îl trimiteți către propriul rank MPI cu tag-ul `TAG_LOOKUP_REQ`.
5. **Bucla principală de procesare a mesajelor (service loop)**. Scheletul nu conține bucla completă de procesare a mesajelor. Aceasta trebuie implementată de voi și trebuie să gestioneze cererile de tip `TAG_LOOKUP_REQ`, răspunsurile finale `TAG_LOOKUP_REP`, semnalele de terminare `TAG_DONE`. Nodul trebuie să continue să proceseze mesaje până când toate lookup-urile inițiate local au fost rezolvate și a primit semnalul `TAG_DONE` de la toate celelalte noduri. La momentul în care un nod își termină toate lookup-urile locale (adică a aflat toate traseele dorite), acesta trebuie să trimită un mesaj `TAG_DONE` către toate celelalte noduri o singură dată.

Atenție! Deși scheletul oferă infrastructura de bază, **simplificarea rutării la o parcurgere liniară prin succesor va fi penalizată**. Chiar dacă implementările care ignoră finger table-ul și returnează `closest_preceding_finger` vor funcționa corect din punct de vedere al rezultatului final, asupra lor se vor aplica penalizările descrise în regulile de punctaj.

7 Rulare, testare și evaluare

Compilare

Tema se implementează în limbajul C/C++ și se compilează folosind `mpicc/mpic++`. În arhiva încărcată pe Moodle, trebuie să existe un fișier `Makefile` cu o regulă *build* care să genereze un binar cu numele `tema2`.

Rulare

Programul se rulează folosind `mpirun`, cu un număr de procese MPI egal cu numărul de fișiere de intrare.

Testele automate și checkerul

Corectitudinea soluției va fi verificată folosind un set de teste automate aflate în repository-ul temei, împreună cu un script Bash (numit *local.sh*) pe care îl puteți rula pentru a vă verifica implementarea. Rularea locală în acest mod necesită existența Docker pe sistemul vostru. Script-ul din repository va fi folosit și pentru testarea automată pe VMChecker, în fix aceleași condiții[2]. Scriptul de testare locală se rulează astfel:

```
$ ./local.sh checker
```

Fiecare test conține un set de fișiere `inR.txt` și un fișier de ieșire de referință. În toate testele pe care le vom rula, există cel puțin două noduri. Checkerul verifică următoarele lucruri:

- corectitudinea nodului responsabil pentru fiecare cheie
- corectitudinea traseului parcurs de fiecare lookup
- absența ciclurilor în rutare
- respectarea formatului de output
- faptul că programul nu se blochează (timeout)
- calitatea rutării (număr de hop-uri).

Evaluarea eficienței (log N vs O(N))

Testele sunt construite astfel încât să permită atât implementări corecte, logaritmice (folosind finger table-ul), cât și implementări corecte, dar liniare (folosind doar succesorul). Ambele variante pot produce rezultatul corect, însă vor fi punctate diferit. Checkerul analizează lungimea traseelor și penalizează soluțiile care se apropie de complexitatea $O(N)$.

Punctaj

Punctajul este divizat după cum urmează:

- **20p** – claritatea codului și a explicațiilor din `README`
- **40p** – corectitudinea rutării și identificarea nodului responsabil pentru fiecare cheie
- **40p** – eficiența rutării (utilizarea corectă a finger table-ului, rutare apropiată de $O(\log N)$).

Penalizări

- **-100p** – cicluri de rutare sau blocaje (tema considerată nefuncțională)
- **-100p** – netransmiterea propriu-zisă de mesaje MPI pe traseul către nodul responsabil de cheie
- **-100p** – comunicarea directă dinspre nodul care face lookup către nodul responsabil de o cheie
- **-20p** – lipsa terminării corecte.

[2]Nota obținută în urma rulării automate poate fi scăzută pe baza elementelor de depunctare descrise mai jos.

Predare pe Moodle

Tema se va încărca aici pe Moodle **sub formă de arhivă ZIP**. Arhiva trebuie să conțină în rădăcină următoarele fișiere:

- **`tema2.c`** – fișierul sursă completat (și eventual alte fișiere sursă/header, dacă este cazul)
- **`Makefile`** – fișierul de compilare care generează binarul `tema2` cu o regulă *build*
- **`README.pdf`** – un scurt document PDF (1–2 pagini) care să conțină numele studentului, explicația soluției alese, eventuale observații privind implementarea.

Tema se predă individual. Orice tentativă de plagiat va fi sancționată conform regulamentului.

8 Bibliografie

1. I. Stoica et al., *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM Computer Communications Review, 2001.
2. D. Karger et al., *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC), 1997.
3. Documentația oficială MPI, *MPI: A Message-Passing Interface Standard*, https://www.mpi-forum.org
4. Laboratorul 8 Algoritmi Paraleli și Distribuiți, *Introducere în programarea distribuită cu MPI*, https://mobylab.docs.crescdi.pub.ro/docs/parallelAndDistributed/laboratory8/.