Chordy Documentatie

Morari Maxim

Facultatea de Informatica, Universitatea Alexandru Ioan Cuza, Iasi, Romania

1 Introducere

Aplicatia Chordy este o implementare in C/C++ Linux a protocolului/agloritmului Chord, ce permite locarea eficienta a colegilor intr-o retea peer-to-peer, care implementeaza, in cazul dat, o tabele hash distribuita.

Cuvantul cheie este "eficient". Intr-o retea peer-to-peer, problema principala este localizarea severului ce (poate) contine informatia ceruta de un client. Daca fiecare nod ar memora majoriatatea nodurilor din retea, s-ar produce o supraincarcare de date, iar daca ar memora, sa presupunem, un nod, succesoru sau, cautarea ar avea complexitatea timp O(N), N fiind numarul de noduri din retea. Chord ofera o complexitate O(log(N)) atat in timp cat si in spatiu.

2 Tehnologii Aplicate

Comunicarea dintre server-client, server-server se bazeaza pe TCP. Este necesar ca mesajele sa fie schimbate corect si sigur, mai ales pentru comunicarea dintre servere. Mesajele dintre ele fac parte din algoritmul de stabilizare a retelei, si de gasire a nodului cu informatia ceruta de client.

Socketuri din libraria **sys/socket.h** sunt folosite pentru stabilirea comunicarii intre noduri din retea.

Pentru concurenta voi folosi libraria **thread** specifica C++, deoarece ea ne permite sa transmitem metode non-statice ca parametru la crearea unui thread. E important sa folosesc threaduri, deoarece am nevoie ca ele sa proceseze in comun datele unui nod Chord. Si, in general, sistemul menajeaza mult mai eficient k threaduri decat k procese separate.

Pentru a stoca perechile (cheie,valoare) ale tabelei hash, folosesc libraria **nlohmann/json**.

Pentru acces exclusiv la fisierele json asociate unui server folosesc **mutex** din C++.

3 Structura Aplicatiei

Fiecare Chord nod are aceeasi structura logica: id, adresa, fingerTable, succesor, si o unitate de stocare de date. Am evidentiat acestea complet doar pentru un nod (la care se conecteaza clientul). Topologia retelei este un inel. Numarul maxim de noduri este 2^m, fiecare nod are un id din acest diapazon, m este predefinit. $Succesor(key)$ este primul nod in starea curenta a retelei, cu $id >= key$, acest nod este corespunzator de toate cheile din intervalul $(predecesor.id, this.id]$.

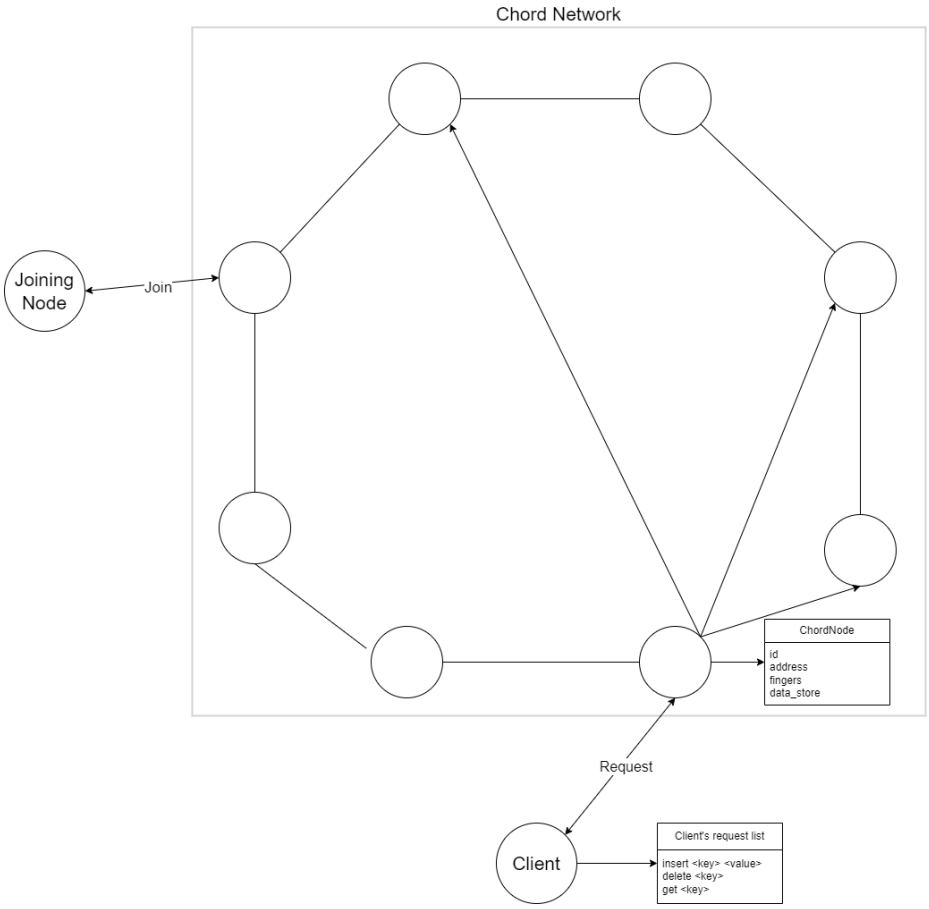

Fig. 1. Structura generala a aplicatiei

Un client se poate conecta la orice nod din retea. Daca informatia ceruta se gaseste intru-un din retea, el o va primi. Daca un nou Chord serve doreste sa se conecteze la o retea existenta, el trebuie se apeleze join pe un oricare nod din retea. In continuare prezint digarame pentru componente separate ale aplicatiei,

3.1 Gasirea unui nod in retea

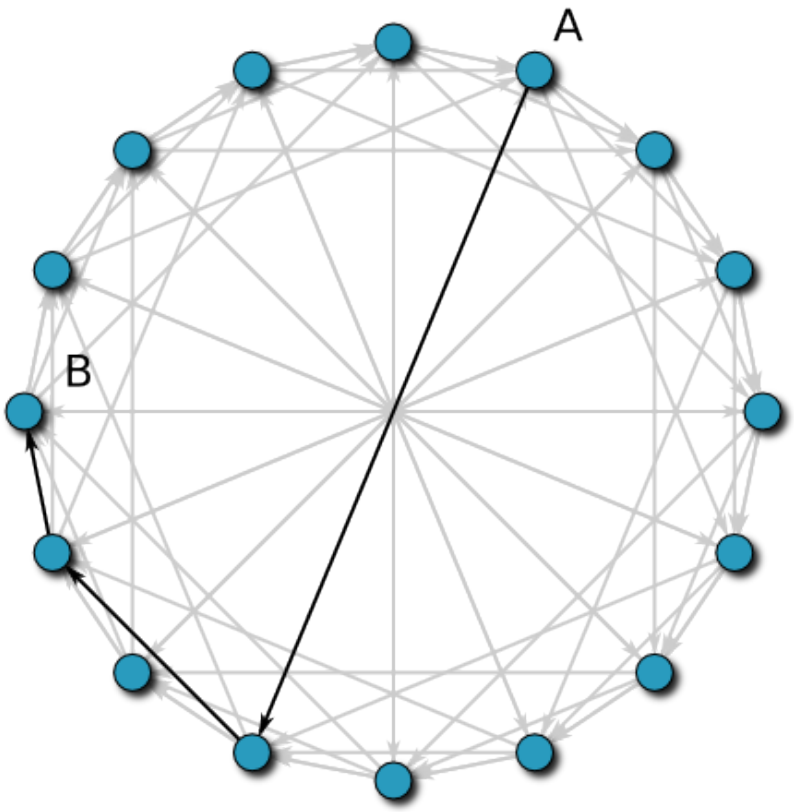

Fig. 2. Nodul A cauta succesorul(key), ce in cazul dat e B

Tabela fingerTable contine m intrari, fiecarei intrari ii corespunde $succesor(fingerTable[i].start)$, $fingerTable[i].start = (n + 2^i)\%2^m, 0 \leq i < m$, n este id'ul nodului curent (pozitia lui in inel). Acesti pointeri ne ajuta sa gasim $succesor(key)$ in $O(log(N))$ in felul urmator: nodul curent cauta cel mai apropiat nod cu $id \leq key$, si ii cere lui sa gaseasca $succesor(key)$. Acest proces se gateste cand $this.id \leq key \leq successor.id$, caz in care $succesor(key)$ este nodurl curent sau succesorul lui. Cum distanta de la nodul curent la noduri din finerTable sunt puteri ale lui 2, la fiecare pas distanta dintre nodul curent si nodul cautat se micsoreaza de aproximativ 2 ori.

3.2 Servirea unei cereri din partea clientului

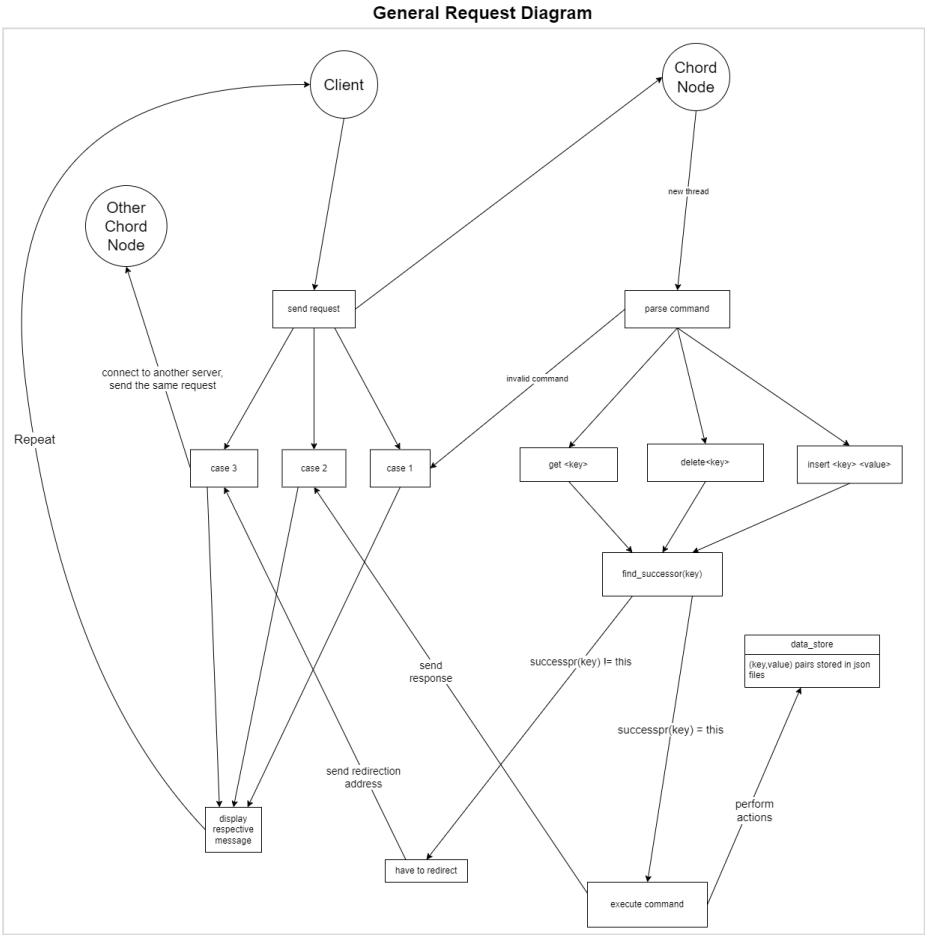

Fig. 3. Diagrama unei cereri

Cand un server primeste o cerere de la client, el calculeaza un hash pentru parametru key. Pe baza acestui hash, el cauta nodul din retea curent respunzator pentru key, $find\_successor(key)$. Daca el insasi e raspunzator, serveste cererea loca, altfel trimite clientului adresa serverului raspunzator pentru key, clientul se conecteaza la el si retrimite aceeasi cerere. De acum cererile sunt trimise serverului din urma, pana la o eventuala redirectionare la alt server, in caz ca serverul curent nu e responsabil de cereri pentru un anumit key.

3.3 Conectarea unui server la o retea existenta

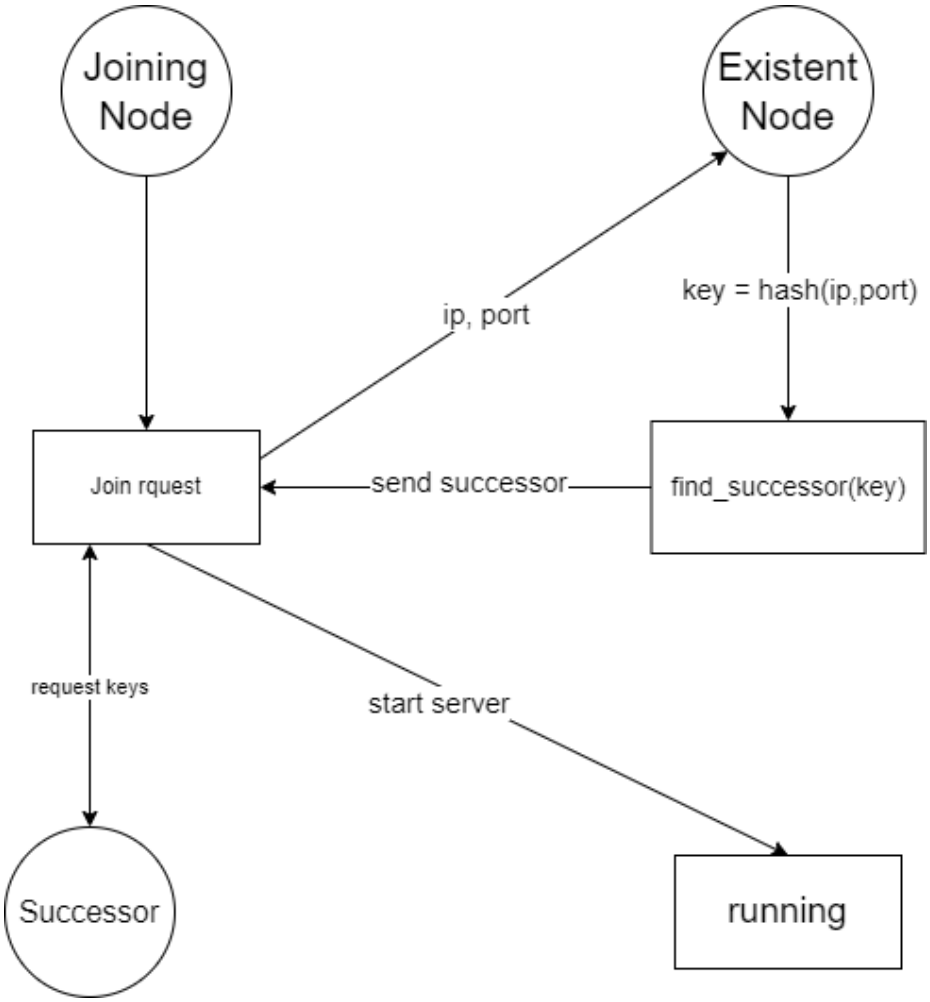

Fig. 4. Diagrama procedurii join

Cand un nou nod doreste sa se conecteze la o retea existenta, el apeleza join unui nod din retea. Acesta ii comuninca succesorul, nodul nou isi seteaza succesorul (astfel devine parte din retea), si cere succesorului chei pentru care nodul nou poate fi raspunzator in noua stare(succesorul a luat raspundere pentru chei inainte conectarii nodului nou, ce acum ar trebui trimise lui).

3.4 Stabilizarea retelei

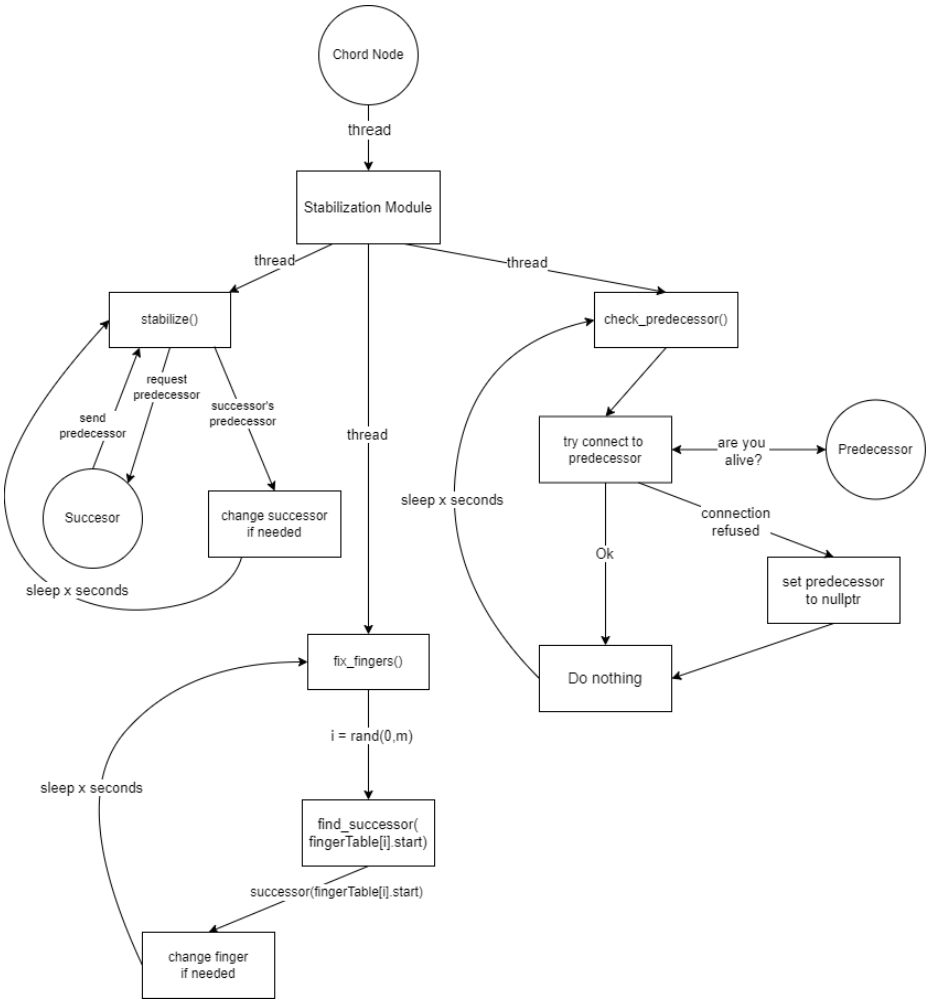

Fig. 5. Diagrama Modulului de Stabilizare

Cand un nod nou se conecteaza la retea, tabelele fingerTable ale unor noduri pot fi in spate, ele trebuie actualizate, modulului de stabilizare ii este repartizata aceasta sarcina. Functia stabilize() verifica daca predecesorul succesorului ar trebuie sa devina noul succesor al nodului curent(caz in care un nod recent nou a devenit predecesorul succesorului nodului curent). Functia fix_fingers() ia rand pe rand intrari din fingerTable si incearca sa le reactualizeze, folosing find_successor(). Functia check_predecessor() verifica daca predecesorul a esuat, caz in care informatia corespunzatoare este invechita.

4 Aspecte de Implementare

4.1 Clientul

In urma unei proceduri standard de conectare la server, si de citire a inputul de la claviatura, clientul trimite cererea la server.

```
if (write(sd, &reqtype, 1) <= 0)
    HANDLE_EXIT("error when sending request to server.\n");
if (write(sd, &request_len, 4) <= 0)
    HANDLE_EXIT("error when sending request to server.\n");
if (write(sd, request, request_len) <= 0)
    HANDLE_EXIT("error when sending request to server.\n");
```

Apoi, el citeste tipul raspunuslui, si se comporta respectiv.

```
if (read(sd, &type, 1) < 0)
    HANDLE_EXIT("error when reading response type from server.\n");
if (type == responseType::REDIRECT)
{
    u32 redirect_ip, redirect_port;
    if (read(sd, &redirect_ip, 4) < 0)
        HANDLE_EXIT("error when reading redirect server address .\n");
    if (read(sd, &redirect_port, 4) < 0)
        HANDLE_EXIT("error when reading redirect server address .\n");
    close(sd);
    sd = set_connection_to(redirect_ip, redirect_port);
}
else if (type == responseType::OK)
    display_response(sd);
else if (type == responseType::UNRECOGNIZED)
    printf("Unrecognized command\n");
```

```c
int display_response(int sd)
{
    int response_length;
    char response[RESPONSE_MAXLEN];
    if (read(sd, &response_length, 4) < 0)
        HANDLE_EXIT("error when reading response length from server.\n");
    int br;
    if ((br = read(sd, response, response_length)) < 0)
        HANDLE_EXIT("error when reading response body from server.\n");
    response[br] = 0;
    printf("%s\n", response);
    fflush(stdout);
    return 0;
}
int set_connection_to(u32 ip, u32 port)
{
    struct sockaddr_in server;
    int sd;
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = ip;
    server.sin_port = port;
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        HANDLE_EXIT("error when calling socket().\n");
    if (connect(sd, (struct sockaddr *)&server, sizeof(struct sockaddr)) == -1)
        HANDLE_EXIT("error when calling connect()\n");
    return sd;
}
```

Toata aceasta procedura se repeta intr-o bucla. u32 este doar un typedef pentru unsigned int. HANDLE_EXIT este un macro pentru tratarea erorilor.

4.2 Server

```c
u32 port = atoi(argv[1]);
if (argc == 2)
{
    printf("Creating a new Chord network\n");
    ChordNode cn(ip, port);
    cout << cn.printInfo();
    cn.run();
}
else
{
    u32 other_port = atoi(argv[2]);
    printf("Joining node %d to the node %d of an existent Chord network\n", port, other_port);
    endpoint other(ip, other_port, 0);
    ChordNode cn(ip, port, &other);
    cout << cn.printInfo();
    cn.run();
}
```

Serverul poate crea o noua retea, daca e pornit cu un singur argument(portul la care va asculta cereri). Altfel, al doilea argument indica portul unui server existent. Adresa ip este cea locala.

In functia run(), serverul porneste functiile periodice de stabilizare in thread-uri separate, open_to_connection() este o procedura standard de deschidere a serverului la un port, apoi astepta cereri si le proceseaza in threaduri separate prin functia treat_node().

```
void ChordNode::run()
{
    int sd;
    start_periodic_functions();
    open_to_connection(htonl(this->address.ip), htons(this->address.port), &sd);
    struct sockaddr_in from;
    socklen_t length = sizeof(from);
    while (1)
    {
        int client = accept(sd, (struct sockaddr *)&from, &length);
        if (client < 0)
        {
            perror("Error when accepting client in main thread");
            continue;
        }
        thread thread_instance(&ChordNode::treat_node, this, client);
        thread_instance.detach();
    }
}
```

In treat_node(), serverul recunoaste tipul cererii, ce este de urmatorul tip:

```
enum requestType
{
    GET_SUCCESSOR , GET_PREDECESSOR,
    FIND_SUCCESSOR, NOTIFICATION, CLIENT_REQUEST,
    END_CONNECTION, PING, REQUEST_KEYS
};
```

Daca tipul cererii este de tipul CLIENT_REQUEST, treat_node() apeleaza try_serve_client(). In aceasta, serverul verifica daca el este raspunzator de cheia din cererea clientului.

```
hash = sha1_hash(args[1]);
succ = find_successor(hash);
if (succ->id != this->id)
{
    resp_type = responseType::REDIRECT;
    redirect_ip = succ->ip;
    redirect_port = succ->port;
    sprintf(response, "Recognized a valid request, but have to redirect %s", ep2str(succ));
    goto send_response;
}
```

Daca el este raspunzator pentru cheia data, el executa comanda respectiva, spre exemplu *get key*:

```
if (command.compare("get") == 0)
{
    auto it = data_store.find(args[1]);
    if (it != data_store.end())
    {
        sprintf(response, "Found (%s,%s)", args[1].c_str(), it->second.c_str());
    }
    else
    {
        sprintf(response, "Recognized request for fetching the value of key \"%s\", but found no such pair", args[1].c_str());
    }
}
```

Serverul cauta in *data_store* daca exista o intrare pentru valoarea *kry* (args[1]), si umple bufferul de raspuns in mod corespunzator.

La urma, serverul trimete raspuns corespunzator clientului

```
send_response:
    if (resp_type == responseType::UNRECOGNIZED)
    {
        if (write(client, &resp_type, 1) < 0)
            thread_handle_error_fn(1, "write rt < 0 to %s", sd2str(client));
    }
    else if (resp_type == responseType::OK)
    {
        thread_notify("sending response=(%s)",response);
        send_string(client, responseType::OK, response);
    }
    else if (resp_type == responseType::REDIRECT)
    {
        redirect_to(client, htonl(redirect_ip), htons(redirect_port));
        printf("Can't serve locally, redirecting client to %s:%d\n", u32_to_string(redirect_ip, HOST_BYTE_ORDER).c_str(), redirect_port);
        keep_connection = STOP_CONNECTION;
    }
    continue;
}
```

Daca serverul n-a recunoscut cererea, trimite doar FAIL, fara mesaj text. Altfel apeleaza send_string(), ce trime buffer-ul *response* clientului, si reia bucla de primire a cererilor pentru acelasi client. Iar in cazul in care nu e responsabil pentru cheia din cerere apeleaza redirect_to(), ce trimete clientului adresa serverului responsabil, si intrerupe bucla ($keep\_connection = STOP\_CONNECTION$).

Clasa data_store_ arata in felul urmator:

```
class data_store_
{
    using map = unordered_map<string, string>;
public:
    std::mutex data_store_mutex;
    std::unordered_map<std::string, std::string> keys;
    string data_store_path;
    data_store_(string address);
    void retrieve_saved_keys();
    std::string &operator[](const std::string &key);
    unique_lock<std::mutex> lock_data_store();
    map::iterator begin();
    map::iterator end();
    map::iterator find(const string &s);
    map::iterator erase(map::iterator &it);
    void insert(const string &key, const string &value);
    string path_from_key(const string &key);
};
```

4.3 Scenarii de utilizare

Protocolul Chord ofera stocarea balansata, scalabila si flexibila a datelor intr-o retea decentralizata. Chordy este o implementare demonstrativa a protocolului, dar ar putea fi dezvolata la o aplicatie reala pentru urmatoarele scenarii:

- **Partajarea si distribuirea fisierelor**
  Chord poate fi folosit in sistemele de partajare de fisiere peer-to-peer in care utilizatorii partajeaza fisiere direct intre ei, fara a se baza pe un server central. Fiecare peer poate folosi protocolul Chord pentru a localiza si a prelua eficient fisierele de la alți colegi din rețea.
- **Content Delivery Networks (CDNs)**
  CDN-urile pot folosi Chord sau DHT-uri similare pentru a gestiona și distribui continutul pe mai multe noduri. Acest lucru poate îmbunatati disponibilitatea si performanta livrarii de continut, asigurandu-se ca continutul este stocat si difuzat din noduri distribuite geografic.
- **Sisteme de baze de date distribuite**
  In bazele de date distribuite, Chord poate fi folosit pentru a localiza nodul responsabil pentru o anumită bucată de date. Acest lucru ajuta la dis-

tribuirea datelor pe mai multe noduri, asigurand in acelasi timp o recuperare eficienta

- **Decentralized Finance (DeFi) Applications**
  In aplicațiile de tip blockchain si finantare descentralizata, Chord sau DHT-uri similare pot fi utilizate pentru a gestiona stocarea descentralizata si regasirea datelor contractelor inteligente, a profilurilor de utilizator sau a altor informații relevante.
- **Retele de senzori**
  Chord poate fi aplicat in retelele de senzorii care sunt distribuiti intr-o zona geografica, iar interogarea eficienta sau recuperarea datelor este esentiala.
- etc.
  Multe alt sisteme ce au nevoie de o abordare eficienta a problemelor de decentralizare a datelor, de stocarea lor balansata recuperarea lor eficienta.

5 Concluzii

In general, am implementat protocolul Chord cu succes si destul de eficient. Raman unele componente ce pot fi mai eficiente, si unele care au ramas nerealizate. Desi am realizat o redirectionare a clientului la alt server, astfel incat raspunul nu trebuie sa fie trimis de 2 ori (o data de la server la server, apoi de la server la client, in cazul in care serverul nu e responsabil de o cheie), raspunsul de cautare a succesorului trece inapoi prin toate nodurile ce au fost parcurse, nu direct de la nodul final la cel original, s-ar putea de elaborat protocolul pentru a face mai eficient aceasta parte. Am abordat minimal, in functia $check\_predecessor()$ tratarea cazurilor cand nodurile pleaca neorganizat din retea. Cand un nod pleaca din retea, aceasta ar trebui reflectata in tabelele $fingerTable$ ale nodurilor ramase, altfel functiile de stabilizare si cautarea succesorului pot esua. Dar aceasta este destul de complicata si tehnica, aproape nedescutata in lucararea originala si pe wikipedia.

6 Bibiografie

- Original Paper
- Wikiepdia article
- Site-ul materiei