Chordy

Iulian Baltag

Faculty of Computer Science Iasi
https://www.info.uaic.ro

Abstract. Implementarea unui design DHT bazat pe Chord si structurat in doua aplicatii server si client.

Keywords: DHT · Chord · TCP.

1 Introducere

O tabela hash distribuita (DHT) functionează prin asocierea cheilor cu nodurile dintr-un sistem distribuit. Fiecare nod va stoca valorile corespunzătoare cheilor distribuite acestuia. Un exemplu bine cunoscut de design pentru DHT este chord. Acesta este un algoritm si protocol pentru o tabelă hash distribuita *peer to peer*. Chord specifică cum o cheie este atribuita unui nod și cum un nod poate descoperi o valoare găsind mai întâi nodul resposnsabil de cheia pentru acea valoare. În aplicația prezentată ulterior un client poate trimite comenzi unui server(nod) pentru adaugarea, ștergerea și găsirea unei chei.

2 Tehnologii Aplicate

- Aplicatia este implementata în limbajul de programare C.
- Am implementat un protocolul de comunicare pe baza TCP, oferind o conexiune fiabilă și orientată pe flux între server și clienți.
  - TCP asigură transmiterea mesajelor în ordine și fără erori, lucru necesar pentru funcționarea unui DHT precum Chord. Rutarea cheilor, comunicarea între noduri, analizarea succesorilor necesită transmiterea de mesaje fără erori.
  - TCP permite conexiunea permanentă a nodurilor. Acest lucru este necesar deoarece nodurile comunică frecvent pentru a menține structura rețelei. Oferă un canal bidirecțional.
- Pentru gestionarea conexiunilor multiple am folosit threaduri. Acestea asigură concurență și sunt mai eficiente ca procesele în privința resurselor și a timpului.
- Pentru stocarea perechilor *cheie-valoare* am folosit o bază de date *sqlite*. Am creat câte un tabel pentru fiecare nod.
- Pentru funcția de hash-ing am folosit librăria *openssl* utilizând SHA1.

3 Structura aplicației

Aplicația Chord este structurată în server și client. Acestea comunică între ele prin socketuri. Utilizatorii vor avea la dispoziție următoarele comenzi:

- **ADD** <*key*> <*value*> Adaugă o pereche cheie-valoare în sistemul distribuit. Perechea cheie-valoare este trimisă nodului responsabil de stocarea acesteia(dupa hashing și eventualele rutări).
- **FIND** <*key*> Găsește valoarea asociată unei chei. Nodul curent verifică dacă cheia îi aparține, dacă nu folosește mecanismul de rutare pentru a găsi nodul responsabil. Valoarea este returnată clientului.
- **REMOVE** <*key*> Ștergerea perechei cheie-valoare asociată cu cheia respectivă. Similar ca mai sus, nodul responsabil de cheie o șterge pe aceasta din memoria sa locală.

Urmatoarea diagramă ilustrează funcționalitatea proiectului:

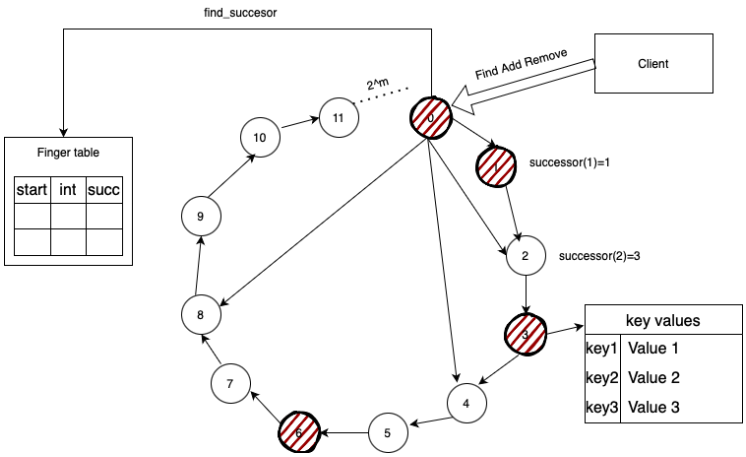

Fig. 1. Diagramă Chord

Se poate utiliza tabela *Finger table* pentru optimizarea căutărilor. Fiecare nod stochează local perechile cheie-valoare asociate acestuia precum și nodurile succesor și predecesor. Nodurile hash-urate cu roșu sunt nodurile "online" din rețea. Un astfel de nod este responsabil de toate valorile care îl preced pe inel începând de după ultimul nod online. În rețea vom avea 2^m noduri și fiecare nod va avea un ID format din primii m biți din hash.

4 Aspecte de implementare

Pentru comunicare vom folosi *socketuri*. În secțiunile de cod de mai jos este ilustrat protocolul prin care clientul comunică cu serverul urmând pașii *connect, send, recv* și inițierea structurii *sockaddr* necesare comunicării.

Inițiere:

```
int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == -1)
        perror("[client] Eroare la socket");

    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = inet_addr(ip);
```

Conectarea la server:

```
     if (connect (s, (struct sockaddr *) &server,sizeof (
         struct sockaddr)) == -1)
        perror ("[client] Eroare la connect().\n");
```

Trimiterea comenzilor:

```
     printf("Se trimite comanda %s\n", comanda);
    send(s, comanda, strlen(comanda), 0);
```

Primirea răspunsului:

```
    int bytes_received = recv(s, buffer, BUFFER_SIZE, 0);
    if (bytes_received == -1)
        perror("[client] Eroare la recv() de la server.\n");
    else
    {
        buffer[bytes_received] = '\0';
        printf("Raspuns primit de la server: %s\n", buffer);
    }
```

Acestea sunt urmate de instrucțiunile *bind, listen, accept* din server. În cadrul serverului ne folosim de threaduri prin intermediul bibliotecii *pthread.h.* Astfel se va crea un nou thread pentru fiecare comandă dată de useri.

```
    pthread_t thread;
        if(pthread_create(&thread, NULL, handle_request, (
            void*)new_socket) != 0) {
            perror("[server] Thread creation failed");
            close(client_socket);
            free(new_socket);
            continue;
        }
        pthread_detach(thread);
```

Fiecare comandă primită din server va fi parsată de către handler **ȘI FORWARDATĂ DACĂ ESTE CAZUL**(la succesorul imediat sau utilizând finger table), comenzile transmise de clienții din afara rețelei vor fi *FIND, ADD , REMOVE.* Vom gestiona secțiunile de cod critice cu ajutorul mutexurilor din biblioteca *pthread.h.*

```
    if(strncmp(buffer, "FIND␣", 5) == 0) {
        //implemenatre FIND
         pthread_mutex_lock(&mutex_chord);
         /* sectiune critica */
         pthread_mutex_unlock(&mutex_chord);
    }
    else if(strncmp(buffer, "ADD␣", 4) == 0) {
        //implementare ADD
         pthread_mutex_lock(&mutex_chord);
         /* sectiune critica */
         pthread_mutex_unlock(&mutex_chord);
    }
    else if(strncmp(buffer, "REMOVE␣", 7) == 0) {
        //implementare REMOVE
         pthread_mutex_lock(&mutex_chord);
         /* sectiune critica */
    }

    .
    . etc
    .

    else {
        snprintf(raspuns, BUFFER_SIZE-1, "Please␣use:␣(␣ADD␣<
           key>␣<value>␣;␣FIND␣<key>␣;␣REMOVE␣<key>␣;␣exit␣)
           :␣");
    }
```

Alte comenzi utilizate pentru menținerea rețelei vor fi *UPDATE_P* , *UPDATE_S* , *REQUEST_KEYS* , *TRANSFER_KEY* , *PING* etc.

Nodurile vor fi reprezentate printr-o structură. Nodul inițial va avea succesorul și predecesorul el însuși urmând să fie actualizate la reuniunea cu alte noduri.

```
typedef struct {
    int id;                 // ID nod (hash-ul portului)
    int port;               // Port nod
    int successor;          // Port succ
    int predecessor;        // Port pred
    int finger_table[M];
    int start[M];
    int backup_successor;   // Al doilea succesor (succesorul
         succesorului)

}Node;

Node node;
```

Criptarea cheiler se va face folosind SHA1 din librăria *openssl*.

```c
int hash_sha1(int key) {
    //Convertim key in string
    char string_cheie[64];
    snprintf(string_cheie, sizeof(string_cheie), "%d", key);

    unsigned char digest[SHA_DIGEST_LENGTH]; //
        SHA_DIGEST_LENGTH = 20

    SHA1((unsigned char*)string_cheie, strlen(string_cheie),
        digest);

    //Interpretam primii 4 bytes din SHA1 ca un numar (32 de
        biti)
    uint32_t hash_val = 0;
    for (int i = 0; i < 4; i++) {
        hash_val = (hash_val << 8) | digest[i];
    }

    //Facem modulo 2^m = RING_SIZE
    return (hash_val % 256);
}
```

Pentru funcționalitatea proiectului am implementat urmatoarele funcții:

```c
void send_join_message(int port_conectare, int port);
void send_update_message(int port, const char *message);
int send_message(int port, const char *message, char*
    response);
void show_node();
int send_find_successor(int target_port, int id);
int closest_preceding_node(int id);
int find_successor(int id);
void update_finger_table();
void show_finger_table();
void start_table();
void *stabilizer();
int preceding_finger(int id);
void transfer_keys_to_new_node(int new_node_port, int
    new_node_pred);
int check_node_alive(int port);
void is_succesor_alive();
```

Inițializarea bazei de date se face in modul urmator:
Ne folosim de o tabelă pentru fiecare nod din rețea

```c
void init_database(int id_nod) {
    char sql[256];
    char *err_msg = NULL;

    if (sqlite3_open("bd_centralizata.db", &db)) {
```

```
        perror("Eroare la deschiderea bazei de date");
        exit(1);
    }
    printf("Succesfully opened 'bd_centralizata.db'.\n");

    //Setam un busy timeout si bd in modul wall pentru a ajuta concurenta
    sqlite3_busy_timeout(db, 1000);
    if (sqlite3_exec(db, "PRAGMA journal_mode=WAL;", NULL, NULL, &err_msg) != SQLITE_OK) {
        fprintf(stderr, "Error setting WAL mode: %s\n", err_msg);
        sqlite3_free(err_msg);
    }

    // Pornim cu un table gol
    snprintf(sql, sizeof(sql), "DELETE FROM nod_%d;", id_nod);
    if (sqlite3_exec(db, sql, 0, 0, &err_msg) != SQLITE_OK) {
        if (strstr(err_msg, "no such table") == NULL) {
            printf("SQL delete error");
            sqlite3_free(err_msg);
            sqlite3_close(db);
            exit(1);
        }
        sqlite3_free(err_msg);
    } else {
        printf("Existing data from 'nod_%d' has been cleared.\n", id_nod);
    }

    //Creeaza tabela
    snprintf(sql, sizeof(sql), "CREATE TABLE IF NOT EXISTS nod_%d (key INTEGER PRIMARY KEY, value TEXT);", id_nod);
    if (sqlite3_exec(db, sql, 0, 0, &err_msg) != SQLITE_OK) {
        printf("SQL table error");
        sqlite3_free(err_msg);
        sqlite3_close(db);
        exit(1);
    }

    printf("Table 'nod_%d' has been created.\n", id_nod);
}
```

Tabelele finger table se actualizează automat la nodurile afectate de intrarea sau ieșirea din rețea a unui alt nod. De asemenea acestea se actualizează periodic pentru toate nodurile active. Pentru menținerea acestor tabele am utilizat funcțiile:

```
    void update_finger_table() {
    for (int i = 0; i < M; i++) {

        int successor = find_successor(node.start[i]);
        if (successor == node.port || successor <= 0) {
            printf("[upd_finger_table] Entry %d remains the same (port %d).\n", i, node.port);
            successor = node.port;
        }

        node.finger_table[i] = successor;
        printf("[upd_finger_table] Finger table[%d]: %d\n", i, successor);
    }
}

void fix_fingers() {
    static int next = 0;
    int start = (node.id + (1 << next)) % (1 << M);

    int successor = find_successor(start);
    if (node.finger_table[next] != successor) {
        printf("[fix_fingers] Updating finger[%d] from %d to %d\n", next, node.finger_table[next], successor);
        node.finger_table[next] = successor;
    }

    next = (next + 1) % M;
}
```

Un punct important în rețeaua chord este transferul de chei în urma părăsirii nodurilor sau după intrarea altor noduri în rețea. Pentru parăsirea voluntară am implementat urmatorul protocol: Nodul care urmează să părăsească rețeaua trimite UPDATE_S la predecesor și UPDATE_P la succesor și transferă cheile succesorului. La JOINUL unui nod în rețea următorul protocol este respectat: Nodul care va fi predecesorul nodului care intră în rețea va trimite mesaje de tip UPDATE_S, UPDATE_P noului nod. Noul nod va trimite o cere REQUEST_KEYS noului său succesor și va primi răspunsuri de tipul TRANSFER **cheie valoare**. Transferul cheilor se realizează în modul următor:

```
void transfer_keys_to_new_node(int new_node_port, int new_node_pred) {
    printf("[transfer_keys] Node %d transferring keys to port %d\n",node.id, new_node_port);

    int new_node_id = hash(new_node_port);
    int pred_new_node = hash(new_node_pred);

```

```
    printf("[transfer_keys] new_node_id = %d, pred_new_node = %d\n",new_node_id, pred_new_node);

    char select_cmd[256];
    snprintf(select_cmd, sizeof(select_cmd),"SELECT key, value FROM nod_%d;", node.id);

    char *error = NULL;
    char **rezultat = NULL;
    int rows, cols;

    pthread_mutex_lock(&mutex_db);
    if (sqlite3_get_table(db, select_cmd, &rezultat, &rows, &cols, &error) != SQLITE_OK) {
        fprintf(stderr, "[transfer_keys_to_new_node] SELECT error: %s\n", error);
        sqlite3_free(error);
        return;
    }

    int index = cols;
    for (int i = 0; i < rows; i++) {
        int k = atoi(rezultat[index]);        // key
        char *val = rezultat[index + 1];      // value
        index += cols;

        int key_hash = k; //pentru simplitate folosim insusi cheia ca hash

        if (
            // Caz normal(pred_new_node< new_node_id)
            ((pred_new_node < new_node_id) &&
             (key_hash > pred_new_node && key_hash <= new_node_id))
            ||
            // Caz wrap-around(pred_new_node > new_node_id)
            ((pred_new_node > new_node_id) &&
             (key_hash > pred_new_node || key_hash <= new_node_id))
             ||
             pred_new_node == new_node_id // Caz unde ramanem cu un singur nod
        ) {
            char msg[BUFFER_SIZE];
            snprintf(msg, sizeof(msg),"TRANSFER_KEY %d %s %d", k, val, new_node_port);

            send_update_message(new_node_port, msg);
            printf("[transfer_keys] Transferred key %d -> node %d OK.\n",k, new_node_port);
```

```
47
48              char delete_cmd[256];
49              snprintf(delete_cmd, sizeof(delete_cmd),"DELETE␣
                    FROM␣nod_%d␣WHERE␣key␣=␣%d;", node.id, k);
50
51              char *delete_error = NULL;
52              if (sqlite3_exec(db, delete_cmd, 0, 0, &
                    delete_error) != SQLITE_OK) {
53                  fprintf(stderr, "Error␣deleting␣key␣%d:␣%s\n"
                        , k, delete_error);
54                  sqlite3_free(delete_error);
55              }
56          }
57      }
58      sqlite3_free_table(rezultat);
59      pthread_mutex_unlock(&mutex_db);
60  }
```

Am implementat în acest proiect:

- mecanismul de stabilire a succesorului și a predecesorului. Voi trimite mesajul JOIN către nodurile existente. Acestea își vor actualiza succesorii și predecesorii.
- mecanismul de rutare a cheilor. Dacă cheia nu aparține nodului curent se transmite succesorilor până la găsirea nodului responsabil de aceasta.
- mecanismul de stabilizare. Nodurile trebuie să verifice în permanență dacă succesorii și predecesorii sunt valizi.
- implementarea funcționalităților DHT. Adăugarea, ștergerea și găsirea cheilor. Inclusiv mecanismul de transfer a cheilor.
- optimizarea cu finger table (o tabelă cu m intrări care ne ajută să facem salturi mai mari de la un nod la altul până când un nod află că o cheie este stocată în nodul imediat succesor).
- utilizarea bazelor de date pentru stocarea perechilor cheie-valoare.

5 Concluzii

Pot fi aduse îmbunătățiri aplicației precum:

- Reținerea unei liste de noduri de backup în cazul căderii simultană a mai multor noduri care se află "în linie"
- Salvarea cheilor în mai multe locații pentru restituirea acestora în cazul deconectării neașteptate a nodurilor

Sunt multe scenarii de utilizare în viața reală a DHT bazat pe Chord precum rețelele peer-to-peer, sisteme de indexare și stocare(BitTorrent), implementări blockchain etc.

6 Bibliografie

1. https://edu.info.uaic.ro/computer-networks/cursullaboratorul.php
2. https://edu.info.uaic.ro/computer-networks/files/NetEx/S12/ServerConcThread/servTcpConcTh2.c
3. https://courses.cs.duke.edu//cps212/spring15/15-744/S07/papers/chord.pdf
4. https://app.diagrams.net/
5. https://www.overleaf.com/learn/latex/Code_listing
6. https://en.wikipedia.org/wiki/Chord_(peer-to-peer)
7. https://docs.openssl.org/master/
8. https://www.sqlite.org/docs.html